Germán Vidal (Ed.)

Functional and (Constraint) Logic Programming

12th International Workshop, WFLP 2003
Valencia, Spain, June 12-13, 2003
Proceedings

Volume Editor

Germán Vidal
Departamento de Sistemas Informáticos y Computación (DSIC)
Universidad Politécnica de Valencia, Spain
Camino de Vera, s/n
E-46022 Valencia (Spain)
Email: gvidal@dsic.upv.es

Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming, WFLP'03
Valencia, Spain, June 12-13, 2003

ISBN:

Depósito Legal:

Impreso en España.

Technical Report DSIC-II/13/03,
http://www.dsic.upv.es
Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia, 2003.

Preface

This technical report contains the proceedings of the *International Workshop on Functional and (Constraint) Logic Programming* (WFLP 2003), held in Valencia (Spain) during June 12-13, 2003, at the Federated Conference on Rewriting, Deduction and Programming (RDP 2003). WFLP 2003 is the twelfth in the series of international meetings aimed at bringing together researchers interested in functional programming, (constraint) logic programming, as well as their integration. It promotes the cross-fertilizing exchange of ideas and experiences among researches and students from the different communities interested in the foundations, applications, and combinations of high-level, declarative programming languages and related areas. Previous WFLP editions have been held in Grado (Italy), Kiel (Germany), Benicàssim (Spain), Grenoble (France), Bad Honef (Germany), Schwarzenberg (Germany), Marburg (Germany), Rattenberg (Germany), and Karlsruhe (Germany).

The Program Committee of WFLP 2003 collected for each paper at least three reviews and held an electronic discussion on all papers during April 2003. Finally, the Program Committee selected 21 regular papers and one system description. In addition to the selected papers, the scientific program included invited lectures by Michael Rusinowitch (LORIA-INRIA-Lorraine, France) and Jan Małuszyński (Linköping University, Sweden). I would like to thank them for their willingness in accepting our invitation.

I would also like to thank all the members of the Program Committee and all the referees for their careful work in the review and selection process. Many sincere thanks to all authors who submitted papers and to all conference participants. I gratefully acknowledge all the institutions and corporations who have supported this conference. Finally, I express my gratitude to all members of the local Organizing Committee whose work has made the workshop possible.

Valencia
June 2003

Germán Vidal
Program Chair
WFLP 2003

Organization

Program Committee

María Alpuente	Technical University of Valencia, Spain
Sergio Antoy	Portland State University, USA
Annalisa Bossi	Università Ca' Foscari di Venezia, Italy
Olaf Chitil	University of York, UK
Rachid Echahed	Institut IMAG, France
Sandro Etalle	University of Twente, The Netherlands
Moreno Falaschi	Università di Udine, Italy
Michael Hanus	CAU Kiel, Germany
Yukiyoshi Kameyama	University of Tsukuba, Japan
Herbert Kuchen	University of Muenster, Germany
Michael Leuschel	University of Southampton, UK
Juan José Moreno-Navarro	Universidad Politécnica de Madrid, Spain
Ernesto Pimentel	University of Málaga, Spain
Mario Rodríguez-Artalejo	Universidad Complutense de Madrid, Spain
Germán Vidal	Technical University of Valencia, Spain

Organizing Committee

Elvira Albert	Santiago Escobar	César Ferri
José Hernández	Carlos Herrero	Pascual Julián
Marisa Llorens	Ginés Moreno	Javier Oliver
Josep Silva	Germán Vidal	Alicia Villanueva

Additional Referees

Elvira Albert	Frank Huch	Salvatore Orlando
Zena Ariola	Pascual Julián	Carla Piazza
Demis Ballis	Christoph Lembeck	Femke van Raamsdonk
Bernd Braßel	Pablo López	María José Ramírez
Ricardo Corin	Salvador Lucas	Guido Sciavicco
Francisco Durán	Massimo Marchiori	Clara Segura
Santiago Escobar	Julio Mariño	Jan-Georg Smaus
José Gallardo	Yasuhiko Minamide	Andrew Tolmach
Raffaella Gentilini	Angelo Montanari	Alberto Verdejo
Angel Herranz	Ginés Moreno	Alicia Villanueva
Teresa Hortalá-González	Roger Müller	
Zhenjiang Hu	Susana Muñoz	

Sponsoring Institutions

Departamento de Sistemas Informáticos y Computación (DSIC)

Universidad Politécnica de Valencia (UPV)

Generalitat Valenciana

Ministerio de Ciencia y Tecnología

European Association for Theoretical Computer Science (EATCS)

European Association for Programming Languages and Systems (EAPLS)

ColognNET: A Network of Excellence in Computational Logic

APPSEM Working Group

Table of Contents

Automated Analysis of Security Protocols

Michael Rusinowitch

LORIA, INRIA - Lorraine
Campus Scientifique, 54506 Vandoeuvre-Lès-Nancy Cedex, France
rusi@loria.fr

Cryptographic protocols such as SET, TLS, Kerberos have been developed to secure electronic transactions. However the design of such protocols often appears to be problematic even assuming that the cryptographic primitives are perfect, i.e. even assuming we cannot decrypt a message without the right key. An intruder may intercept messages, analyse them, modify them with low computing power and then carry out malevolent actions. This may lead to a variety of attacks such as well-known Man-In-The-Middle attacks.

Even in this abstract model, the so-called Dolev-Yao model, protocol analysis is complex since the set of states to consider is huge or infinite. One should consider messages of any size, infinite number of sessions. The interleaving of parallel sessions generates a large search space. Also when we try to slightly relax the perfect encryption hypothesis by taking into account some algebraic properties of operators then the task gets even more difficult.

We have developed in Cassis/LORIA team several translation and constraint solving techniques for automating protocol analysis in Dolev-Yao model and some moderate extensions of it. Protocol specifications as they can be found in white papers are compiled by the CASRUL system [6, 2] and then are passed on decision procedures for checking whether they are exposed to flaws [1].

Compiler: The CASRUL compiler performs a static analysis to check the executability of the protocol (i.e. whether each principal has enough knowledge to compose the messages he is supposed to send), and then compiles the protocol and intruder activities into an Intermediate Format based on first-order multiset rewriting. The Intermediate Format unambiguously defines an operational semantics for the protocol. Afterwards, different translators can be employed for translating the Intermediate Format into the input language of different analysis tools. The Intermediate Format can be also generated in a typed mode (the untyped one is the default), which leads to smaller search spaces at the cost of abstracting away type-flaws (if any) from the protocol. This compiler is the front-end of several efficient verification tools designed in the context of the European project AVISS [1]. It is open to connection with other tools. Compared to CAPSL, a related tool, CASRUL is able to handle protocols where a principal receives a cipher, say $\{Na\}_K$, and later receives the key K and then uses Na in some message. In our case, the principal will store

[1] This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-39252 AVISPA project

$\{Na\}_K$ and will decrypt it when he later receives the key. This is useful for e-commerce protocols such as non-repudiation protocols.

Constraint solving for verification. The detection of flaws in a hostile environment can be reduced to solving symbolic constraints in the message space. The intruder should send messages that comply with the protocol specification in order to get undetected by honest agents. After the last protocol step, the intruder should be able to derive the secret from the replies sent by the honest agents (and possibly using some initial knowledge).

For building messages the intruder can decompose compound messages, decrypt (or build) ciphers when he has the right key. Whether the intruder can compose some message from a given set of messages can be checked in polynomial time. On the other hand, whether there exists an attack on a protocol (in a given number of sessions) is an NP-complete problem [7]. The choice of messages to be sent by the intruder introduces non-determinism. Moreover, we have proved that when the protocol is subject to an attack, then there always exists an attack such that the sum of sizes of the messages exchanged between agents is linear w.r.t. the protocol size.

Experiments. The CASRUL implementation of intruder deduction rules is efficient. For a report on experiments see [2]. The CASRUL tool has a web-based graphical user-interface (accessible at the URL: `www.loria.fr/equipes/cassis/softwares/casrul/`). Among the 51 protocols in the Clark/Jacob library, 46 can be expressed in the tool specification language. Among the 51 protocols 35 are flawed, and CASRUL can find a flaw in 32 of them, including a previously unknown type flaw in Denning Sacco Protocol. The tool finds the 32 attacks in less than one minute (1.4 GHz processor).

Current extension. We have recently extended the framework and the complexity results to the case where messages may contain the XOR operator and where the Dolev-Yao intruder is extended by the ability to compose messages with the XOR operator (see [4]). This extension is non-trivial due to the complex interaction of the XOR properties such as associativity and commutativity and the standard Dolev-Yao intruder rules. The technical problems raised by the equational laws are somewhat related to those encountered in semantic unification. We have also considered an intruder equipped with the ability to exploit prefix properties of encryption algorithms based on cipher-block-chaining (CBC).

References

1. Alessandro Armando, David Basin, Mehdi Bouallagui, Yannick Chevalier, Luca Compagna, Sebastian Moedersheim, Michael Rusinowitch, Mathieu Turuani, Luca Vigano, Laurent Vigneron. The AVISS Security Protocols Analysis Tool - system description In *Int. Conference on Automated Verification*

'02, LNCS, Copenhagen, Denmark, July 27-31, 2002. System available at: `www.informatik.uni-freiburg.de/~softech/research/projects/aviss`

2. Y. Chevalier and L. Vigneron. A Tool for Lazy Verification of Security Protocols. In *Proc. of ASE'01*. IEEE CS Press, 2001.
3. Y. Chevalier and L. Vigneron. Automated Unbounded Verification of Security Protocols In *Int. Conference on Automated Verification '02*, LNCS, Copenhagen, Denmark, July 27-31, 2002
4. Y. Chevalier and R. Ksters and M. Rusinowitch and M. Turuani. An NP Decision Procedure for Protocol Insecurity with XOR to appear in LICS'2003, Ottawa, june 2003.
5. J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. `www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`
6. F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In *Proc. of LPAR'00*, LNCS 1955, pp. 131–160. Springer, 2000. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR 2000*, LNCS 1955, pages 131–160. Springer, 2000.
7. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 2001. long version in: Theoretical Computer Science A, 299 (2003), pp 451-475

On Integrating Rules into the Semantic Web (Abstract)

Jan Małuszyński

Dept. of Computer and Information Science,
Linköping University, S 581 83 Linköping, Sweden
janma@ida.liu.se

Key words: Semantic Web, XML, rules, logic programs

Introduction

Traditionally web pages have been created for human users. A question gaining recently wide interest is how to express the meaning of the web data so that the machines can reason about it, thus making possible more intelligent applications. In particular, this issue is addressed by the Semantic Web Activity (http://www.w3.org/2001/sw/) of the World Wide Web Consortium (W3C), which de facto sets standards for the web.

It is often argued (see e.g. [3]) that in web applications, such as web services, there is a need of "if-then" rules, which can be formally treated within various non-monotonic reasoning frameworks originating from logic programming. So far W3C did not propose yet any standards for such rules. In this context we discuss how the problem of placing and using rules on the web relates to the already existing W3C standards and proposals.

The presention will provide a brief outline of the W3C approach to the semantic web, and author's view on integrating LP-like rules in this framework.

The Semantic Web

The W3C approach to the semantic web aims at defining layered languages, (see e.g. [3]) to be used as standards for describing the syntax and the semantic aspects of data on the web. This general idea is sometimes referred to as the semantic web "tower".

The syntactic layer. The base language of the semantic web tower is XML [7], the dominant standard for data encoding and exchange on the web. Essentially the data represented in XML can be seen as labeled ordered trees. Such trees are encoded as documents with paranthesis structure marked by *tags*. Several mechanisms has been proposed for defining classes of documents. A widely used one is XML Schema [8], where the definitions are written in XML.

Adding Semantics to XML. The key issue is how to associate the meanings with XML documents. For a given class of XML documents we have thus to define a *semantic function* into some domain. As an example consider the the XML Schema language. It consists of XML documents with well defined meanings: each of them specifies a class of XML documents.

The semantic function provides a reference basis for development of the software for manipulating objects represented by the language specified by a given XML schema. An example of such a software are XML Schema validators: for a given schema c a validator can check whether a given XML document is in the set defined by c or not.

Web Ontology Languages. The semantics of data can be provided by linking the data with a description of the application domain, called *ontology.* Intuitively, an ontology specifies a vocabulary for denoting objects in the domain, classes of objects, and binary relations between classes, in particular the sub-class relation. Various kinds of Description Logics (DL) has been proposed as formalisms for expressing ontologies. With this formalisation ontologies are specified as sets of DL axioms.

In the context of the Semantic Web ontologies are to be specified in XML. For this one needs *ontology languages* which make it possible to represent formulae of Description Logics in XML. The Description Logic semantics provides then the meaning of the ontologies specified in the ontology language. Automatic reasoning systems for the DL, such as the FaCT reasoner [4], are based on the semantics of the Description Logics and make it possible to reason about the defined ontologies. Several ontology languages for the Semantic Web has been proposed. The most recent one, the Web Ontology Language OWL [6], is related to its predecessor DAML+OIL [1].

The Description Logics are used for modeling of application domains. The basic modeling primitives are classes and the subclass relation. Classes of a given ontology could be linked to classes of XML elements representing objects of the application. The issue of relating ontologies to XML schemata is discussed in [5].

Adding Rules to the Semantic Web

A rule language with a well defined semantics can be made accessible on the web by following the approach of ontology languages. After having defined a standard representation of the rules in XML one can place on the web a rule engine based on the semantics of the rule language. Links to prototypes of this kind are being developed within Rule Markup Initiative (http://www.ruleml.org). However, as pointed out in [3], there is a consensus that rulebases on the semantic web should refer to the vocabularies introduced by ontologies. The question is thus how to integrate rules and ontologies in a semantically coherent way. Focusing on if-then rules formalised as clauses of logic programs, the question is how to integrate logic programs with ontologies. The approach to this problem presented in [3] identifies restrictions under which ontology axioms expressed in Description Logic can be compiled into Datalog clauses, thus providing a basis for integration of ontologies and rules. In this approach the description of the application domain and the application specific rules, two conceptually different aspects, are expressed in the same formalism. An alternative approach to integration would be to view classes of a given ontology as types for rules. Such an approach would separate both aspect mentioned above. The integration would be achieved by type checking of the rules wrt the separately specified types (or by inferring types for given rules). Referring to the above mentioned relation between ontologies and XML Schema, we will discuss how our previous work on types for constraint logic programs [2] can be adapted to the XML context.

References

1. DAML+OIL, 2001. http://www.daml.org/2001/03/daml+oil-index.html.
2. W. Drabent, J. Maluszynski, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–610, 2002.
3. Benjamin N. Grosof and Ian Horrocks. Description logic programs: Combining logic programs with description logic. citeseer.nj.nec.com/551089.html.
4. Ian Horrocks. Fact and iFaCT. In *Description Logics*, 1999. Proc. of the Int. Description Logics Workshop (DL'99).
5. M. Klein, D. Fensel, F. van Harmelen, and I. Horrocks. The relation between ontologies and schema-languages: Translating oil-specifications in xml-schema. In *Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods*, 2000. http://delicias.dia.fi.upm.es/WORKSHOP/ECAI00/7.pdf.
6. OWL Web Ontology Language Reference, W3C Working Draft, 2003. http://www.w3.org/TR/owl-ref.
7. Extensible Markup Language (XML) 1.0 (Second Edition), 2000. http://www.w3.org/TR/REC-xml.
8. XML Schema Part 1: Structures, 2001. http://www.w3.org/TR/xmlschema-1.

Abstract Correction of Functional Programs*

M. Alpuente[1], D. Ballis[2], S. Escobar[1], M. Falaschi[2], and S. Lucas[1]

[1] DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain. {alpuente,sescobar,slucas}@dsic.upv.es.
[2] Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. {demis,falaschi}@dimi.uniud.it.

Abstract. DEBUSSY is an (abstract) declarative diagnosis tool for functional programs which are written in OBJ style. The debugger does not require the user to either provide error symptoms in advance or answer any question concerning program correctness. In this paper, we formalize an inductive learning methodology for repairing program bugs in OBJ-like programs. Correct program rules are automatically synthesized from examples which might be generated as an outcome by the DEBUSSY diagnoser.

1 Introduction

This paper is motivated by the fact that the debugging support for functional languages in current systems is poor [25], and there are no general purpose, good semantics-based debugging tools available. Traditional debugging tools for functional programming languages consist of tracers which help to display the execution [20, 6, 12, 19] but which do not enforce program correctness adequately as they do not provide means for finding nor reparing bugs in the source code w.r.t. the intended program semantics. This is particularly dramatic for equational languages such as those in the OBJ family, which includes OBJ3, CafeOBJ and Maude.

Abstract diagnosis of functional programs [2] is a declarative diagnosis framework extending the methodology of [7], which relies on (an approximation of) the immediate consequence operator $T_{\mathcal{R}}$, to identify bugs in functional programs. Given the intended specification $\mathcal{I}$ of the semantics of a program $\mathcal{R}$, the debugger checks the correctness of $\mathcal{R}$ by a single step of the abstract immediate consequence operator $T^{\kappa}_{\mathcal{R}}$, where the abstraction function κ stands for $depth(k)$ cut [7]. Then, by a simple static test, the system can determine all the rules which are wrong w.r.t. a particular abstract property. The framkweork is goal independent and does not require the determination of symptoms in advance. This is in contrast with traditional, semi-automatic debugging of functional programs [18, 17, 24], where the debugger tries to locate the node in an execution tree which is ultimately responsible for a visible bug symptom. This is done by

* This work has been partially supported by CICYT under grant TIC2001-2705-C03-01.

asking the user, which assumes the role of the oracle. When debugging real code, the questions are often textually large and may be difficult to answer.

In this paper, we endow the functional debugging method of [2] with a bug-correction program synthesis methodology which, after diagnosing the buggy program, tries to correct the erroneous components of the wrong code automatically. The method uses unfolding in order to discriminate positive from negative examples (resp. uncovered and incorrect equations) which are automatically produced as an outcome by the diagnoser. Informally, our correction procedure works as follows. Starting from an *overly general* program (that is, a program which covers all the positive examples as well as some negative ones), the algorithm unfolds the program and deletes program rules until reaching a suitable specialization of the original program which still covers all the positive examples and does not cover any negative one. Both, the example generation as well as the top-down correction processes, exploit some properties of the abstract interpretation framework of [2] which they rely on. Let us emphasize that we do not require any particular condition on the class of the programs which we consider. This is particularly convenient in this context, since it should be undesirable to require strong properties, such as termination or confluence, to a buggy program which is known to contain errors.

We would like to clarify the contributions of this paper w.r.t. [1], where a different unfolding-based correction method was developed which applies to synthetizing multiparadigm, functional-logic programs from a set of positive and negative examples. First, the method for automatically generating the example sets is totally new. In [1] (abstract) non-ground examples were computed as the outcome of an abstract debugger based on the *loop-check* techniques of [3], whereas now we compute (concrete) ground examples after a *depth-k* abstract diagnosis phase [7] which is conceptually much simpler and allows us to compute the example sets more efficiently. Regarding the top-down correction algorithm, the one proposed in this paper significantly improves the method in [1]. We have been able to devise an abstract technique for testing the "overgenerality" applicability condition, which saves us from requiring program termination or the slightly weaker condition of *μ-termination* (termination of context-sensitive rewriting [15]). Finally, we have been able to demonstrate the correctness of the new algorithm for a much larger class of programs, as we do not even need confluence whereas [1] applies only to inductively sequential or noetherian and constructor-based programs (depending on the lazy/eager narrowing strategy chosen).

The debugging methodology which we consider can be very useful for a functional programmer who wants to debug a program w.r.t. a preliminary version which was written with no efficiency concern. Actually, in software development a specification may be seen as the starting point for the subsequent program development, and as the criterion for judging the correctness of the final software product. Therefore, a debugging tool which is able to locate bugs in the user's program and correct the wrong code becomes also important in this context. In general, it also happens that some parts of the software need to be improved during the software life cycle, e.g. for getting a better performance. Then the old programs (or large parts of them) can be usefully (and automatically) used as a specification of the new ones. For instance, the executability of OBJ specifications supports prototype-driven incremental development methods [13].

The rest of the paper is organized as follows. Section 2 summarizes some preliminary definitions and notations. Section 3 recalls the abstract diagnosis framework for functional programs of [2]. Section 4 formalizes the correction problem in this framework. Section 5 illustrates the example generation methodology. Section 6 presents the top-down correction method together with some examples and correctness results. Section 7 concludes.

2 Preliminaries

Term rewriting systems provide an adequate computational model for functional languages. In this paper, we follow the standard framework of term rewriting (see, [4, 14]). For simplicity, definitions are given in the one-sorted case. The extension to many–sorted signatures is straightforward, see [21]. In the paper, syntactic equality of terms is represented by $\equiv$. By $\mathcal{V}$ we denote a countably infinite set of variables and Σ denotes a set of function symbols, or signature, each of which has a fixed associated arity. $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$ denote the non-ground word (or term) algebra and the word algebra built on $\Sigma \cup \mathcal{V}$ and Σ, respectively. $\mathcal{T}(\Sigma)$ is usually called the Herbrand universe ($\mathcal{H}_\Sigma$) over Σ and will be simply denoted by $\mathcal{H}$. $\mathcal{B}$ denotes the Herbrand base, namely the set of all ground equations which can be built with the elements of $\mathcal{H}$. A Σ-equation $s = t$ is a pair of terms $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$, or *true*.

Terms are viewed as labelled trees in the usual way. Positions are represented by sequences of natural numbers denoting an access path in a term. Given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t which are rooted by symbols in S. $t|_u$ is the subterm at the position u of t. $t[r]_u$ is the term t with the subterm at the position u replaced with r. By $Var(s)$ we denote the set of variables occurring in the syntactic object s, while $[s]$ denotes the set of ground instances of s. A *fresh* variable is a variable that appears nowhere else.

A *substitution* is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\mathcal{T}(\Sigma, \mathcal{V})$. A substitution θ is more general than σ, denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ. We write $\theta_{\upharpoonright s}$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s. The *empty substitution* is denoted by ϵ. A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1}, such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. An equation set E is unifiable, if there exists ϑ such that, for all $s = t$ in E, we have $s\vartheta \equiv t\vartheta$, and ϑ is called a *unifier* of E. We let $mgu(E)$ denote 'the' *most general unifier* of the equation set E [16].

A *term rewriting system* (TRS for short) is a pair $(\Sigma, \mathcal{R})$, where $\mathcal{R}$ is a finite set of reduction (or rewrite) rule schemes of the form $\lambda \to \rho$, λ, $\rho \in \mathcal{T}(\Sigma, \mathcal{V})$, $\lambda \notin \mathcal{V}$ and $Var(\rho) \subseteq Var(\lambda)$. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$ and call $\mathcal{R}$ the program. For TRS $\mathcal{R}$, $r \ll \mathcal{R}$ denotes that r is a new variant of a rule in $\mathcal{R}$ such that r contains only *fresh* variables. Given a TRS $(\Sigma, \mathcal{R})$, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma := \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} := \{f \mid f(t_1, \ldots, t_n) \to r \in \mathcal{R}\}$ and $\mathcal{C} := \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. The elements of $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called *constructor terms*, while elements in $\mathcal{T}(\mathcal{C})$ are called *values*. A *pattern* is a term of the form $f(\bar{d})$ where $f/n \in \mathcal{D}$ and $\bar{d}$ is a n-tuple of constructor terms. A rewrite step is the application of a rewrite rule to an expression. A term s *rewrites* to a term t via $r \ll \mathcal{R}$, $s \to_r t$, if there exist $u \in O_\Sigma(s)$, $r \equiv \lambda \to \rho$, and substitution σ such that $s|_u \equiv \lambda\sigma$ and $t \equiv s[\rho\sigma]_u$. We say that $\mathcal{S} := t_0 \to_{r_0} t_1 \to_{r_1} t_2 \ldots \to_{r_{n-1}} t_n$ is a *rewrite sequence* from term t_0 to term t_n. When no confusion can arise, we will omit any subscript (i.e. $s \to t$). A term s is a *normal form*, if there is no term t with $s \to_{\mathcal{R}} t$. t is the normal form of s if $s \to_{\mathcal{R}}^* t$ and t is a normal form (in symbols $s \to_{\mathcal{R}}^! t$).

The narrowing mechanism is commonly applied to evaluate terms containing variables. Narrowing non-deterministically instantiates variables so that a rewrite step is enabled. This is done by computing mgu's. Formally, $s \overset{\sigma}{\leadsto}_r t$ is a *narrowing step* via $r \ll \mathcal{R}$, if there exist $p \in O_\Sigma(s)$ and $r \equiv \lambda \to \rho$ such that $\sigma = mgu(\{\lambda = s|_p\})$ and $t \equiv s[\rho]_p\sigma$.

3 Denotation of functional programs

In this section we first recall the semantic framework introduced in [2]. We will provide a finite/angelic relational semantics [8], given in fixpoint style, which associates an input-output relation to a program, while intermediate computation steps are ignored. Then, we formulate an abstract semantics which approximates the evaluation semantics of the program.

In order to formulate our semantics for term rewriting systems, the usual Herbrand base is extended to the set of all (possibly) non-ground equations [10, 11]. $\mathcal{H}_\mathcal{V}$ denotes the *$\mathcal{V}$-Herbrand universe* which allows variables in its elements, and is defined as $\mathcal{T}(\Sigma, \mathcal{V})/_{\cong}$, where $\cong$ is the equivalence relation induced by the preorder $\leq$ of "relative generality" between terms, i.e. $s \leq t$ if there exists σ s.t. $t \equiv \sigma(s)$. For the sake of simplicity, the elements of $\mathcal{H}_\mathcal{V}$ (equivalence classes) have the same representation as the elements of $\mathcal{T}(\Sigma, \mathcal{V})$ and are also called terms. $\mathcal{B}_\mathcal{V}$ denotes the *$\mathcal{V}$-Herbrand base*, namely, the set of all equations $s = t$ modulo variance, where $s, t \in \mathcal{H}_\mathcal{V}$. A subset of $\mathcal{B}_\mathcal{V}$ is called a $\mathcal{V}$-Herbrand interpretation. We assume that the equations in the denotation are renamed apart. The ordering $\leq$ for terms is extended to equations in the obvious way, i.e. $s = t \leq s' = t'$ iff there exists σ s.t. $\sigma(s) = \sigma(t) \equiv s' = t'$.

3.1 Concrete semantics

The considered concrete domain $\mathbb{E}$ is the lattice of $\mathcal{V}$-Herbrand interpretations, i.e., the powerset of $\mathcal{B}_\mathcal{V}$ ordered by set inclusion.

In the sequel, a semantics for program $\mathcal{R}$ is a $\mathcal{V}$-Herbrand interpretation. Since in functional programming, programmers are generally concerned with computing values (ground constructor normal forms), the semantics which is usually considered is $Sem_{\mathsf{val}}(\mathcal{R}) := \{s = t \mid s \rightarrow^{!}_{\mathcal{R}} t, t \in \mathcal{T}(\mathcal{C})\}$. Sometimes, we will call *proof* of equation $s = t$, a rewrite sequence from term s to value t.

Following [8], in order to formalize our evaluation semantics via fixpoint computation, we consider the following immediate consequence operator.

Definition 1. [2] *Let $\mathcal{I}$ be a Herbrand interpretation, $\mathcal{R}$ be a TRS. Then,*

$$T_{\mathcal{R}}(\mathcal{I}) = \{t = t \mid t \in \mathcal{T}(\mathcal{C})\} \cup \{s = t \mid r = t \in \mathcal{I}, s \rightarrow_{\mathcal{R}} r\}.$$

The following proposition is immediate.

Proposition 1. [2] *Let $\mathcal{R}$ be a TRS. The $T_{\mathcal{R}}$ operator is continuous on $\mathbb{E}$.*

Definition 2. [2] *The least fixpoint semantics of a program $\mathcal{R}$ is defined as $\mathcal{F}_{\mathsf{val}}(\mathcal{R}) = T_{\mathcal{R}} \uparrow \omega$.*

Example 1. Suppose you toss a coin after having chosen one of its faces. If the face revealed after the coin flip is the predicted one, you win a prize. The problem can be modeled by the following specification I (written in OBJ-like syntax):

```
obj GAMESPEC is
sorts Nat Reward .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op prize : -> Reward .
   op sorry-no-prize : -> Reward .
   op coinflip : Nat -> Reward .
   op win? : Nat -> Reward .
   var X : Nat .
   eq coinflip(X) = win?(X) .
   eq win?(s(s(X))) = sorry-no-prize .
   eq win?(s(0)) = prize .
   eq win?(0) = sorry-no-prize .
endo
```

Face values are expressed by naturals `0` and `s(0)`; besides, specification $\mathcal{I}$ tells us that we win the prize at stake (expressed by the constructor *prize*), if the revealed face is $s(0)$, while we get no prize whenever the revealed face is equal to `0`.

The associated least fixpoint semantics is

$$\begin{aligned}\mathcal{F}_{\mathsf{val}}(\mathcal{I}) = \{ & \mathtt{prize} = \mathtt{prize}, \mathtt{sorry\text{-}no\text{-}prize} = \mathtt{sorry\text{-}no\text{-}prize}, \\ & \mathtt{win?(0)} = \mathtt{sorry\text{-}no\text{-}prize}, \mathtt{win?(s(0))} = \mathtt{prize}, \\ & \mathtt{win?(s(s(X))} = \mathtt{sorry\text{-}no\text{-}prize}, \mathtt{coinflip(0)} = \mathtt{sorry\text{-}no\text{-}prize}, \\ & \mathtt{coinflip(s(0))} = \mathtt{prize}, \mathtt{coinflip(s(s(X))} = \mathtt{sorry\text{-}no\text{-}prize}\}.\end{aligned}$$

The following result establishes the equivalence between the (fixpoint) semantics computed by the $T_{\mathcal{R}}$ operator and the evaluation semantics $Sem_{\mathsf{val}}(\mathcal{R})$.

Theorem 1 (soundness and completeness). [2] *Let $\mathcal{R}$ be a TRS. Then, $Sem_{\mathsf{val}}(\mathcal{R}) = \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.*

3.2 Abstract semantics

Starting from the concrete fixpoint semantics of Definition 2, we give an abstract semantics which approximates the concrete one by means of abstract interpretation techniques. In particular, we will focus our attention on abstract interpretations achieved by means of a *depth*(k) cut [7], which allows to finitely approximate an infinite set of computed equations.

First of all we define a *term abstraction* as a function $/_k : (\mathcal{T}(\Sigma, \mathcal{V}), \leq) \to (\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}}), \leq)$ which cuts terms having a depth greater than k. Terms are cut by replacing each subterm rooted at depth k with a new variable taken from the set $\hat{\mathcal{V}}$ (disjoint from $\mathcal{V}$). *depth*(k) terms represent each term obtained by instantiating the variables of $\hat{\mathcal{V}}$ with terms built over $\mathcal{V}$. Note that $/_k$ is finite. We denote by $T/_k$ the set of *depth*(k) terms $(\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}})/_k)$. We choose as abstract domain $\mathbb{A}$ the set $\mathcal{P}(\{a = a' \mid a, a' \in T/_k\})$ ordered by the Smyth's extension of ordering $\leq$ to sets, i.e. $X \leq_S Y$ iff $\forall\, y \in Y\ \exists\, x \in X : x \leq y$. Thus, we can lift term abstraction $/_k$ to a Galois Insertion of $\mathbb{A}$ into $\mathbb{E}$ by defining

$$\kappa(E) := \{s/_k = t/_k \mid s = t \in E\}$$
$$\gamma(A) := \{s = t \mid s/_k = t/_k \in A\}$$

Now we can derive the optimal abstract version of $T_{\mathcal{R}}$ simply as $T^{\kappa}_{\mathcal{R}} := \kappa \circ T_{\mathcal{R}} \circ \gamma$ and define the abstract semantics of program $\mathcal{R}$ as the least fixpoint of this (obviously) continuous operator, i.e. $\mathcal{F}^{\kappa}_{\mathsf{val}}(\mathcal{R}) := T^{\kappa}_{\mathcal{R}} \uparrow \omega$. Since $/_k$ is finite, we are guaranteed to reach the fixpoint in a finite number of steps, that is, there exists a finite natural number h such that $T^{\kappa}_{\mathcal{R}} \uparrow \omega = T^{\kappa}_{\mathcal{R}} \uparrow h$. Abstract interpretation theory assures that $T^{\kappa}_{\mathcal{R}} \uparrow \omega$ is the best correct approximation of $\mathit{Sem}_{\mathsf{val}}(\mathcal{R})$. Correct means $\mathcal{F}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \leq_S \kappa(\mathit{Sem}_{\mathsf{val}}(\mathcal{R}))$ and best means that it is the maximum w.r.t. $\leq_S$.

By the following proposition, we provide a simple and effective mechanism to compute the abstract fixpoint semantics.

Proposition 2. [2] *For $k > 0$, the operator $T^{\kappa}_{\mathcal{R}} : T/_k \times T/_k \to T/_k \times T/_k$ holds the property $\widetilde{T}^{\kappa}_{\mathcal{R}}(X) \leq_S T^{\kappa}_{\mathcal{R}}(X)$ w.r.t. the following operator:*

$$\widetilde{T}^{\kappa}_{\mathcal{R}}(X) = \kappa(B) \cup \{\sigma(u[l]_p)/_k = t \mid u = t \in X, p \in O_{\Sigma\cup\mathcal{V}}(u), \\ l \to r \ll \mathcal{R}, \sigma = mgu(u|_p, r)\}$$

Definition 3. [2] *The effective abstract least fixpoint semantics of a program $\mathcal{R}$ is defined as $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) = \widetilde{T}^{\kappa}_{\mathcal{R}} \uparrow \omega$.*

Proposition 3 (Correctness). [2] *Let $\mathcal{R}$ be a TRS and $k > 0$.*

1. $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \leq_S \kappa(\mathcal{F}_{\mathsf{val}}(\mathcal{R})) \leq_S \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.
2. *For every $e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ such that $Var(e) \cap \hat{\mathcal{V}} = \emptyset$, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.*

Example 2. Consider again the specification in Example 1. Its effective abstract least fixpoint semantics for $\kappa = 3$ (without considering symbol `win?`) becomes

$$\widetilde{\mathcal{F}}^{3}_{\mathsf{val}}(\mathcal{I}) = \{\ \texttt{prize} = \texttt{prize}, \texttt{sorry-no-prize} = \texttt{sorry-no-prize}, \\ \texttt{coinflip(0)} = \texttt{sorry-no-prize}, \texttt{coinflip(s(0))} = \texttt{prize}, \\ \texttt{coinflip(s(s(}\hat{\texttt{X}}\texttt{)))} = \texttt{sorry-no-prize}\}.$$

4 The Correction Problem

The problem of repairing a faulty functional program can be addressed by using inductive learning techniques guided by appropriate examples. Roughly speaking, given a wrong program and two example sets specifying positive (pursued) and negative (not pursued) computations respectively, our correction scheme aims at synthesizing a set of program rules that replaces the wrong ones in order to deliver a corrected program which is "consistent" w.r.t. the example sets [5]. More formally, we can state the correction problem as follows.

Problem formalization. Let $\mathcal{R}$ be a TRS, E^+ and E^- be two finite sets of equations such that

1. $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{I})$;
2. $E^- \cap (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I})) \neq \emptyset$.

The correction problem consists in constructing a TRS $\mathcal{R}^c$ satisfying the following requirements

1. $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{R}^c)$;
2. $E^- \cap Sem_{\mathsf{val}}(\mathcal{R}^c) = \emptyset$.

Equations in E^+ (resp. E^-) are called *positive* (resp. *negative*) examples. The TRS $\mathcal{R}_c$ is called *correct* program. Note that by construction positive and negative example sets are disjoint, which permits to drive the correction process towards a discrimination between E^+ and E^-.

Deductive and Inductive Learners. The automatic search for a new rule in an induction process can be performed either bottom-up (i.e. from an overly specific rule to a more general) or top-down (i.e. from an overly general rule to a more specific). There are some reasons to prefer the top-down or *backward reasoning* process to the bottom–up or *forward reasoning* process [9]. On the one hand, it eliminates the need for navigating through all possible logical consequences of the program. On the other hand, it integrates inductive reasoning with the deductive process, so that the derived program is guaranteed to be correct. Unfortunately, it is known that the deductive process alone (i.e. unfolding) does not generally suffice for coming up with the corrected program (unless the program is "overly general", i.e. it covers all the positive examples) and inductive generalization techniques are necessary [9, 22, 23].

In [1], we presented a general bottom-up inductive generalization methodology along with a top-down inductive correction method. The former transforms a given program into a program which is "overly general" w.r.t. an example set, so that the latter can be applied. Therefore, in the sequel we only formalize an efficient top-down correction methodology, which we prove to be correct for a wider class of TRSs than the method which could be naïvely obtained by particularizing [1] to the functional setting.

5 How to generate example sets automatically

Before giving a constructive method to derive a correct program, we present a simple methodology for automatically generating example sets, so that the user does not need to provide error symptoms, evidences or other kind of information which would require a good knowledge of the program semantics that she probably lacks.

In the following, we observe that we can easily compute "positive" equations, i.e. equations which appear in the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$, since all equations in $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$ not containing variables in $\hat{\mathcal{V}}$ belong to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$, as stated in the following lemma.

Lemma 1. *Let $\mathcal{I}$ be a TRS and $E_P := \{e | e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \wedge Var(e) \cap \hat{\mathcal{V}} = \emptyset\}$. Then, $E_P \subseteq Sem_{\mathsf{val}}(\mathcal{I})$.*

Now, by exploiting the information in $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$ and $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$, we can also generate a set of "negative" equations which belong to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{R})$ of the wrong program $\mathcal{R}$ but not to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$ of the specification $\mathcal{I}$.

Lemma 2. *Let $\mathcal{R}$ be a TRS, $\mathcal{I}$ be a specification of the intended semantics and $E_N := \{e | Var(e) \cap \hat{\mathcal{V}} = \emptyset \wedge \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \not\leq_S \{e\} \wedge e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})\}$. Then, $E_N \subseteq (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I}))$.*

Starting from sets E_P and E_N, we construct the positive and negative example sets E^+ and E^- which we use for the correctness process, by considering the restriction of E_P and E_N to examples of the form $l = c$ where l is a pattern and c is a value, i.e. a term formed only by constructor symbols and variables. The motivation for this is twofold. On the one hand, it allows us to ensure correctness of the correction algorithm without any further requirements on the class of programs which we consider. On the other hand, by considering these "data" examples, the inductive process becomes independent from the extra auxiliary functions which might appear in $\mathcal{I}$, since we start synthesizing directly from data structures.

The sets E^+ and E^- are defined as follows.

$$\begin{aligned} E^+ &= \{l = c \mid f(t_1, \ldots, t_n) = c \in E_P \wedge f(t_1, \ldots, t_n) \equiv l \text{ is a pattern } \wedge \\ &\qquad\qquad \wedge\, c \in \mathcal{T}(\mathcal{C}) \wedge f \in \Sigma_{\mathcal{R}}\} \\ E^- &= \{l = c \mid l = c \in E_N \wedge l \text{ is a pattern } \wedge c \in \mathcal{T}(\mathcal{C})\} \end{aligned}$$

where $\Sigma_{\mathcal{R}}$ is the signature of program $\mathcal{R}$.

In the sequel, the function which computes the sets E^+ and E^-, according to the above description, is called ExampleGeneration($\mathcal{R}, \mathcal{I}$).

6 Program correction via example-guided unfolding

In this section we present a basic top-down correction method which is based on the so-called *example-guided unfolding* [5], which is able to specialize a program by applying unfolding and deletion of program rules until coming up with a correction. The top-down correction process is "guided" by the examples, in the sense that transformation steps focus on discriminating positive from negative examples. In order to successfully apply the method, the semantics of the program to be specialized must include the positive example set E^+ (that is, $E^+ \subseteq \mathit{Sem}_{\mathsf{val}}(\mathcal{R})$). Programs satisfying this condition are called *overly general* (w.r.t. E^+).

The over-generality condition is not generally decidable, as we do not impose program termination [1]. Fortunately, when we consider the abstract semantics framework of [2] we are able to ascertain a useful sufficient condition to decide whether a program is overly general, even if it does not terminate. The following proposition formalizes our method.

Proposition 4. *Let $\mathcal{R}$ be a TRS and E^+ be a set of positive examples. If, for each $e \in E^+$, there exists $e' \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ s.t.*

1. *$e' \leq e$;*
2. *$\mathit{Var}(e') \cap \hat{\mathcal{V}} = \emptyset$;*

then, $\mathcal{R}$ is overly general w.r.t. E^+.

Now, by exploiting Proposition 4, it is not difficult to figure out a procedure OverlyGeneral$(\mathcal{R}, E)$ testing this condition w.r.t. a program $\mathcal{R}$ and a set of examples E, e.g. a boolean function returning *true* if program $\mathcal{R}$ is overly general w.r.t. E and *false* otherwise.

6.1 The unfolding operator

Informally, *unfolding* a program $\mathcal{R}$ w.r.t. a rule r delivers a new specialized version of $\mathcal{R}$ in which the rule r is replaced with new rules obtained from r by performing a narrowing step on the rhs of r.

Definition 4 (rule unfolding). *Given two rules $r_1 \equiv \lambda_1 \to \rho_1$ and r_2, we define the* rule unfolding *of r_1 w.r.t. r_2 as $\mathcal{U}_{r_2}(r_1) = \{\lambda_1\sigma \to \rho' \mid \rho_1 \overset{\sigma}{\rightsquigarrow}_{r_2} \rho'\}$.*

Definition 5 (program unfolding). *Given a TRS $\mathcal{R}$ and a rule $r \ll \mathcal{R}$, we define the* program unfolding *of r w.r.t. $\mathcal{R}$ as follows $\mathcal{U}_{\mathcal{R}}(r) = \big(\mathcal{R} \cup \bigcup_{r' \in \mathcal{R}} \mathcal{U}_{r'}(r)\big) \setminus \{r\}$.*

Note that, by Definition 5, for any TRS $\mathcal{R}$ and rule $r \ll \mathcal{R}$, r is never in $\mathcal{U}_{\mathcal{R}}(r)$.

Definition 6. *Let $\mathcal{R}$ be a TRS, r be a rule in $\mathcal{R}$. The rule r is* unfoldable *w.r.t. $\mathcal{R}$ if $\mathcal{U}_{\mathcal{R}}(r) \neq \mathcal{R} \setminus \{r\}$.*

Now, we are ready to prove that the "transformed" semantics, obtained after applying the unfolding operator to a given program $\mathcal{R}$, still contains the semantics of $\mathcal{R}$. In symbols, $\mathit{Sem}_{\mathsf{val}}(\mathcal{R}) \subseteq \mathit{Sem}_{\mathsf{val}}(\mathcal{U}_{\mathcal{R}}(r))$, where r is an unfoldable rule. We call this property *unfolding correctness*.

The following auxiliary Lemmata are instrumental for the proof.

Lemma 3. *Let $r_1 \equiv \lambda_1 \to \rho_1$ and r_2 be two rules. Let $t_0, t_1, t_2 \in \tau(\Sigma \cup \mathcal{V})$. Then, $t_0 \to_{r_1,p_1} t_1 \to_{r_2,p_2} t_2$, where $p_2 \in O_\Sigma(t_1)$ corresponds to some position $p \in O_\Sigma(\rho_1)$, iff $t_0 \to_{r',p_1} t_2$, where $r' \in \mathcal{U}_{r_2}(r_1)$.*

In the following we denote the length of a rewrite sequence $\mathcal{S}$ by $|\mathcal{S}|$.

Lemma 4. *Let $\mathcal{R}$ be a TRS, $r \ll \mathcal{R}$ be an unfoldable rule and $\mathcal{R}' = \mathcal{U}_\mathcal{R}(r)$. Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and $c \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. Then,*

1. *if $\mathcal{S}$ is a rewrite sequence from t to c in $\mathcal{R}$, then there exists a rewrite sequence $\mathcal{S}'$ from t to c in $\mathcal{R}'$;*
2. *if r occurs in $\mathcal{S}$, then $|\mathcal{S}'| < |\mathcal{S}|$.*

The following corollary establishes the correctness of unfolding.

Corollary 1 (unfolding correctness). *Let $\mathcal{R}$ be a TRS, $r \ll \mathcal{R}$ be an unfoldable rule and $\mathcal{R}' = \mathcal{U}_\mathcal{R}(r)$. Let $e \equiv (l = c)$ be an equation such that $l \in \mathcal{T}(\Sigma, \mathcal{V})$ and $c \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. Then, if $e \in Sem_{\mathsf{val}}(\mathcal{R})$, then $e \in Sem_{\mathsf{val}}(\mathcal{R}')$.*

6.2 The top-down correction algorithm

Basically, the idea behind the basic correction algorithm is to eliminate rules from the program in order to get rid of the negative examples without losing the derivations for the positive ones. Clearly, this cannot be done naïvely removing program rules, since sometimes a rule is used to prove both a positive and a negative example. So, before applying deletion, we need to specialize programs in order to ensure that the deletion phase only affects those program rules which are not necessary for proving the positive examples. This specialization process is carried out by means of the unfolding operator of Definition 5. Considering this operator for specialization purposes has important advantages. First, positive examples are not lost by repeatedly applying the unfolding operator, since unfolding preserves the proper semantics (see Corollary 1). Moreover, the nature of unfolding is to "compile" rewrite steps into the program, which allows us to shorten and discriminate among the rewrite rules which occur in the proofs of the positive and negative examples.

Figure 1 shows the correction algorithm, called TDCorrector, which takes as input a program $\mathcal{R}$ and a specification of the intended semantics $\mathcal{I}$, also expressed as a program. First, TDCorrector computes the example sets E^+ and E^- by means of ExampleGeneration, following the method presented in Section 5. Then, it checks whether program $\mathcal{R}$ is overly general following the scheme of Proposition 4, and finally it enters the main correction process.

This last phase consists of a main loop, in which we perform an unfolding step followed by a rule deletion until no negative example is covered (approximated) by the abstract semantics of the current transformed program $\mathcal{R}_n$. This amounts to say that no negative example belongs to the concrete semantics of $\mathcal{R}_n$. We note that the **while** loop guard is decidable, as the abstract semantics is finitely computable. Note the deep difference w.r.t. the algorithm of [1], where

```
procedure TDCorrector($\mathcal{R}$, $\mathcal{I}$)
  $(E^+, E^-) \leftarrow$ ExampleGeneration($\mathcal{R}$, $\mathcal{I}$)
  if not OverlyGeneral($\mathcal{R}$, $E^+$) then Halt
  $k \leftarrow 0$; $\mathcal{R}_k \leftarrow \mathcal{R}$
  while $\exists\, e^- \in E^- : \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}_k) \leq_S \{e^-\}$ do
    if $\exists\, r \in \mathcal{R}_k$ s.t. $r$ is unfoldable and $r \in First(E^+)$ then
      $\mathcal{R}_{k+1} \leftarrow \mathcal{U}_{\mathcal{R}_k}(r)$; $k \leftarrow k+1$
    end if
    for each $r \in \mathcal{R}_k$ do
      if OverlyGeneral($\mathcal{R}_k \backslash \{r\}$, $E^+$) then $\mathcal{R}_k \leftarrow \mathcal{R}_k \backslash \{r\}$
    end for
  end while
end procedure
```

Fig. 1. The top-down correction algorithm.

decidability is ensured by requiring both confluence and (μ-termination) of the program.

During the unfolding phase, we select a rule upon which performing a program unfolding step. In order to specialize the program w.r.t. the example sets, we pick up an unfoldable rule which occurs in some proof of a positive example. More precisely, we choose a rule which appears first in a proof of a positive example, i.e. $e \in First(E^+)$, where function *First* is formally defined as follows.

Definition 7. *Let $\mathcal{R}$ be a TRS and E be an example set. Then, we define*

$$First(E) := \bigcup_{e \in E} \{r \mid e \in \widetilde{T}^{\kappa}_{\{r\}}(\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}))\}.$$

Once unfolding has been accomplished, we proceed to remove the “redundant” rules, that is, all the rules which are not needed to prove the positive example set E^+. This can be done by repeatedly testing the overgenerality of the specialized program w.r.t. E^+ and removing one rule at each iteration of the inner **for** loop. Roughly speaking, if program $\mathcal{R}_k \setminus \{r\}$ is overly general w.r.t. E^+, then rule r can be safely eliminated without losing E^+. Then, we can repeat the test on another rule.

Let us consider the coin-flip game to illustrate our algorithm in practice.

Example 3. Consider the following OBJ program $\mathcal{R}$ which is wrong w.r.t. the specification $\mathcal{I}$ of Example 1.

```
obj GAME is
sorts Nat Reward .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op prize : -> Reward .
   op sorry-no-prize : -> Reward .
   op coinflip : Nat -> Reward .
   var X : Nat .
   eq coinflip(s(X)) = coinflip(X) .        (1)
   eq coinflip(0) = prize .                 (2)
```

```
    eq coinflip(0) = sorry-no-prize .         (3)
  endo
```

Note that program $\mathcal{R}$ is non-confluent and computes both values `prize` and `sorry-no-prize` for any natural $\texttt{s}^n\texttt{(0)}$, $n > 0$. By fixing $\kappa = 3$, we get the following effective least fixpoint abstract semantics for $\mathcal{R}$.

$$\widetilde{\mathcal{F}}^3_{\mathsf{val}}(\mathcal{R}) = \{ \texttt{prize} = \texttt{prize}, \texttt{sorry-no-prize} = \texttt{sorry-no-prize}, \\ \texttt{coinflip(0)} = \texttt{prize}, \texttt{coinflip(0)} = \texttt{sorry-no-prize}, \\ \texttt{coinflip(s(0))} = \texttt{prize}, \texttt{coinflip(s(0))} = \texttt{sorry-no-prize}, \\ \texttt{coinflip(s(s(}\hat{\texttt{X}}\texttt{)))} = \texttt{prize}, \texttt{coinflip(s(s(}\hat{\texttt{X}}\texttt{)))} = \texttt{sorry-no-prize}\}.$$

Cosidering the abstract fixpoint semantics $\widetilde{\mathcal{F}}^3_{\mathsf{val}}(\mathcal{I})$ computed in Example 2 and following the methodology of Section 5, we obtain the example sets below:

$$E^+ = \{\texttt{coinflip(s(0))} = \texttt{prize}, \texttt{coinflip(0)} = \texttt{sorry-no-prize}\}$$
$$E^- = \{\texttt{coinflip(s(0))} = \texttt{sorry-no-prize}, \texttt{coinflip(0)} = \texttt{prize}\}.$$

Now, since program $\mathcal{R}$ fulfills the condition for overgenerality expressed by Proposition 4, the algorithm proceeds and enters the main loop. Here, program rule (1) is unfolded, because (1) is unfoldable and $r \in \mathit{First}(E^+)$. So, the transformed program is

```
    eq coinflip(s(s(X))) = coinflip(X) .      (4)
    eq coinflip(s(0)) = prize .               (5)
    eq coinflip(s(0)) = sorry-no-prize .      (6)
    eq coinflip(0) = prize .                  (7)
    eq coinflip(0) = sorry-no-prize .         (8)
```

Subsequently, a deletion phase is executed in order to check whether there are rules not needed to cover the positive example set E^+. The algorithm discovers that rules (4), (6), (7) are not necessary, and therefore are removed producing the correct program which consists of rules (5) and (8).

Note that the above example cannot be repaired by using the correction method of [1].

6.3 Correctness of algorithm TDCorrector

In this section, we prove the correctness of the top-down correction algorithm TDCorrector, i.e., we show that it produces a specialized version of $\mathcal{R}$ which is a correct program w.r.t. E^+ and E^-, provided that $\mathcal{R}$ is overly general w.r.t. E^+. A condition is necessary for establishing this result: no negative/positive couple of the considered examples must be proven by using the same sequence of rules. Let us start by giving an auxiliary definition and a technical lemma.

Definition 8. *Let $\mathcal{R}$ be a TRS. The* unfolding succession $\boldsymbol{S}(\mathcal{R}) \equiv \mathcal{R}_0, \mathcal{R}_1, \ldots$ *of program $\mathcal{R}$ is defined as follows:*

$$\mathcal{R}_0 = \mathcal{R}, \quad \mathcal{R}_{i+1} = \begin{cases} \mathcal{U}_{\mathcal{R}_i}(r) & \textit{where } r \in \mathcal{R}_i \textit{ is unfoldable and } r \textit{ is positive} \\ \mathcal{R}_i & \textit{otherwise} \end{cases}$$

where a rule is positive*, if it occurs in some proof of a positive example $e^+ \in E^+$.*

Note that rules belonging to $\mathit{First}(E^+)$ are positive rules. The next result states that we are always able to transform a program $\mathcal{R}$ into a program $\mathcal{R}'$ by a suitable number of unfolding steps, in such a way that any given proof of an example in $\mathcal{R}$ can be mimicked by a one-step proof in $\mathcal{R}'$.

Lemma 5. *Let $\mathcal{R}$ be a TRS and $t = c$ be an example. Let $t \to_{r_1} \ldots \to_{r_n} c$, $n \geq 1$. Then, for each unfolding succession $\boldsymbol{S}(\mathcal{R})$, there exists $\mathcal{R}_k$ occurring in $\boldsymbol{S}(\mathcal{R})$ such that $t \to_{r^*} c$, $r^* \in \mathcal{R}_k$.*

The following result immediately derives from Claim (1) of Proposition 3 .

Lemma 6. *Let $\mathcal{R}$ be a TRS and e an equation. Then, if $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \not\leq_S \{e\}$, then $e \notin \mathit{Sem}_{\mathsf{val}}(\mathcal{R})$.*

Now we are ready to prove the correctness of the algorithm.

Theorem 2. *Let E^+ and E^- be two sets of examples and $\mathcal{R}$ be a TRS overly general w.r.t. E^+. If there are no $e^+ \in E^+$ and $e^- \in E^-$ which can be proven by using the same sequence of rules, then the program $\mathcal{R}^c$ computed by* TDCorrector*, if any, is correct, i.e. $E^+ \subseteq \mathit{Sem}_{\mathsf{val}}(\mathcal{R}^c)$ and $E^- \cap \mathit{Sem}_{\mathsf{val}}(\mathcal{R}^c) = \emptyset$.*

Finally, let us mention that termination of the program is a sufficient condition for the termination of the TDCorrector algorithm.

7 Conclusions

In this paper, we have proposed a methodology for synthesizing (partially) correct functional programs written in OBJ style, which complements the diagnosis method which was developed previously in [2]. Our methodology is based on a combination, in a single framework, of a diagnoser which identifies those parts of the code containing errors, together with a deductive program learner which, once the bug has been located in the program, tries to repair it starting from evidence examples (uncovered as well as incorrect equations) which are essentially obtained as an outcome of the diagnoser. This method is not comparable to [1] as it is lower-cost and works for a much wider class of TRSs than the method which can be obtained as an instance of [1] for pure functional programs.

We are currently extending the prototypical implementation of the diagnosis system Debussy [2] (available at `http://www.dsic.upv.es/users/elp/soft.html`) with a correction tool which is based on the proposed abstract correction methodology.

References

1. M. Alpuente, D. Ballis, F. J. Correa, and M. Falaschi. Automated Correction of Functional Logic Programs. In Proc. of *ESOP 2003*, vol. 2618 of *LNAI*, 2003.
2. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In Proc. of *LOPSTR'02*, *LNCS* to appear. Springer, 2003.

3. M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, 22(3):221–252, 1995.
4. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
5. H. Bostrom and P. Idestam-Alquist. Induction of Logic Programs by Example–guided Unfolding. *Journal of Logic Programming*, 40:159–183, 1999.
6. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In Proc. of *IFL 2000*, vol. 2011 of *LNCS*, pages 176–193. Springer, 2001.
7. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
8. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
9. N. Dershowitz and U. Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494, 1993.
10. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *TCS*, 69(3):289–318, 1989.
11. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.
12. Andy Gill. Debugging haskell by observing intermediate data structures. In Graham Hutton, editor, *ENTCS*, vol. 41. Elsevier Science Publishers, 2001.
13. J. A. Goguen and G. Malcom. *Software Engineering with OBJ.* Kluwer Academic Publishers, Boston, 2000.
14. J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, vol. I, pages 1–112. Oxford University Press, 1992.
15. S. Lucas. Termination of (Canonical) Context-Sensitive Rewriting. In *Proc. RTA'02*, pages 296–310. Springer *LNCS* 2378, 2002.
16. M. J. Maher. Equivalences of Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, pp. 627–658. Morgan Kaufmann, 1988.
17. H. Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In Proc. of *ICFP '99*, pages 36–47. ACM Press, 1999.
18. H. Nilsson and P. Fritzson. Algoritmic debugging for lazy functional languages. *JFP*, 4(1):337–370, July 1994.
19. Henrik Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *JFP*, 11(6):629–671, November 2001.
20. J. T. O'Donell and C. V. Hall. Debugging in Applicative Languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.
21. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
22. A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
23. A. Pettorossi and M. Proietti. Transformation of Logic Programs. In *Handbook of Logic in Artificial Intelligence*, vol. 5. Oxford University Press, 1998.
24. J. Sparud and H. Nilsson. The architecture of a debugger for lazy functional languages. In Proc. of *AADEBUG'95*, 1995.
25. P. Wadler. Functional Programming: An angry half-dozen. *ACM SIGPLAN Notices*, 33(2):25–30, 1998.

Runtime Verification of Concurrent Haskell (work in progress)

Volker Stolz[1] and Frank Huch[2]

[1] RWTH Aachen, 52056 Aachen, Germany
[2] Christian-Albrecht-University of Kiel, 24118 Kiel, Germany

Abstract. In this article we use model checking techniques to debug Concurrent Haskell programs. LTL formulas specifying assertions or other properties are verified at runtime. If a formula cannot be satisfied, the debugger emits a warning and prints the path leading to the violation. It is possible to dynamically add formulas at runtime, giving a degree of flexibility which is not available in static verification of source code. We give a comprehensive example of using the new techniques to detect lock-reversal in Concurrent Haskell programs.

1 Introduction

Today, almost any larger software application is no longer a sequential program but an entire concurrent system made of a varying number of processes. These processes often share resources which must be adequately protected against concurrent access. Usually this is achieved by concentrating these actions in *critical sections* which are protected by semaphores. If a semaphore is taken, another process wishing to take it is suspended until the semaphore is released. Combining two or more semaphores can easily lead to situations where a dead lock might occur. If we record a trace of actions leading up to this event (or any other crash of the program), we can give the developer useful advice on how to try to debug this error. Also, we do not have to limit ourselves to recording events on semaphores. It is useful to have unique markers throughout the program detailing periodically the position of the current run. Although this technique mostly resembles *hello-debugging*, we will show that these traces offer an added value: We can do runtime verification on the trace of the program currently running by either interfacing to an external tool, or, as we will do in our approach, to a runtime verifier embedded into the system.

Usually one is interested in conditions which hold in specific places or throughout the entire execution of the program. These properties cannot be easily determined or verified at compile time because it is not possible to verify all possible paths of program execution. But violation of these conditions can be detected in a post-mortem analysis or at runtime, e.g. during intensive testing. Especially, we can aggregate information from one or more runs and detect behaviour which did not lead to a crash but indicates that a condition might be violated in the future or in subsequent runs. This is regularly the case if program behaviour is dependant on scheduling or external input.

The properties we are interested in are not necessarily classic model checking properties like liveness or fairness [9], but are known by the developer to hold at certain times. Commonly, these so-called assertions are verified by additional code placed by ambitious programmers at the appropriate locations in the source and usually removed before delivering the product because they increase program size and runtime.

A temporal logic like the *linear-time logic* (LTL) [1] lends itself naturally to describing properties depending on the changing states of a program over time. Formulas must never be evaluated to `false` during program execution. Our system allows for dynamic addition of new formulas when the system evolves and additional properties must hold.

We show how to employ the technique of specifying a formula and that verifying it at runtime can be a significant help in testing Concurrent Haskell [8] programs. Additionally, the system records the path leading up to the state where the property was violated and thus aids the developer in debugging the program.

The article is structured as follows: Section 2 gives a brief introduction to Haskell and the technique of recording traces and using them for debugging. Section 3 provides some background on LTL and describes the LTL runtime verifier. We then extend the checker to concurrent programs. Section 4 shows a sample verification of the *lock-reversal* problem. Section 5 concludes the paper.

2 Tracing Haskell programs

First, we remind the reader of the purely functional lazy programming language Haskell [7]. Although there is extensive work of debugging Haskell (see [12] for a comparison), these mostly focus on the difficulties on debugging the evaluation of expressions in a functional language: Evaluation might be deferred until the value is required for subsequent calculations. This leads to an evaluation order which cannot be easily derived from program source code (e.g. it is not possible to single-step through program execution as you would in C or JAVA). We will limit ourselves to debugging the IO behaviour of Haskell programs. Also, the IO actions provide reference points for positions in the trace.

We are especially interested in the extension to Concurrent Haskell [8] which integrates concurrent lightweight threads in Haskell's IO monad. As these threads can communicate, different schedules can lead to different results. Communication can take place by means of `MVars` (*mutable variables*). These MVars also take the role of semaphores protecting critical sections which makes them especially interesting to analyse. A tool for visualising communication between processes and explicitely manipulating the scheduling is the *Concurrent Haskell Debugger* portrayed in [3].

In the IO monad, threads can create MVars (`newEmptyMVar`), read values from MVars (`takeMVar`) and write values to MVars (`putMVar`). If a thread tries to read from an empty MVar or write to a full MVar, then it suspends until the MVar is filled respectively emptied by another thread. MVars can be used as simple semaphores, e.g. to assure mutual exclusion or for simple inter-thread communication. Based on MVars higher-level objects can be built, but are out of the scope of this paper.

The following example shows a Concurrent Haskell program which starts two processes and might produce either ”`ab`” or ”`ba`” as output:

```
main :: IO ()                     write :: MVar () -> Char -> IO ()
main = do                         write mvar c = do
  sync1 <- newEmptyMVar             putChar c
  sync2 <- newEmptyMVar             putMVar mvar ()
  forkIO $ write sync1 'a'
  forkIO $ write sync2 'b'
  mapM_ takeMVar [sync1,sync2]
```

The final synchronisation in the main process is required because the Haskell runtime system terminates on reaching the end of the program regardless of any other processes.

As we concern ourselves mostly with debugging concurrent programs, the usual point of failure will be related to those operations. Thus, we overload the Concurrent Haskell primitives to add information about the current action to the debugging trace. Additionally, we add a `mark` statement which is explicitely recorded in the trace. Let's take a look at a modified version of the program above which uses a different kind of synchronisation:

```
main :: IO ()                        write :: MVar () -> Char -> IO ()
main = do                            write mvar c = do
  sync <- newMVar ()                   takeMVar mvar
  forkIO $ do mark "process 1"         putChar c
              write sync 'a'           putMVar mvar ()
  forkIO $ do mark "process 2"
              write sync 'b'
  takeMVar sync
```

The following trace containing the result of actions and `mark` statements corresponds to an execution order which will end with a deadlock:

```
[ "0:putmvar 0", "0:fork 1", "1:process 1", "0:fork 2", "2:process 2",
  "0:takemvar 1", "1:takemvar 1", "2:takemvar 1" ]
```

We can now filter the trace for interesting events, e.g. we are not really interested in when a new process was created. It is thus easy to determine which execution order brought our system to a halt and the *Concurrent Haskell Debugger* can help the developer in trying to reproduce and debug this particular behaviour.

Instead of developing a variety of algorithms which work on the generated trace, we can resort to a well-known technique which is generally used to reason about state-based behaviour in time: Temporal logics. Using temporal logics it is possible to build a formula which must hold over time, for example: “If proposition P holds along a path, Q must also hold eventually”. Propositions correspond in a way to the actions and markers recorded in our trace. Before we proceed to the introduction of temporal logics, let's refine our view of what should be in a program trace.

Temporal logics assume that in each state the propositions currently holding are explicitely set. Clearly, this does not match our tracing approach where only particular events are recorded. In regular temporal logics, this property would be dropped when advancing in time. We adapt the following strategy to closer model temporal logics state space with our trace: All events on the trace are passed to a user-definable set of filters which decide whether to activate (`setProps`) or delete propositions (`releaseProps`) based on their internal state and the current event.

Propositions in a formula not present in the global state are considered `false` (closed world assumption). Also, we allow the combination of setting and deleting propositions with a single statement to modify the set of propositions atomically.

In Section 4 we will show how to generate formulas on the fly by analysing the trace and verify them.

3 Model checking LTL

Linear-time temporal logic is a subset of the Computation Tree Logic CTL* and extends propositional logic with additional operators which describe events along a computation path. The new operators have the following meaning:

- "Next" ($\mathbf{X}\ f$): The property holds in the next step
- "Eventually" ($\mathbf{F}\ f$): The property will hold at some state in the future (also: "in the future","finally")
- "Globally" ($\mathbf{G}\ f$): At every state on the path the property holds
- "Until" ($g\ \mathbf{U}\ h$): Combines to properties in the sense that if the second property holds, the first property has to hold along the preceding path
- "Release" ($g\ \mathbf{R}\ h$): Dual of $\mathbf{U}$; expresses that the second property holds along the path up to and including the first state where the first property holds, although the first property is not required to hold eventually.

The semantics of LTL is defined with respect to all paths in a given Kripke structure (a transition system with states labelled by atomic propositions (AP)). The path semantics of LTL is defined as follows:

Definition 1. *(Path Semantics of LTL) An infinite word over sets of propositions $\pi = p_0p_1p_2\ldots \in (\mathcal{P}(AP))^\omega$ is called a path. A path π satisfies an LTL–formula φ ($\pi \models \varphi$) in the following cases:*

$$\begin{array}{rcll}
p_0\pi & \models & P & \textit{iff } P \in p_0 \\
\pi & \models & \neg\varphi & \textit{iff } \pi \not\models \varphi \\
\pi & \models & \varphi \wedge \psi & \textit{iff } \pi \models \varphi \textit{ and } \pi \models \psi \\
p_0\pi & \models & \boldsymbol{X}\,\varphi & \textit{iff } \pi \models \varphi \\
p_0p_1\ldots & \models & \varphi\ \boldsymbol{U}\ \psi & \textit{iff } \exists i \in \mathbb{N} : p_ip_{i+1}\ldots \models \psi \textit{ and } \forall j < i : p_jp_{j+1}\ldots \models \varphi
\end{array}$$

The other operations are defined in terms of the above:

$$\begin{aligned}
\mathbf{F}\ \varphi &= \texttt{true}\ \mathbf{U}\ \varphi \\
\mathbf{G}\ \varphi &= \neg\mathbf{F}\ \neg\varphi
\end{aligned}$$

As an example we consider the formula P **U** Q. It is valid on the path $\{P\}\{P\}\{Q\}\ldots$ but it is neither valid on the path $\{P\}\{P\}\emptyset\{Q\}\ldots$ nor on the path $\{P\}\{P\}\{P\}\ldots$.

From the definition of the path semantics one can easily derive the following equivalence:

$$\varphi\ \mathbf{U}\ \psi\ \sim\ \psi \vee (\varphi \wedge (\mathbf{X}\ (\varphi\ \mathbf{U}\ \psi)))$$

Using this equivalence it is easy to implement a checker that successively checks the formula for a given path.

3.1 Runtime verification

In traditional model checking, the complete state space is derived from a specification and all possible paths are checked. There are two main disadvantages to this approach: First, the specification has to be derived from the source code which is to be checked. This usually involves parsing the source and often additional annotations, e.g. in special comments in the files. Then, the state space has to be generated, either before running the model checker (often requiring unmanageable amounts of storage) or on-the-fly, in which case e.g. detection of cycles gets harder.

Runtime verification does not need the state space beforehand, but simply tracks state changes in a running program. Thus, it limits itself to verifying that a formula holds along the path the program actually takes. This means that although errors could not be detected in the current run, they may still be present in the state space of the program. Various runs taking different paths may be necessary to find a path which violates the formula. Atomic propositions in a state are set or deleted explicitely by the program where additional statements have either been introduced by the developer or a separate tool which derived them from some sort of specification.

3.2 Implementing LTL in Haskell

In this section we sketch the rough details of the LTL runtime checking engine. A separate thread accepts requests to change the active set of properties from the user's program. It also handles requests by the application to add new formulas to the pool of functions to check. A `step` message causes the active formulas to be instantiated with the currently holding propositions. Evaluation of a formula may have three different outcomes:

1. `true`: The formula was proved and can be removed from the pool.
2. `false`: The verification failed. In case this condition does not coincide with a program crash, the user might want to proceed to check the remaining formulas.
3. a new formula: After instantiation, the formula reduced to neither `true` nor `false`. The new formula has to be checked on the remaining path.

We map these three cases into the data type `Either Bool (LTL a)` and define a function for checking an LTL formula over propositions of type `a`. Exemplarily, we present its definition for a basic proposition and the `until` operator:

```
checkStep :: Ord a => LTL a -> Set a -> Either Bool (LTL a)

checkStep (StateProp p) stateProps = Left (p `elementOf` stateProps)

checkStep u@(U phi psi) stateProps =
 let phiEval = checkStep phi stateProps in
   case checkStep psi stateProps of
     Left True  -> Left True
     Left False -> case phiEval of
                     Left False -> Left False
                     Left True  -> Right u
                     Right phi' -> Right (And phi' u)
     Right psi' -> case phiEval of
                     Left False -> Right psi'
                     Left True  -> Right (Or psi' u)
                     Right phi' -> Right (Or psi' (And phi' u))
```

In the last two cases of the outer case expression we reconstruct the formula that has to be checked in the next step.

3.3 Verifying concurrent programs

As long as only one process is generating a debugging trace that we use for verification, the interpretation of the debugging trace is clear. Each new entry in the trace corresponds to a transition in the Kripke structure. What happens if we allow more than one process to manipulate the global state? Consider the following example: Two processes (P_1 and P_2) want to set a different proposition each, unaware of the other process (see Figure 1). Although the states of the two processes are aligned on the time axis, it is wrong to assume that they proceed in lock-step! Concurrent Haskell has an interleaving semantics, so if we join the propositions from both processes we obtain the model in Figure 2. Notice how each path resembles a possible execution order in Haskell. A left branch indicates process P_1 taking a step while a right branch corresponds to a step by process P_2. If there is only one arc transition leaving a state, this is the only possible step because one of the processes has already reached its final state (final proposition underlined).

Because of this interleaving it is clear that you should not use "Ne**X**t" explicitely in your formula but instead use "**U**ntil" or another expression which captures that an arbitrary amount of steps may be taken before a certain process reaches its next state.

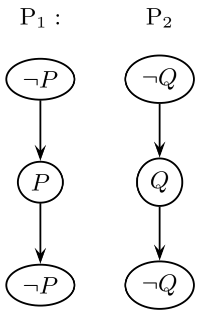


Fig. 1. Separated state spaces

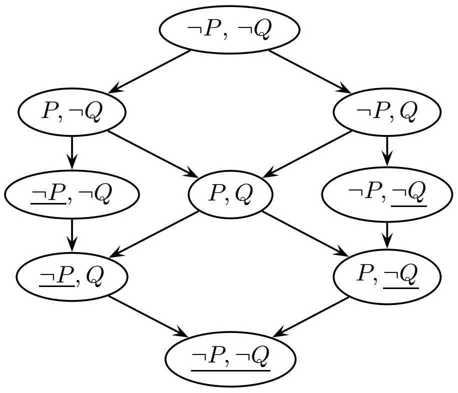


Fig. 2. Interleaved state spaces

3.4 Using LTL to check assertions

The small Haskell program in Figure 3 shows how to use the LTL verifier to discover a simple bug in the following protocol: Clients can ask a server to store a value. They write their `Set` request into an MVar and the new value into a second. The server reads the command and acts accordingly. The programmer expects that the server always stores the value corresponding to the previous `Set` message. But the protocol has a slight flaw which will be discovered during testing (cf. the Message Sequence Chart in Figure 4):

– The first client writes its `Set` command into the first MVar, but is preempted before sending the value `0`.
– The server takes this `Set` and waits for a value in the second MVar.
– The second client now issues a `Set` and writes `42` into the second MVar.
– The server can continue. It retrieved the command from the first client, but the value from the second client!

To detect this error behaviour during testing, it is adequate to insert the stated assertion that after each client access the system has to be in a consistent state, i.e. if we wrote a `0` there must not be a different proposition for the actual value at every step of the program. A specialised proposition `IntProp` is used in the example to avoid conversion of the values to strings. We obtain the following LTL formula: $\mathbf{G}\neg(set_0 \wedge value_{42})$.

The formula to verify is passed to the LTL checker at the beginning of the program. Formulas are additionally qualified by a name. The source shows how Haskell's algebraic data types can easily capture the term-structure of a formula. As in the definition of LTL, we can use a Haskell function "`g`" to express "Globally" through the predefined operators.

At each step where the set of propositions is modified the formula is checked and stepped through time. This is especially important to bear in mind for the

```
data SetMsg = Set

main :: IO ()
main = do
check "safe" (g (Not ((StateProp (IntProp 0 "set")) `And`
                      (StateProp (IntProp 42 "value")))))
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar
  forkIO $ client m1 m2 42
  forkIO $ client m1 m2 0
  server m1 m2 0
```

```
server :: MVar SetMsg
       -> MVar Int
       -> Int -> IO ()
server m1 m2 v = do
  msg <- takeMVar m1
  case msg of
    Set -> do
       v' <- takeMVar m2
       server m1 m2 v'
```

```
client :: MVar SetMsg
       -> MVar Int
       -> Int -> IO ()
client m1 m2 n = do
  putMVar m1 Set
  setProp (IntProp n "set")
  putMVar m2 n
  setProp (IntProp n "value")
  releaseProps [(IntProp n "set"),
                (IntProp n "value")]
  client m1 m2 n
```

Fig. 3. Client/Server example with flawed protocol

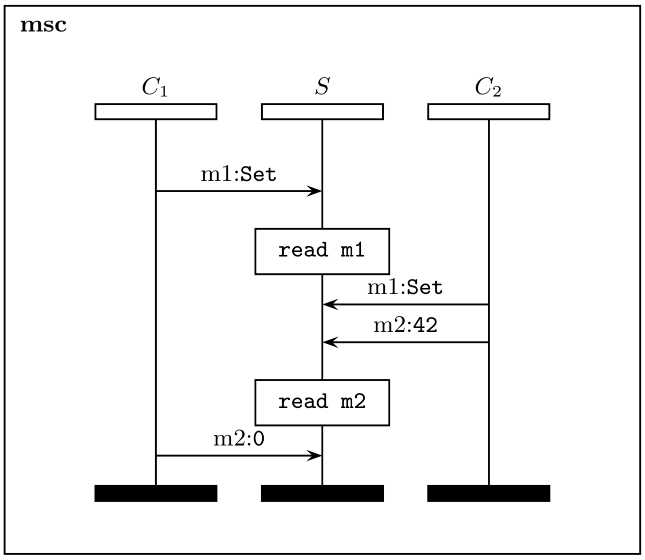


Fig. 4. Message Sequence Chart for erroneous run

last `setProp` statement followed by `releaseProps`. Given enough time and a "favourable" scheduling, as soon as the condition is violated the runtime verifier will stop the program.

4 Example application: Lock-reversal

In this section we will discuss how to implement a rather complex check using the new LTL runtime verifier. Let's state the problem: If two locks are taken in a specific order (with no release steps in between), the debugger should warn the user if he also uses these locks in swapped order (*lock-reversal*) because in concurrent programs this would mean that two processes could deadlock when their execution is scheduled in an unfortunate order.

We have two concurrent processes competing repeatedly for two locks A and B that guard a critical section. It is not possible to acquire several locks in one ago, so both locks have to be obtained sequentially. If we assume that the first process tries to obtain them in the order A, B and the second one in B, A, we can see that sooner or later the system ends up in a state where process 1 holds lock A and waits for B to become available while process 2 holds B waiting on A! This situation is exactly the circular holding pattern which indicates a deadlock (cf. chapter 7 of [2]). In small programs the bug might be easy to spot. For larger applications we can conclude that it is very hard to prevent this error from simply perusing the source code. Even when testing the program the error might not occur for various runs. For example it may be possible that process 1 and 2 are never executed interleaved but rather in a sequential manner.

In that case, we will still find the patterns of `1:takemvar A, 1:takemvar B` and `2:takemvar B, 2:takemvar A` in our trace. We can now warn the developer that his application has the potential to enter a deadlock under certain conditions. However, this warning might not be adequate as it is always possible to construct a program which takes the locks in reverse order and will never deadlock. This can for example be achieved by adding another lock, a so-called gate lock, to the program which assures mutual exclusion and prevents the system from deadlocking. An algorithm which avoids this kind of false warnings in the presence of gate locks is detailed in [10].

Because the number of MVars used in a program is not known beforehand, we have to extract the order in which MVars are taken at runtime from the debugging trace. We therefore attach an event listener to the debugger which gets passed a copy of each tracing event. If we inspect the already generated trace at runtime, it is possible to keep a list of known usage patterns and generate new formulas that warn appropriately. As the trace is simply a list of events or actions, Haskell's list processing capabilities and laziness make it very easy to calculate those informations.

This particular technique of noticing inadvertent lock-reversal at runtime is employed in the development phase of the recent FreeBSD operating system kernel under the name of *witness* [5]. In [10] the same technique is used in the Java PathFinder [13], a model checker for Java applications.

For debugging Concurrent Haskell programs, we provide an additional algebraic datatype which contains the `ThreadId` of the corresponding thread so that formulas can contain per-thread expressions. The two basic operations on MVars generate the following propositions:

```
– takeMVar: setProp     (ThreadProp <threadId> "locks <mvar>")
– putMVar:  releaseProp (ThreadProp <threadId> "locks <mvar>")
```

The remainder of the functions in the Concurrency module uses these two functions, so we instrumented our library to make it unnecessary to overload each single operation.

In the following we give a sample LTL formula for two locks which will evaluate to `false` in case locks are used in reverse order. Instead of using the algebraic datatype from the assertion, we will write $holds_{(p_i,l_x)}$ if process i holds lock x. The formula is passed to the model checker. The effect of the "Globally" is to check this formula over and over again for each step the program makes:

$$\mathbf{G}\ \neg\ (holds_{(p_j,l_x)} \wedge \neg holds_{(p_j,l_y)} \wedge (holds_{(p_j,l_x)}\ \mathbf{U}\ holds_{(p_j,l_y)}))$$

When the test program (see Figure 5) is executed first two `newMVar` invocations

```
main :: IO ()
main = do
  addEventListener lockVerifier
  mvA <- newMVar ()
  mvB <- newMVar ()
  take mvA mvB
  forkIO (...take mvB mvA...)
```

```
take :: MVar () -> MVar () -> IO ()
take m1 m2 = do
  takeMVar m1
  takeMVar m2
  ...
  putMVar m2 ()
  putMVar m1 ()
```

Fig. 5. Test scenario for lock-reversal

will be recorded in the trace because the initial values have to be written using `putMVar`. They get translated to `releaseProp` statements for the model checker. As the current state contains no properties yet, they are of no effect. When the `take` function is invoked for the first time from the main process, its access pattern is recorded by the even listener who generates the appropriate formula $\mathbf{G}\ \neg\ \varphi$ and passes it on to the model checker with

$$\varphi = holds_{(p_2,l_B)} \wedge \neg holds_{(p_2,l_A)} \wedge (holds_{(p_2,l_B)}\ \mathbf{U}\ holds_{(p_2,l_A)})$$

When the second thread takes the first MVar, this will set the corresponding proposition and execute a transition in the LTL checker. The formula is instantiated with the current state $\{\neg holds_{(p_2,l_A)}, holds_{(p_2,l_B)}\}$ and reduced to $\neg\varphi' \wedge\ \mathbf{G}\ \neg\varphi$, where

$$\begin{aligned}\varphi' &= holds_{(p_2,l_B)}\ \mathbf{U}\ holds_{(p_2,l_A)}\\ &= holds_{(p_2,l_A)}\ \vee\ (holds_{(p_2,l_B)} \wedge\ \mathbf{X}\ \varphi')\end{aligned}$$

.

After stepping the formula and taking the next MVar we can see that the new state $\{holds_{(p_2,l_A)}, holds_{(p_2,l_B)}\}$ evaluates φ' to `true`, which in turn gets negated and invalidates the whole formula. The model checker has determined that the specified property does not hold for the current path and acts accordingly, e.g. prints a warning to the user.

5 Related work

The Java PathFinder [10] is an instrumentation of the Java Virtual Machine which is able to collect data about problematic traces and run a traditional model checking algorithm. The Java PathExplorer [4] picks up the idea of runtime verification and uses the Maude rewriting logic tool [11] to implement e.g. future time LTL or past time LTL. Both approaches still require several components while our Haskell library is self-contained. Compaq's `Visual Threads` [6] debugger for POSIX threads applications also allows scripted, rule-based verification.

6 Conclusion and future work

We presented a debugging framework for Concurrent Haskell that records way points in program execution and allows the developer to trace a particular path through the program to draw conclusions about erroneous behaviour. Conditions which have to hold throughout program execution or at specific times can be expressed as formulas in the linear-time temporal logics LTL. The debugging trace provides the state transition annotations which are checked by the LTL model checking engine against the set of formulas. Apart from specific debugging modules the developer can use explicit annotations to add information to the debugging trace. The debugger also supports printing the tail of the trace from the point where proving the particular formula which turned `false` started.

It is also possible to extract information about loops in the program from the trace by placing unique markers throughout the source code. This reverse engineered program flow analysis could be used to generate new possible traces to check, although care must be taken before drawing conclusions from a derived trace as it might never occur in the original program. Both the locking example and the client/server example could be formulated more generally if we allow variables in formulas for Thread-Ids or variable contents.

We hope to integrate this debugging approach with the *Concurrent Haskell Debugger* to provide a comprehensive debugging facility for Concurrent Haskell. The effect of runtime verification on performance and memory requirements have yet to be examined for larger examples. We expect to profit from the inherent sharing of sub-formulas because of our choice for a lazy functional programming language. More information on the library for LTL runtime verification can be found at http://www-i2.informatik.rwth-aachen.de/˜stolz/Haskell/ .

References

1. A.Pnueli. The Temporal Logics of Programs. In *Proceddings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
2. A.Silberschatz and P.B.Galvin. *Operating System Concepts*. Addison-Wesley, 4th edition, 1994.
3. T. Böttcher and F. Huch. A Debugger for Concurrent Haskell. In *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*, Madrid, Spain, September 2002.
4. K. Havelund and G. Rosu. Java PathExplorer - A Runtime Verification Tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 2001.
5. J.H.Baldwin. Locking in the Multithreaded FreeBSD Kernel. In Samuel J. Leffler, editor, *Proceedings of BSDCon 2002, February 11-14, 2002, San Francisco, California, USA*. USENIX, 2002.
6. J.J.Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In W.Visser, K.Havelund, G.Brat, and S.Park, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, CA, USA, August/September 2000. Springer.
7. S. Peyton Jones et al. Haskell 98 Report. Technical report, `http://www.haskell.org/`, 1998.
8. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 1996.
9. E.M.Clarke Jr., O.Grumberg, and D.A.Peled. *Model Checking*. MIT Press, 1999.
10. K.Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In W.Visser, K.Havelund, G.Brat, and S.Park, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, CA, USA, August/September 2000. Springer.
11. M.Clavel, F.J.Durán, S.Eker, P.Lincoln, N.Martí-Oliet, J.Meseguer, and J.F.Quesada. The Maude system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications, RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer.
12. O.Chitil, C.Runciman, and M.Wallace. Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL 2000)*, volume 2011 of *Lecture Notes in Computer Science*, 2001.
13. W.Visser, K.Havelund, G.Brat, and S.Park. Model checking programs. In *Proc. of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.

A Monadic Semantics for Core Curry[⋆]

Andrew Tolmach and Sergio Antoy

Dept. of Computer Science, Portland State University,
P.O.Box 751, Portland OR 97207, USA
{apt,antoy}@cs.pdx.edu

Abstract. We give a high-level operational semantics for the essential core of the Curry language, including higher-order functions, call-by-need evaluation, non-determinism, narrowing, and residuation. The semantics is structured in monadic style, and is presented in the form of an executable interpreter written in Haskell.

1 Introduction

The functional logic language Curry combines lazy functional programming with logic programming features based on both narrowing and residuation. Describing the semantics of these features and their interaction in a common framework is a non-trivial task, especially because functional and logic programming have rather different semantic traditions. The "official" semantics for Curry [4], largely based on work by Hanus [6, 5], is an operational semantics based on definitional trees and narrowing steps. Although fairly low-level, this semantics says nothing about sharing behavior. Functional languages are typically given denotational or natural ("big step") operational semantics. In more recent work [1], Albert, Hanus, Huch, Oliver, and Vidal propose a natural semantics, incorporating sharing, for the first-order functional and narrowing aspects of Curry; however, in order to collect all non-deterministic solutions and to incorporate residuation, they fall back on a small-step semantics. While small-step semantics have the advantage of being closer to usable abstract machines, they tend to be lower-level than big-step models, and perhaps harder to reason about.

In this paper, we describe a variant of the semantics by Albert, *et al.* that remains big-step, but delivers all non-deterministic solutions and handles residuation. It is also fully higher-order. Our treatment is not especially original; rather, we have attempted to apply a variety of existing techniques to produce a simple and executable semantic interpreter. In particular, we follow Seres, Spivey, and Hoare [13] in recording all possible non-deterministic solutions in a lazily constructed *forest*, which can be traversed in a variety of orders to ensure fair access to all solutions. We use *resumptions* [12, Ch. 12] to model the concurrency needed to support residuation. We organize the interpreter using *monads* in the style popularized by Wadler [15], which allows a modular presentation of non-determinism [7, 3] and concurrency. We believe that the resulting semantics will be useful for understanding Curry and exploring language design alternatives. It may also prove useful as a stepping stone towards new practical implementations of the language.

[⋆] This work has been supported in part by the National Science Foundation under grants CCR-0110496 and CCR-0218224

```
data Exp = Var Var                           type Var = String
         | Int Int                           type Constr = String
         | Abs Var Exp                       type Prim = String
         | App Exp Exp                       type Pattern = (Constr,[Var])
         | Capp Constr [Exp]
         | Primapp Prim Exp Exp
         | Case Exp [(Pattern,Exp)]
         | Letrec [(Var,Exp)] Exp
```

Fig. 1. Expressions.

Our semantics is defined as an executable interpreter, written in Haskell, for a core subset of Curry. We present the interpreter in three phases. Section 2 describes the evaluation of the functional subset of the language. Section 3 introduces non-determinism, logic variables, and narrowing. Section 4 adds residuation. Section 5 offers brief conclusions. The reader is assumed to be familiar with Haskell and with monads.

2 The Functional Language Interpreter

The abstract syntax for the functional part of our core expression language is given in Figure 1. The language is not explicitly typed, but we assume throughout that we are dealing with typeable expressions. Function abstractions (`Abs`) have exactly one argument; multiple-argument functions are treated as nested abstractions. Functions need not be lambda-lifted to top level. For simplicity, the only primitive type is integers (`Int`) and all primitive operators are binary; these limitations could be easily lifted. Primitive operators, which are named by strings, act only on constructor head-normal forms. Each constructor is named by a string c, and is assumed to have a fixed arity $ar(c) \geq 0$. Constructors can be invented freely as needed; we assume that at least the nullary constructors `True`, `False`, and `Success` are available.

Constructor applications (`Capp`) and patterns (within `Case` expressions) and primitive applications (`Primapp`) are fully applied. Unapplied or partially applied constructors and primitives in the source program must be η-expanded. `Case` expressions analyze constructed values. Patterns are "shallow;" they consist of a constructor name c and a list of $ar(c)$ variables. The patterns for a single `Case` are assumed to be mutually exclusive, but not necessarily exhaustive. More complicated pattern matching in the source language must be mapped to nested `Case` expressions. Such nested case expressions can be used to encode definitional trees [2]. Finally, `Letrec` expressions introduce sets of (potentially) mutually recursive bindings.

Curry is intended to use "lazy" or, more properly, *call-by-need* evaluation, as opposed to simple call-by-name evaluation (without sharing). Although the results of call-by-name and call-by-need cannot be distinguished for purely functional computations, the time and space behavior of the two strategies are very different. More essentially, the introduction of non-determinism (Section 3) makes the difference between strategies observable. We therefore model call-by-need evaluation explicitly using a mutable *heap* to represent shared values. (The heap is also used to represent recursion without

```
type HPtr = Int
data Value =
   VCapp Constr [HPtr]
 | VInt Int
 | VAbs Env Var Exp
type Env = Map Var HPtr
data Heap = Heap {free :: [HPtr], bindings :: Map HPtr HEntry}
data HEntry =
   HValue Value
 | HThunk Env Exp
 | HBlackhole
```

Fig. 2. Key data types for evaluation.

recourse to a Y-combinator.) This approach corresponds to the behavior of most real implementations of call-by-need languages; its use in formal semantics was introduced by Launchbury [10] and elaborated by Sestoft [14], and was also adopted by Albert, *et al.* [1].

Following a very old tradition [9], we interpret expressions in the context of an *environment* that maps variables to heap locations, rather than performing substitution on expressions. The same expression (e.g., the body of an abstraction) can therefore evaluate to different values depending on the values of its free variables (e.g., the abstraction parameter). In this approach we view the program as immutable code rather than as a term in a rewriting system.

The main evaluation function is

```
eval :: Env -> Exp -> M Value
```

which evaluates an expression in the specified environment and returns the corresponding constructor head-normal form (HNF) value embedded in a monadic computation. The monad

```
newtype M a = M (Heap -> A (a,Heap))
```

represents stateful computations on heaps. Type `A a` is another monad, representing *answers* involving type `a`.

The key type definitions, shown in Figure 2, are mutually recursive. Values correspond to HNFs of expressions. Constructed values (`VCapp`) correspond to constructor applications; the components of the value are described by pointers into the heap (`HPtr`). Closures (`VAbs`) correspond to abstraction expressions, tagged with an explicit environment to resolve variable references in the abstraction body. Environments and values are only well-defined in conjunction with a particular heap that contains bindings for the heap pointers they mention.

A heap is a map `bindings` from heap pointers (`HPtr`) to heap entries (`HEntry`), together with a supply `free` of available heap pointers. Heap entries are either HNF values, unevaluated expressions (`HThunk`), or “black holes” (`HBlackhole`). We expect thunk entries to be overwritten with value entries as evaluation proceeds. Black holes

```
data Map a b = Map [(a,b)]
mempty :: Map a b
mempty = Map []
mget :: Eq a => Map a b => a -> b
mget (Map l) k = fromJust (lookup k l)
mset :: Map a b -> (a,b) -> Map a b
mset (Map l) (k,d) = Map ((k,d):l)
```

Fig. 3. A specification for the `Map` ADT. Note that the ordering of the list guarantees that each `mset` of a given key supersedes any previous `mset` for that key. An efficient implementation would use a sorted tree or hash table (and hence would put stronger demands on the class of `a`).

are used to temporarily overwrite a heap entry while a thunk for that entry is being computed; attempting to read a black-hole value signals (one kind of) infinite recursion in the thunk definition [8, 14].

Both `Env` and `Heap` rely on an underlying abstract data type `Map a b` of applicative finite maps from `a` to `b`, supporting simple get and set operations (see Figure 3). Note that `mset` returns a *new* map, rather than modifying an existing one. It is assumed that `mget` always succeeds. The map operations are lifted to the `bindings` component of heaps as `hempty`, `hget`, and `hset`. The function

```
hfresh :: Heap -> (HPtr,Heap)
```

picks and returns a fresh heap pointer. As evaluation progresses, the heap can only grow; there is no form of garbage collection. We also don't perform environment *trimming* [14], though this would be easy to add.

The evaluation function returns a monadic computation `M Value`, which in turn uses the answer monad `A`. Using monads allows us to keep the code for `eval` simple, while supporting increasingly sophisticated semantic domains. Our initial definition for `M` is given in Figure 4. Note that `M` is essentially just a function type used to represent computations on heaps. The "current" heap is passed in as the function argument, and a (possibly updated) copy is returned as part of the function result. As usual, bind (>>=) operations represent sequencing of computations; `return` injects a value into the monad without changing the heap; `mzero` represents a failed evaluation; `mplus` represents alternative evaluations (which will be used in Section 3). The monad-specific operations include `fresh`, which returns a fresh heap location; `fetch`, which returns the value bound to a pointer (assumed valid) in the current heap; and `store`, which extends or updates the current heap with a binding. The function `run` executes a computation starting from the empty heap.

Type `A` is also a monad, representing *answers*. Note that the uses of bind, `return`, `mzero` and `mplus` in the bodies of the functions defined on `M` are actually the monad operators for `A` (*not* recursive calls to the `M` monad operators!). In this section, we equate A with the exception monad `Maybe a`, so that an answer is either `Just` a pair (HNF value,heap) or `Nothing`, representing "failure." Failure occurs only when a required arm is missing from a non-exhaustive `Case` expression, or when an attempt is made to fetch from a black-holed heap location. `A` gets instance definitions of `>>=`, `return`,

```
newtype M a = M (Heap -> A (a,Heap))
instance Monad M where
  (M m1) >>= k = M (\h -> do (a',h') <- m1 h
                             let M m2 = k a' in m2 h')
  return x = M (\h -> return (x,h))
instance MonadPlus M where
  mzero = M (\_ -> mzero)
  (M m1) `mplus` (M m2) = M (\h -> m1 h `mplus` m2 h)
fresh :: M HPtr
fresh = M (\h -> return (hfresh h))
store :: HPtr -> HEntry -> M ()
store p e = M (\h -> return ((),hset h (p,e)))
fetch :: HPtr -> M HEntry
fetch x = M (\h -> return (hget h x,h))
run :: M a -> A (a,Heap)
run (M m) = m hempty
```

Fig. 4. The evaluation monad.

```
instance Monad Maybe where         instance MonadPlus Maybe where
    Just x  >>= k = k x                mzero         = Nothing
    Nothing >>= k = Nothing
    return        = Just
```

Fig. 5. The `Maybe` type as a monad.

and `mzero` for `Maybe` from the standard library (Figure 5). Note that `(>>=)` propagates failure.

With this machinery in place, the actual `eval` function is quite short (Figure 6). Evaluation of expressions already in HNF is trivial, except for constructor applications, for which each argument expression must be allocated as a separate thunk (since it might be shared). Evaluation of applications is also simple. Assuming that the program is well-typed, the operator expression must evaluate to an abstraction. The argument expression is allocated as a thunk and bound to the formal parameter of the abstraction; the body of the abstraction is evaluated in the resulting environment.

`Letrec` bindings just result in thunk allocations for the right-hand sides. To make the bindings properly recursive, all the thunks share the same environment, to which all the bound variables have been added.

The key memoization step required by call-by-need occurs when evaluating a `Var` expression. In a well-typed program, each variable must be in the domain of the current environment. The corresponding heap entry is fetched: if this is already in HNF, it is simply returned. If it is a thunk, it is recursively evaluated (to HNF), and the resulting value is written over the thunk before being returned.

A `Case` expression is evaluated by first recursively evaluating the expression being "cased over" to HNF. In a well-typed program, this must be a `VCapp` of the same type as the case patterns. If the `VCApp` constructor matches one of the patterns, the pattern

```
eval :: Env -> Exp -> M Value
eval env (Int i) =  return (VInt i)
eval env (Abs x b) = return (VAbs env x b)
eval env (Capp c es) =
     do ps <- mapM (const fresh) es
        zipWithM_ store ps (map (HThunk env) es)
        return (VCapp c ps)
eval env (App e0 e1) =
     do VAbs env' x b <- eval env e0
        p1 <- fresh
        store p1 (HThunk env e1)
        eval (mset env' (x,p1)) b
eval env (Letrec xes e) =
     do let (xs,es) = unzip xes
        ps <- mapM (const fresh) xes
        let env' = foldl mset env (zip xs ps)
        zipWithM_ store ps (map (HThunk env') es)
        eval env' e
eval env (Var x) =
     do let p = mget env x
        h <- fetch p
        case h of
          HThunk env' e' ->
            do store p (HBlackhole)
               v' <- eval env' e'
               store p (HValue v')
               return v'
          HValue v -> return v
          HBlackhole -> mzero
eval env (Case e pes) =
     do VCapp c0 ps <- eval env e
        let plookup [] = mzero
            plookup (((c,xs),b):pes) | c == c0   = return (xs,b)
                                     | otherwise = plookup pes
        (xs,b) <- plookup pes
        let env' = foldl mset env (zip xs ps)
        eval env' b
eval env (Primapp p e1 e2) =
     do v1 <- eval env e1
        v2 <- eval env e2
        return (doPrimapp p v1 v2)

doPrimapp :: Prim -> Value -> Value -> Value
doPrimapp "eq" (VInt i1) (VInt i2) | i1 == i2  = VCapp "True" []
                                   | otherwise = VCapp "False" []
doPrimapp "add" (VInt i) (VInt j) = VInt (i+j)
doPrimapp "xor" (VCapp "True" []) (VCapp "True" []) = VCapp "False" []
doPrimapp "xor" (VCapp "True" []) (VCapp "False" []) = VCapp "True" []
...
```

Fig. 6. Call-by-need `eval` function and auxiliary `doPrimapp` function.

variables are bound to the corresponding VCApp operands (represented by heap pointers), and the corresponding right-hand side is evaluated in the resulting environment. If no pattern matches, the evaluation fails, indicated by returning mzero.

To evaluate a primitive application, the arguments are evaluated to HNF in left-to-right order and the resulting values are passed to the auxiliary function doPrimapp, which defines the behavior of each primitive operator.

Finally, to interpret a whole-program expression, we can define

```
interp :: Exp -> Maybe (Value,Heap)
interp e = run (eval mempty e)
```

which executes the monadic computation produced by evaluating the program in an empty initial environment. If we are only interested in the head constructor of the result value, we can project out the Value component and ignore the Heap.

3 Non-determinism, Logic Variables, and Narrowing

We now revise and extend our interpreter to incorporate the key logic programming features of Curry. To do this, we must record multiple possible results for evaluation, by redefining the monadic type A of answers. (Changing A implicitly also changes M, although the code defining M's functions doesn't change.) By choosing this monad appropriately, we can add non-deterministic features to our existing interpreter without making any changes to the deterministic fragment; the hidden "plumbing" of the bind operation will take care of threading the multiple alternatives. Deterministic choices are injected into the monad using return; non-deterministic choice will be represented by mplus; as before, failure of (one) non-deterministic alternative will be represented by mzero. The definition of A is addressed in Section 3.2.

3.1 Logic Variables and Narrowing

First, we show how to add logic variables and narrowing to the language and interpreter. This requires surprisingly little additional machinery, as shown in Figure 7. We add a new expression form Logic to declare scoped logic variables; the Curry expression "e where x free" is encoded as (Logic "x" e). We add a corresponding HNF VLogic HPtr, which is essentially a reference into the heap. Logic declarations are evaluated by allocating a fresh heap location p, initially set to contain VLogic p, binding the logic variable to this location, and executing the body in the resulting environment.

We now must consider how to handle VLogic values within the eval function. The most important change is to Case; if the "cased-over" expression is bound to a VLogic value, the evaluator performs *narrowing*. (We temporarily assume all cases are "flexible.") This is done by considering each provided case arm in turn. For a pattern with constructor c and argument parameters $x_1, \ldots, x_n$, the evaluator allocates n fresh logic variable references $p_1, \ldots, p_n$, overwrites the cased-over logic variable in the heap with (VCApp c [p_1,...,p_n]), extends the environment with bindings of x_i to p_i, and recursively evaluates the corresponding case arm in that environment. All the resulting monadic computations are combined using mplus; this is the only place where mplus

```
lift :: M a -> M a
lift (M m) = M (\h -> forestLift (m h))

data Exp = Logic Var Exp | ... as before ...
data Value = VLogic HPtr | ... as before ...

eval env (Var x) =
     lift $ ... as before ...

eval env (Logic x e) =
     do p <- allocLogic
        eval (mset env (x,p)) e

eval env (Case e pes) =
     do v <- eval env e
        case v of
          VCapp c0 ps -> ... as before ...
          VLogic p0 ->
            foldr mplus mzero (map f pes)
            where
              f ((c,xs),e') =
                do ps <- mapM (const allocLogic) xs
                   store p0 (HValue (VCapp c ps))
                   let env' = foldl mset env (zip xs ps)
                   eval env' e'

allocLogic :: M HPtr
allocLogic = do p <- fresh ; store p (HValue (VLogic p)) ; return p
```

Fig. 7. Evaluating logic features.

is used in the interpreter, and hence the sole source of non-determinism. Note that other possible non-deterministic operators can be encoded using `Case`; e.g., the Flat Curry expression (`or` e_1 e_2) can be encoded as

```
or e1 e2 ≡ Logic "dummy" (Case (Var "dummy")
                                 [(("Ldummy",[]), e1),
                                  (("Rdummy",[]), e2)])
```

We also need to make a few other small changes to the `eval` code (not shown here) to guard against the appearance of `VLogic` values in strict positions, namely the operator position of an `App` and the operand positions of a `Primapp`; in such cases, `eval` will return `mzero`. Note that because of this latter possibility for failure, the left-to-right evaluation semantics of `Primapp`s can be observed. For example, the evaluation of

```
Logic "x"
  (Primapp "xor"
     (Var "x")
     (Case (Var "x") [(("True",[]), Capp "False" [])]))
```

fails, but would succeed (with `True`) if the order of arguments to `and` were reversed. This characteristic may seem rather undesirable; we consider alternatives in Section 4.

3.2 Monads for Non-determinism

It remains to define type `A` in such a way that it can record multiple non-deterministic answers. The standard choice of monad for this purpose is *lists* [15]. In this scheme, non-deterministic choice of a value is represented by a list of values; `return` a produces the singleton list `[`a`]`; `mplus` is concatenation `(++)`; m `>>=` k applies k to each element in m and concatenates the resulting lists of elements; and `mzero` is the empty list `[]`. However, if we want to actually execute our interpreter and inspect the answers, the list monad has a significant problem: its `mplus` operation does not model *fair* non-deterministic choice. Essentially this is because evaluating m_1 `++` m_2 forces evaluation of the full spine of m_1, so $\perp$ `‘mplus‘` m `=` $\perp$. If the left alternative leads to an infinite computation, the right alternative will never be evaluated at all. For example, evaluating `Letrec ["f",or (Var "f") (Int 1)] (Var "f")` should produce the answer 1 (infinitely many times). However, if we represent answers by lists, our interpreter will compute (roughly speaking)

```
(eval (Var "f")) ‘mplus‘ (return (VInt 1))
= (eval (Var "f")) ++ [VInt 1]
```

If we attempt to inspect this answer, we immediately cause a recursive evaluation of `f`, which produces the same thing; we never see any part of the answer. In effect, using this monad amounts to performing depth-first search of the tree of non-deterministic choices, which is incomplete with respect to the expected semantics.

To avoid this problem, we adopt the idea of Seres, Spivey, and Hoare [13], and represent non-determinism by a lazy *forest* of trees of values (Figure 8). We set `type A a = Forest a`. As before, we represent choices as a list, with `mplus` implemented as `(++)`, but now the lists are of trees of values. To obtain the values, we can traverse the trees using any ordering we like; in particular, we can use breadth-first rather than depth-first traversal:

```
interp :: Exp -> [(Value,Heap)]
interp =  bfs (run (eval mempty e))
```

This approach relies fundamentally on the laziness of the forest structure. Non-trivial tree structures are built using the `forestLift` operator, which converts an arbitrary forest into a singleton one by making all the trees into branches of a single new tree. Applying `forestLift` to a value v before concatenating it into the forest with `mplus` will delay the point at which v is encountered in a breadth-first traversal, and hence allow the other argument of `mplus` to be explored first. For the example above, the forest answer will have the form

```
(forestLift (eval (Var "f"))) ‘mplus‘ (return (VInt 1))
= (Forest [Fork (eval (Var "f"))]) ‘mplus‘ (Forest [Leaf (VInt 1)])
= Forest [(Fork (eval (Var "f"))) ++ (Leaf (VInt 1))]
= Forest [Fork (eval (Var "f")), Leaf (VInt 1)]
```

Applying `bfs` to this answer will produce `VInt 1` (the head of its infinite result) before it forces the recursive evaluation of `f`.

To make use of `forestLift`, we need to add a new `lift` operator to `M`, defined in Figure 7. We have some flexibility about where to place calls to `lift` in our interpreter

```
newtype Forest a = Forest [Tree a]
data Tree a = Leaf a | Fork (Forest a)
instance Monad Forest where
  m >>= k = forestjoin (forestmap k m)
  return a = Forest [Leaf a]
instance MonadPlus Forest where
  mzero = Forest []
  (Forest m1) `mplus` (Forest m2) = Forest (m1 ++ m2)

forestLift :: Forest a -> Forest a
forestLift f = Forest [Fork f]

forestjoin :: Forest (Forest a) -> Forest a
forestjoin (Forest ts) = Forest (concat (map join' ts))
  where join' :: Tree (Forest a) -> [Tree a]
        join' (Leaf (Forest ts)) = ts
        join' (Fork xff) = [Fork (forestjoin xff)]

treemap :: (a -> b) -> Tree a -> Tree b
treemap f (Leaf x) = Leaf (f x)
treemap f (Fork xf) = Fork (forestmap f xf)

forestmap :: (a -> b) -> Forest a -> Forest b
forestmap f (Forest ts) = Forest (map (treemap f) ts)

bfs :: Forest a -> [a]
bfs (Forest ts) = concat (bfs' ts)
 where bfs' :: [Tree a]  -> [[a]]
       bfs' ts = combine (map levels ts)
       levels :: Tree a -> [[a]]
       levels (Leaf x) = [[x]]
       levels (Fork (Forest xf)) = []:bfs' xf
       combine :: [[[a]]] -> [[a]]
       combine = foldr merge []
       merge :: [[a]] -> [[a]] -> [[a]]
       merge (x:xs) (y:ys) = (x ++ y):(merge xs ys)
       merge xs [] = xs
       merge [] ys = ys
```

Fig. 8. Monad of forests and useful traversal functions (adapted from [13]).

code. For fairness, we must make sure that there is a `lift` in every infinite cycle of computations made by the interpreter. The simplest way to guarantee this is to apply `lift` on each call to `eval`. If we do this, there is a clear parallel between the evaluation of the answer structure and the behavior of a *small-step* semantics. However, more parsimonious placement of `lifts` will also work; for example, since every cyclic computation must involve a heap reference, it suffices to `lift` only the evaluation of `Var` expressions, as shown in Figure 7.

```
type Flex = Bool
type Concurrent = Bool
type Prim = (String,Concurrent)
data Exp = Case Flex Exp [(Pattern,Exp)] | ... as before ...

eval env (Case flex e pes) =
     do v <- eval env e
        case v of
          VCapp c ps -> ... as before ...
          VLogic p0 | flex ->
            ... as before ...
            store p0  ...
            yield $
            ... as before ...
          VLogic _ | otherwise -> mzero

eval env (Primapp (p,concurrent) e1 e2) =
     do (v1,v2) <-
          if concurrent then
            conc (eval env e1) (eval env e2)
          else do v1 <- eval env e1; v2 <- eval env e2; return (v1,v2)
        ... as before ...

doPrimapp "and" (VCapp "Success" []) (VCapp "Success" []) =
                                                   VCapp "Success" []
```

Fig. 9. Interpreter changes to support residuation.

4 Residuation

In real Curry, appearance of a logic variable in a rigid position causes evaluation to *residuate*, i.e., suspend until the logic variable is instantiated (to a constructor-rooted expression). Residuation only makes sense if there is the possibility of concurrent computation—otherwise, the suspended computation can never hope to be restarted. The only plausible place to add concurrency to our existing core language is for evaluation of arguments to primitives. We use an interleavings semantics for concurrency; the result is semantically simple though not practically efficient (since the number of interleavings can easily grow exponentially).

We can add residuation support to our core language by making only minor changes to our interpreter, as shown in Figure 9. We extend the core language syntax by adding a boolean flag to each primitive operator indicating whether its arguments are to be evaluated concurrently. The most obvious candidate for this evaluation mode is the parallel `and` operator normally used in Curry to connect pairs of constraints. We also add a flag on `Case` expressions to distinguish flexible and rigid cases.

The evaluator relies on significant changes to the underlying monad `M`, which is modified to describe concurrent computations using *resumptions* [12, Ch. 12], a standard method from denotational semantics. Each monadic computation is now modeled as a *stream* of partial computations. The `conc` operator takes two such streams and produces a computation that (non-deterministically) realizes all possible interleavings of the streams. The atomicity of interleaving is controlled by uses of `yield`; placing a `yield` around a computation indicates a possible interleaving point.

With this monadic support in place, our approach to residuation is simple, and requires very few changes in the `eval` function. Attempts to perform a rigid case over an unresolved logic variable simply fail (just as in other strict contexts). However, arguments to concurrent primitives are evaluated using the `conc` operator, so that if there are any possible evaluation sequences that resolve the logic variable before it is cased over, those sequences will be tried (along with potentially many other sequences that lead to failure). To make this approach work, we must permit enough interleaving that all possible causal interactions between the two argument evaluation sequences are captured. A brute-force approach would be to `yield` before each recursive call to `eval`. However, since logic variable heap locations can only be updated by the `store` operation in the interpreter code for flexible `Case` expressions, it suffices to `yield` following that `store`.[1]

To illustrate how interleaving works, we can define canonical code sequences for reading and writing logic variables:

```
wr x k ≡ Case True  (Var x) [(("True",[]), k)]
rd x k ≡ Case False (Var x) [(("True",[]), k)]
```

Here `wr` x k writes `True` into logic variable x (assumed to be not already set), and then continues by evaluating expression k. Conversely, `rd` x k attempts to read the contents of logic variable x (assumed to contain `True`); if this is successful, it continues by evaluating k; otherwise, it fails. Now consider the expression

```
Logic "x" (Logic "y" (Primapp ("and",True)
                      (rd "x" (wr "y" (Capp "Success" [])))
                      (wr "x" (rd "y" (Capp "Success" [])))))
```

The application of the concurrent primitive `and` causes evaluation of the two argument expressions to be interleaved. Each `wr` induces a `yield` immediately following the store into the variable, so there are three possible interleavings, shown in the three columns below. The first and third of these fail (at the point marked `*`); only the second succeeds.

execution order	*left-arg*	*right-arg*	*left-arg*	*right-arg*	*left-arg*	*right-arg*
1	`rd "x" *`			`wr "x"`		`wr "x"`
2	`wr "y"`		`rd "x"`			`rd "y" *`
3		`wr "x"`	`wr "y"`		`rd "x"`	
4		`rd "y"`		`rd "y"`	`wr "y"`	

Note that it is perfectly possible to label ordinary primitives like addition as concurrent. The net effect of this will be to make the result independent of argument evaluation

[1] It is also essential to use black-holing as described in Section 2, thus providing a (different) kind of atomicity around thunk evaluations; otherwise some interleavings might cause a thunk heap location to be updated twice with different values.

```
data State a = Done a Heap
             | Running Heap (M a)
newtype M a = M (Heap -> A (State a))
instance Monad M where
  (M m1) >>= k = M (\h ->
    do s <- m1 h
       case s of
         Done a' h' -> let M m2 = k a' in m2 h'
         Running h' m' -> return (Running h' (m' >>= k)))
  return x = M (\h -> return (Done x h))
fresh :: M HPtr
fresh = M (\h -> let (v,h') = hfresh h in return (Done v h'))
store, fetch  ... modified similarly ...
yield :: M a -> M a
yield m = M (\h -> return (Running h m))
conc :: M a -> M b -> M (a,b)
conc m1 m2 = (m1 `thn` m2) `mplus` (liftM swap (m2 `thn` m1))
  where
    (M m1') `thn` m2 = M (\h ->
     do s <- m1' h
        case s of
          Done a h' -> return (Running h' (m2 >>= \b -> return (a,b)))
          Running h' m' -> let M m'' = conc m' m2 in m'' h')
    swap (a,b) = (b,a)
run :: M a -> A (a,Heap)
run = run' hempty
 where
   run' h (M m) =
     do s <- m h
        case s of
          Done a h' -> return (a, h')
          Running h' m' -> run' h' m'
```

Fig. 10. Monad changes to supporting residuation.

order, thus removing one of the drawbacks we noted to the narrowing semantics of Section 3.1. In fact, it is hard to see that anything *less* than concurrency *can* achieve order-independence. In other words, making primitive applications independent of argument order seems to be no easier than adding residuation.

It remains to describe the implementation of resumptions, which is entirely within monad `M`, revised as shown in Figure 10. Each computation now returns one of two `State`s: either it is `Done`, producing a value and updated heap, or it is still `Running`, carrying the heap as updated so far together with a new `M` computation describing the remainder of the (original) computation. Simple computations (such as primitive `fresh`, `store` and `fetch` operations) just return `Done` states. Computations returning `Running` states are generated by the `yield` operator. The `conc` operator non-deterministically tries both orders for evaluating its arguments at each stage of partial computation. As before, support for non-determinism is given by monad `A` (which does not change).

5 Conclusion and Future Work

We have presented a simple executable semantics for the core of Curry. The full code for the three interpreter versions described here is available at `http://www.cs.pdx.edu/~apt/curry-monads`. The structure of our semantics sheds some light on how the basic components of the language interact. In particular, we can see that the addition of non-determinism, logic variables and narrowing can be accomplished just by making a suitable shift in interpretation monad. We could emphasize this modularity further by presenting the relevant monads as compositions of *monad transformers* [11].

While we think this semantics is attractive in its own right, it would obviously be useful to give a formal characterization of its relationship with the various existing semantics for Curry; we have not yet attempted this. As additional future work, we plan to pursue the systematic transformation of this semantics into a small-step form suitable as the basis for realistic interpreters and compilers.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for functional logic languages. In M. Comini and M. Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier Science Publishers, 2002.
2. S. Antoy. Definitional trees. In *Proc. 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
3. K. Claessen and P. Ljunglof. Typed logical variables in Haskell. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.
4. M. Hanus, editor. *Curry: An Integrated Functional Logic Language*. Available at `http://www.informatik.uni-kiel.de/~mh/curry/`.
5. M. Hanus. A unified computation model for declarative programming. In *Proc. 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97)*, pages 9–24, 1997.
6. M. Hanus. A unified computation model for functional and logic programming. In *Proc. POPL'97, 24st ACM Symp. on Principles of Programming Languages*, pages 80–93, 1997.
7. R. Hinze. Prological features in a functional setting — axioms and implementations. In M. Sato and Y. Toyama, editors, *Third Fuji International Symp. on Functional and Logic Programming (FLOPS'98)*, pages 98–122. World Scientific, Apr. 1998.
8. R. E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.
9. P. Landin. The mechanical evaluation of expressions. *Computer J.*, 6(4):308–320, Jan. 1964.
10. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL '93, 20th Annual ACM Symp. on Principles of Programming Languages*, pages 144–154, 1993.
11. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. 22nd ACM Symp. on Principles of Programming Languages*, pages 333–343, 1995.
12. D. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.
13. S. Seres, M. Spivey, and T. Hoare. Algebra of logic programming. In *Proc. International Conference on Logic Programming (ICLP'99)*, pages 184–199, Nov. 1999.
14. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.
15. P. Wadler. The essence of functional programming. In *Proc. POPL'92, Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 1–14, 1992.

Improving (Weakly) Outermost-Needed Narrowing: Natural Narrowing[⋆]

Santiago Escobar

DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain.
`sescobar@dsic.upv.es`

Abstract. *Outermost-needed narrowing* is a sound and complete optimal demand-driven strategy for the class of inductively sequential constructor systems. Its parallel extension (known as *weakly*) deals with non-inductively sequential constructor systems. In this paper, we present *natural narrowing*, a suitable extension of (weakly) outermost-needed narrowing which is based on a refinement of the demandness notion associated to outermost-needed narrowing. Intuitively, natural narrowing always reduces or instantiates the most demanded position in a term. We formalize the strategy for left-linear constructor systems though, for the class of inductively sequential constructor systems, natural narrowing behaves even better than outermost-needed narrowing in the avoidance of failing computations. Regarding inductively sequential constructor systems, we introduce a bigger class of systems called *inductively sequential preserving* where natural narrowing preserves optimality for sequential parts of the program.

1 Introduction

A challenging problem in modern programming languages is the discovery of sound and complete evaluation strategies which are “optimal” w.r.t. some efficiency criterion (e.g. the number of evaluation steps) and which are easily implementable.

For orthogonal term rewriting systems (TRSs), Huet and Lévy’s *needed rewriting* is optimal. However, automatic detection of needed redexes is difficult (or even impossible) and Huet and Lévy defined the notion of ‘strong sequentiality’, which provides a formal basis for the mechanization of sequential, normalizing rewriting computations [14]. In [14], Huet and Lévy defined the (computable) notion of strongly needed redex and showed that, for the (decidable) class of strongly sequential orthogonal TRSs, the steady reduction of strongly needed redexes is normalizing. When strongly sequential TRSs are restricted to constructor systems (CSs), we obtain the class of inductively sequential CSs [13]. Intuitively, a CS is inductively sequential if there exists some branching selection structure in the rules. Sekar and Ramakrishnan provided *parallel needed reduction* as the extension of Huet and Lévy needed rewriting for non-inductively sequential CSs [19].

⋆ Work partially supported by CICYT TIC2001-2705-C03-01, and MCYT grants HA2001-0059 and HU2001-0019.

A sound and complete narrowing strategy for the class of inductively sequential CSs is *outermost-needed narrowing* [6]. The optimality properties of outermost-needed narrowing and the fact that inductively sequential CSs are a subclass of strongly sequential programs justifies the fact that outermost-needed narrowing is useful in functional logic programming as the functional logic counterpart of Huet and Lévy's strongly needed reduction [13,14]. For non-inductively sequential CSs, *weakly outermost-needed narrowing* [5] is defined as the functional logic counterpart of Sekar and Ramakrishnan parallel needed reduction [19]. Unfortunately, (weakly) outermost-needed narrowing does not work appropriately on non-inductively sequential CSs.

Example 1. Consider Berry's program [19] where `T` and `F` are constructor symbols:

```
B(T,F,X) = T     B(F,X,T) = T     B(X,T,F) = T
```

This CS is not inductively sequential, since it is not clear whether terms of shape $\mathtt{B}(t_1,t_2,t_3)$ can be narrowed sequentially when t_1, t_2, and t_3 are rooted by defined function symbols or variables; where 'to be narrowed sequentially' is understood as the property of narrowing only positions which are unavoidable (or "needed") in order to obtain a normal form [6]. Although the CS is not inductively sequential[1], some terms can still be narrowed sequentially. For instance, the term $t = \mathtt{B(X,B(F,T,T),F)}$ has a unique optimal narrowing sequence which achieves the associated normal form `T`:

$$\mathtt{B(X,\underline{B(F,T,T)},F)} \leadsto_{id} \underline{\mathtt{B(X,T,F)}} \leadsto_{id} \mathtt{T}$$

However, weakly outermost-needed narrowing is not optimal since, apart of the previous optimal sequence, the sequence

$$\mathtt{B(X,\underline{B(F,T,T)},F)} \leadsto_{\{\mathtt{X}\mapsto\mathtt{T}\}} \underline{\mathtt{B(T,T,F)}} \leadsto_{id} \mathtt{T}$$

is also obtained. The reason is that weakly outermost-needed narrowing partitions the CS into inductively sequential subsets $\mathcal{R}_1 = \{\mathtt{B(X,T,F)\ =\ T}\}$ and $\mathcal{R}_2 = \{\mathtt{B(T,F,X)\ =\ T}, \mathtt{B(F,X,T)\ =\ T}\}$ in such a way that the first step of the former (optimal) narrowing sequence is obtained w.r.t. subset $\mathcal{R}_1$ whereas the first step of the latter (useless) narrowing sequence is obtained w.r.t. subset $\mathcal{R}_2$.

As Sekar and Ramakrishnan argued in [19], strongly sequential systems have been so far used in programming languages and violations are not frequent. However, it is a cumbersome restriction and its relaxation is quite useful in some contexts. It would be convenient for programmers that such restriction could be relaxed while optimal evaluation is preserved for strongly sequential parts of a program. The availability of optimal and effective evaluation strategies without such restriction can encourage programmers to formulate non-inductively sequential programs, which could be still executed in an optimal way.

[1] Note that since the TRS is orthogonal and right-ground, a computable and optimal rewrite strategy based on tree automata exists without any requirement on inductively sequential CSs; though possibly non-effective (see [10]). On the other hand, there exist normalizing, sequential rewrite strategies for almost orthogonal TRSs; though possibly non-optimal (see [8]).

Example 2. Consider the problem of coding the rather simple inequality

$$x + y + z + w \leqslant h$$

for natural numbers x, y, z, w, and h. A first (and naïve) approach can be to define an inductively sequential program:

```
solve(X,Y,Z,W,H) = (((X + Y) + Z) + W) ⩽ H
0 ⩽ N = True                0 + N = N
s(M) ⩽ 0 = False            s(M) + N = s(M + N)
s(M) ⩽ s(N) = M ⩽ N
```

Consider the (auxiliary) factorial function $n!$. The program is effective for term[2] t_1 = `solve(1,Y,0,10!,0)` since `False` is returned in 5 rewrite steps but not very effective for t_2 = `solve(10!,0,1,W,0)` or t_3 = `solve(10!,0+1,0+1,W,1)` since several rewrite steps should be performed on `10!` before narrowing the two terms to `False`.

When a more effective program is desired for some input terms, an ordinary solution is to include some assertions by hand into the program while preserving determinism of computations. For instance, if terms t_1, t_2, and t_3 are part of such input terms of interest, we could obtain the following program

```
solve(X,s(Y),s(Z),W,s(H)) = solve(X,Y,s(Z),W,H)
solve(X,0,s(Z),W,s(H)) = solve(X,0,Z,W,H)
solve(s(X),Y,0,W,s(H)) = solve(X,Y,0,W,H)
solve(0,s(Y),0,W,s(H)) = solve(0,Y,0,W,H)
solve(0,0,0,0,H) = True
solve(s(X),Y,0,W,0) = False
solve(X,0,s(Z),W,0) = False
solve(0,s(Y),Z,0,0) = False
solve(X,s(Y),s(Z),s(W),0) = False
```

Note that this program is not inductively sequential but orthogonal, and some terms can still be narrowed sequentially, e.g. when considering the term t_4 = `solve(X,Y,0+s(s(0)),W,s(0))`, we have the following narrowing sequences[3]:

$$\begin{array}{l}
\texttt{solve(X,Y,}\underline{\texttt{0+s(s(0))}}\texttt{,W,s(0))} \\
\quad \leadsto_{id} \texttt{solve(X,Y,s(s(0)),W,s(0))} \\
\quad \leadsto_{\{\texttt{Y}\mapsto\texttt{0}\}} \texttt{solve(X,0,s(0),W,0)} \mid \leadsto_{\{\texttt{Y}\mapsto\texttt{s(Y}'\texttt{)}\}} \texttt{solve(X,Y}'\texttt{,s(s(0)),W,0)} \\
\quad \leadsto_{id} \texttt{False} \mid \leadsto_{\{\texttt{Y}'\mapsto\texttt{0}\}} \texttt{False} \mid \leadsto_{\{\texttt{Y}'\mapsto\texttt{s(Y}''\texttt{)},\texttt{W}\mapsto\texttt{0},\texttt{X}\mapsto\texttt{0}\}} \texttt{False} \\
\qquad\qquad \mid \leadsto_{\{\texttt{Y}'\mapsto\texttt{s(Y}''\texttt{)},\texttt{W}\mapsto\texttt{s(W}'\texttt{)}\}} \texttt{False}
\end{array}$$

Indeed, note that even if we add the rule `solve(X,s(Y),0,W,0) = False`, which makes the program almost orthogonal, t_4 still has a sequential evaluation.

Modern (multiparadigm) programming languages apply computational strategies which are based on some notion of demandness of a position in a term by a rule (see [7] for a survey discussing this topic). Programs in these languages

[2] Natural numbers 1,2,... are used as a shorthand for `s(s(...s(0)))`, where `s` is applied $1, 2, \ldots$ times.

[3] Symbol '`|`' is used as a separator to denote different narrowing steps from a similar term.

are commonly modeled by left-linear CSs and computational strategies take advantage of this constructor condition (see [1,2,18]). Furthermore, the semantics of these programs normally promote the computation of values (or constructor head-normal forms) rather than head-normal forms. For instance, (weakly) outermost-needed narrowing considers the constructor condition though some refinement is still possible as shown by the following example.

Example 3. Consider the following TRS from [9] defining the symbol ÷ which encodes the division function between natural numbers.

```
0 ÷ s(N) = 0                        M − 0 = M
s(M) ÷ s(N) = s((M−N) ÷ s(N))      s(M) − s(N) = M − N
```

Consider the term $t =$ `10! ÷ 0`, which is a (non-constructor) head-normal form. Outermost-needed narrowing forces[4] the reduction of the first argument and evaluates `10!` in an useless way. The reason is that outermost-needed narrowing uses a data structure called *definitional tree* which encodes the order of evaluation without testing whether rules associated to each branch could ever be matched to the term or not. A similar problem occurs with term `X ÷ 0` since it is instantiated to `0` and `s` when they are not really necessary.

In this paper, we provide a refinement of the demandness notion associated to outermost-needed narrowing. We introduce an extension of outermost-needed narrowing called *natural narrowing*. Intuitively, natural narrowing always narrows the most demanded position in a term. Our definition applies to left-linear CSs though it is based on the notion of inductively sequential term. However, for the class of inductively sequential CSs, natural narrowing behaves even better than outermost-needed narrowing in the avoidance of failing computations. Regarding inductively sequential CSs, we introduce a bigger class of TRSs called *inductively sequential preserving* where natural narrowing preserves optimality for sequential parts of the program.

2 Preliminaries

We assume some familiarity with term rewriting (see [20] for missing definitions) and narrowing (see [12] for missing definitions). Throughout the paper, $\mathcal{X}$ denotes a countable set of variables and $\mathcal{F}$ denotes a signature, i.e. a set of function symbols $\{\mathtt{f}, \mathtt{g}, \ldots\}$. We denote the set of terms built from $\mathcal{F}$ and $\mathcal{X}$ by $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A term is said to be linear if it has no multiple occurrences of a single variable. Let $\mathit{Subst}(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ denote the set of substitutions. We denote by id the "identity" substitution: $\mathit{id}(x) = x$ for all $x \in \mathcal{X}$. Terms are ordered by the preorder $\leq$ of "relative generality", i.e. $s \leq t$ if there exists σ s.t. $\sigma(s) = t$. Term t is a renaming of s if $t \leq s$ and $s \leq t$. A substitution σ is more general than θ, denoted by $\sigma \leq \theta$, if $\theta = \sigma\gamma$ for some substitution γ. By $\mathcal{P}os(t)$ we denote

[4] Indeed, this fact depends on how a real implementation builds definitional trees. However, as shown in Example 15 below, the problem persists for other terms.

the set of positions of a term t. Given a set $S \subseteq \mathcal{F} \cup \mathcal{X}$, $\mathcal{P}os_S(t)$ denotes positions in t where symbols in S occur. We denote the root position by Λ. Given positions p, q, we denote its concatenation as $p.q$. Positions are ordered by the standard prefix ordering $\leq$. The subterm at position p of t is denoted as $t|_p$, and $t[s]_p$ is the term t with the subterm at position p replaced by s. The symbol labelling the root of t is denoted as $root(t)$. A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $l \notin \mathcal{X}$. The left-hand side (*lhs*) of the rule is l and r is the right-hand side (*rhs*). A TRS is a pair $\mathcal{R} = (\mathcal{F}, R)$ where R is a set of rewrite rules. $L(\mathcal{R})$ denotes the set of *lhs*'s of $\mathcal{R}$. A TRS $\mathcal{R}$ is left-linear if for all $l \in L(\mathcal{R})$, l is a linear term. Given $\mathcal{R} = (\mathcal{F}, R)$, we take $\mathcal{F}$ as the disjoint union $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors* and symbols $f \in \mathcal{D}$, called *defined functions*, where $\mathcal{D} = \{root(l) \mid l \to r \in R\}$ and $\mathcal{C} = \mathcal{F} - \mathcal{D}$. A pattern is a term $f(l_1, \ldots, l_k)$ where $f \in \mathcal{D}$ and $l_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, for $1 \leq i \leq k$. A TRS $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ is a constructor system (CS) if all lhs's are patterns. A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ narrows to s (at position p with substitution σ), written $t \leadsto_{\{p,\sigma,l\to r\}} s$ (or just $t \leadsto_\sigma s$), if $\sigma(t|_p) = \sigma(l)$ and $s = \sigma(t[r]_p)$, for some (possibly renamed) rule $l \to r \in R$, $p \in \mathcal{P}os(t)$, and substitution σ. The subterm $\sigma(l)$ is called a redex. A term t is a head-normal form (or root-stable) if it cannot be narrowed to a redex. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap* if there is a non-variable position $p \in \mathcal{P}os(l)$ and a most-general unifier σ such that $\sigma(l|_p) = \sigma(l')$. The pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ is called a critical pair. A critical pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ with $p = \Lambda$ is called an *overlay.* A critical pair $\langle s, t \rangle$ such that $s = t$ is called *trivial.* A left-linear TRS without critical pairs is called *orthogonal.* A left-linear TRS whose critical pairs are trivial overlays is called *almost orthogonal.*

3 Natural Narrowing

Some evaluation (rewrite or narrowing) strategies proposed up to date are called demand driven and can be classified as call-by-need. Loosely speaking, *demandness* is understood as follows: if possible, a term t is evaluated at the top; otherwise, some arguments of the root of t are (recursively) evaluated if they might promote the application of a rewrite rule at the top, i.e. if a rule "demands" the evaluation of these arguments. In [7], Antoy and Lucas present the idea that modern functional (logic) languages include evaluation strategies which consider elaborated definitions of 'demandness' which are, in fact, related [2,5,6,11,15,16,18].

In [1], we provided a notion of demandness which gives us a useful algorithm to calculate the rules and demanded positions necessary in an evaluation sequence.

Definition 1 (Disagreeing positions). [1] *Given terms* $t, l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*, we let* $\mathcal{P}os_{\neq}(t, l) = minimal_{\leq}(\{p \in \mathcal{P}os(t) \cap \mathcal{P}os_{\mathcal{F}}(l) \mid root(l|_p) \neq root(t|_p)\})$*.*

Definition 2 (Demanded positions). [1] *We define the set of* demanded *positions of a term* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *w.r.t.* $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *(a lhs of a rule defining* *root*(*t*)), *i.e. the set of (positions of) maximal disagreeing subterms as:*

$$DP_l(t) = \begin{cases} \mathcal{P}os_{\neq}(t,l) & \mathit{if}\ \mathcal{P}os_{\neq}(t,l) \cap \mathcal{P}os_{\mathcal{C}}(t) = \varnothing \\ \varnothing & \mathit{otherwise} \end{cases}$$

and the set of demanded positions of t *w.r.t. TRS* $\mathcal{R}$ *as:* $DP_{\mathcal{R}}(t) = \cup\{DP_l(t) \mid l \to r \in \mathcal{R} \wedge root(t) = root(l)\}$.

Example 4. Consider again Example 2 and term t_4. We have $DP_{\mathcal{R}}(t_4) = \{1,2,3,4\}$

$DP_{\mathtt{solve(X,s(Y),s(Z),W,s(H))}}(t_4) = \{2,3\}$ $\quad DP_{\mathtt{solve(X,0,s(Z),W,s(H))}}(t_4) = \{2,3\}$
$DP_{\mathtt{solve(s(X),Y,0,W,s(H))}}(t_4) = \{1,3\}$ $\quad DP_{\mathtt{solve(0,s(Y),0,W,s(H))}}(t_4) = \{1,2,3\}$
$DP_{\mathtt{solve(0,0,0,0,H)}}(t_4) = \{1,2,3,4\}$ $\quad DP_{\mathtt{solve(s(X),Y,0,W,0)}}(t_4) = \varnothing$
$DP_{\mathtt{solve(X,0,s(Z),W,0)}}(t_4) = \varnothing$ $\quad DP_{\mathtt{solve(0,s(Y),Z,0,0)}}(t_4) = \varnothing$
$DP_{\mathtt{solve(X,s(Y),s(Z),s(W),0)}}(t_4) = \varnothing$

Note that the restriction of disagreeing positions to positions with non-constructor symbols disables the evaluation of subterms which could never produce a redex (see [1,2,18]). Outermost-needed narrowing also considers this constructor condition though some refinement is still possible, as it was shown in Example 3.

In [7], Antoy and Lucas introduced the idea that some evaluation strategies select "popular" demanded positions over the available set. We formalize here a notion of popularity of demanded positions which makes the difference by counting the number of times a position is demanded by some rule and, then, we select the most (frequently) demanded positions. This information, i.e. the number of times a position is demanded, is crucial and it is managed by using multisets instead of sets for representing demanded positions.

We (re)-define the set of demanded positions of t w.r.t. TRS $\mathcal{R}$, $DP^{\mathfrak{m}}_{\mathcal{R}}(t)$, as follows.

Definition 3 (Demanded positions). *We define the set of demanded positions of a term* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *w.r.t. TRS* $\mathcal{R}$ *as:* $DP^{\mathfrak{m}}_{\mathcal{R}}(t) = \oplus\{DP_l(t) \mid l \to r \in \mathcal{R} \wedge root(t) = root(l)\}$ *where* $M_1 \oplus M_2$ *is the union of multisets* M_1 *and* M_2.

Example 5. Continuing Example 4, $DP^{\mathfrak{m}}_{\mathcal{R}}(t_4) = \{4,1,1,1,2,2,2,2,3,3,3,3,3\}$.

The following definition establishes the strategy used in natural narrowing for selecting demanded positions. In order to obtain optimal narrowing strategies, computed substitutions must be unavoidable to achieve a head-normal form (see [6] for details). In the following, $x \preceq y$ denotes that the number of occurrences of x in the multiset is less or equal to the number of occurrences of y.

Definition 4 (Natural narrowing). *Given a term* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *and a TRS* $\mathcal{R}$, $\mathfrak{m}(t)$ *is defined as the smallest set satisfying*

$$\mathfrak{m}(t) \ni \begin{cases} (\Lambda, id, l{\to}r) & \mathit{if}\ l \to r \in \mathcal{R}, \exists\sigma.\sigma(l) = t \\ (p.q, \theta, l{\to}r) & \mathit{if}\ t\ \mathit{is\ not\ a\ redex\ w.r.t.}\ \mathcal{R}, DP_{\mathcal{R}} \cap \mathcal{P}os_{\mathcal{X}}(t) = \varnothing, p \in DP_{\mathcal{R}}, \\ & \quad \mathit{and}\ (q, \theta, l \to r) \in \mathfrak{m}(t|_p) \\ (p, \theta\circ\sigma, l{\to}r) & \mathit{if}\ t\ \mathit{is\ not\ a\ redex\ w.r.t.}\ \mathcal{R}, p \in DP_{\mathcal{R}} \cap \mathcal{P}os_{\mathcal{X}}(t), c \in \mathcal{C}, \\ & \quad \sigma(t|_p) = c(\overline{w}),\ \mathit{and}\ (p, \theta, l \to r) \in \mathfrak{m}(\sigma(t)) \end{cases}$$

where $\overline{w}$ *are fresh variables and* $DP_{\mathcal{R}} = maximal_{\preceq}(DP^{\mathfrak{m}}_{\mathcal{R}}(t))$.

We consider $\mathtt{m}$ simply as a function returning positions if substitutions and rules selected for narrowing are not relevant or directly deducible from the context. Intuitively, instantiation is performed only in those variables which are part of the most demanded positions, whereas reduction is only performed when no most demanded position associated to a variable is available. We say term t narrows by *natural narrowing* to term s using substitution σ, denoted by $t \overset{\mathtt{m}}{\rightsquigarrow}_{\{p,l\to r,\sigma\}} s$ (or simply $t \overset{\mathtt{m}}{\rightsquigarrow}_{\sigma} s$), if $(p,\sigma,l\to r) \in \mathtt{m}(t)$.

Example 6. Continuing now Example 5. It is easy to see that $\mathtt{m}(t_4) = \{3\}$ since positions 1, 2, and 4, which correspond to variables, are also demanded but the redex at position 3 is the most demanded. Then, the only possible narrowing step is: $\mathtt{solve(X,Y,}\underline{\mathtt{0+s(s(0))}}\mathtt{,W,s(0))} \overset{\mathtt{m}}{\rightsquigarrow}_{id} \mathtt{solve(X,Y,s(s(0)),W,s(0))}$.

From now on, we investigate the properties of natural narrowing.

3.1 Neededness and sequentiality

First, we recall the definition of a definitional tree and an inductively sequential CS. A definitional tree of a defined symbol f is a finite, non-empty set $\mathcal{T}$ of linear patterns partially ordered by $\leq$ and having the following properties (up to renaming of variables) [3]:

leaf property The maximal elements, referred to as the *leaves*, of $\mathcal{T}$ are renamings of the left-hand sides of the rules defining f. Non-maximal elements are referred to as *branches*.

root property The minimum element of $\mathcal{T}$ is referred to as the *root*[5].

parent property If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the *parent* of π and π as a *child* of π'.

induction property All the children of a same parent differ from each other only at the position, referred to as *inductive*, of a variable of their parent.

A defined symbol f is called *inductively sequential* if there exists a definitional tree $\mathcal{T}$ with pattern $f(x_1,\ldots,x_k)$ (where $x_1,\ldots,x_k$ are different variables) whose leaves contain all and only the rules defining f. In this case, we say that $\mathcal{T}$ is a definitional tree for f, denoted as $\mathcal{T}_f$. A left-linear CS[6] $\mathcal{R}$ is *inductively sequential* if all its defined function symbols are inductively sequential. An inductively sequential CS can be viewed as a set of definitional trees, each defining a function symbol. It is often convenient and simplifies understanding to provide a graphical representation of definitional trees as a tree of patterns where the inductive position in branch nodes is surrounded by a box [3].

[5] This definition of root property differs from that of [3]; in [3]: "the minimum element, referred to as the root, of $\mathcal{T}$ is $f(x_1,\ldots,x_k)$, where $x_1,\ldots,x_k$, are fresh, distinct variables." This provides us with a more flexible definition which fits our interests.

[6] Left-linear CSs is the largest class where *outermost-needed narrowing* can be defined [4].

$$\begin{tikzcd}
 & \boxed{\mathtt{M}} \div \mathtt{N} \arrow[dl] \arrow[dr] & & & \mathtt{M} \div \boxed{\mathtt{N}} \arrow[d] & \\
0 \div \boxed{\mathtt{N}} \arrow[d] & & \mathtt{s(M)} \div \boxed{\mathtt{N}} \arrow[d] & & \boxed{\mathtt{M}} \div \mathtt{s(N)} \arrow[dl] \arrow[dr] & \\
0 \div \mathtt{s(N)} & & \mathtt{s(M)} \div \mathtt{s(N)} & 0 \div \mathtt{s(N)} & & \mathtt{s(M)} \div \mathtt{s(N)}
\end{tikzcd}$$

Definitional tree (*a*) Definitional tree (*b*)

Fig. 1. The two possible definitional trees for symbol ÷.

$$\begin{tikzcd}
 & & & \mathtt{solve(X,\boxed{Y},Z,W,H)} \arrow[dl] \arrow[dr] & & & \\
 & & \mathtt{solve(X,0,\boxed{Z},W,H)} \arrow[dl] & & \mathtt{solve(X,s(Y),\boxed{Z},W,H)} \arrow[dr] & & \\
 & \mathtt{solve(X,0,s(Z),W,\boxed{H})} \arrow[dl] \arrow[d] & & & & \mathtt{solve(X,s(Y),s(Z),W,\boxed{H})} \arrow[dl] \arrow[dr] & \\
\mathtt{solve(X,0,s(Z),W,0)} & \mathtt{solve(X,0,s(Z),W,s(H))} & & & \mathtt{solve(X,s(Y),s(Z),\boxed{W},0)} \arrow[d] & & \mathtt{solve(X,s(Y),s(Z),W,s(H))} \\
 & & & & \mathtt{solve(X,s(Y),s(Z),s(W),0)} & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & \mathtt{solve(\boxed{X},Y,Z,W,H)} \arrow[dll] \arrow[drr] & & & \\
 & & \mathtt{solve(0,\boxed{Y},Z,W,H)} \arrow[dl] \arrow[d] & & & & \mathtt{solve(s(X),Y,Z,W,\boxed{H})} \arrow[d] \arrow[dr] & \\
 & \mathtt{solve(0,0,\boxed{Z},W,H)} \arrow[dl] & \mathtt{solve(0,s(Y),Z,W,\boxed{H})} \arrow[dl] \arrow[d] & & & & \mathtt{solve(s(X),Y,\boxed{Z},W,0)} \arrow[d] & \mathtt{solve(s(X),Y,\boxed{Z},W,s(H))} \arrow[d] \\
\mathtt{solve(0,0,0,0,H)} & \mathtt{solve(0,s(Y),Z,\boxed{W},0)} \arrow[d] & \mathtt{solve(0,s(Y),\boxed{Z},W,s(H))} \arrow[d] & & & & \mathtt{solve(s(X),Y,0,W,0)} & \mathtt{solve(s(X),Y,0,W,s(H))} \\
 & \mathtt{solve(0,s(Y),Z,0,0)} & \mathtt{solve(0,s(Y),0,W,s(H))} & & & & &
\end{tikzcd}$$

Fig. 2. A partition of definitional trees for symbol `solve`

Example 7. Symbol ÷ in Example 3 is inductively sequential since there exists a definitional tree for pattern $\mathtt{M} \div \mathtt{N}$. Figure 1 shows the two possible associated definitional trees.

When a CS is not inductively sequential, a rule partition which obtains inductively sequential subsets can be applied.

Example 8. Symbol `solve` in Example 2 is non-inductively sequential, since a rule partition which splits its rules into subsets which are inductively sequential can be applied. Figure 2 shows the definitional tree of each partition.

Now, we extend to terms the notion of an inductively sequential symbol. This is the key idea of the paper. We denote the maximal set of rules which can eventually be matched to a term t after some evaluation by $match_t(\mathcal{R}) = \{l \in L(\mathcal{R}) \mid DP_l(t) \neq \varnothing \text{ or } l \leq t\}$. Note that given a term t with variables and a substitution σ, $match_{\sigma(t)}(\mathcal{R}) \subseteq match_t(\mathcal{R})$.

Definition 5 (Inductively sequential term). *Let* $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ *be a CS and* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*. We say* t *is* inductively sequential *if for all* $p \in \mathcal{P}os_{\mathcal{D}}(t)$*, there exists a definitional tree* $\mathcal{T}$ *with root pattern* π *such that* $\pi \leq t|_p$ *and leaves of* $\mathcal{T}$ *are all and only instances of* $match_{t|_p}(\mathcal{R})$*.*

$$\begin{tikzcd}
 & & & \mathtt{solve(X,Y,\boxed{Z},W,s(0))} \arrow[dl] \arrow[dr] & & \\
 & & \mathtt{solve(\boxed{X},Y,0,W,s(0))} \arrow[dl] \arrow[d] & & \mathtt{solve(X,\boxed{Y},s(Z),W,s(0))} \arrow[dl] \arrow[dr] & \\
 & \mathtt{solve(0,\boxed{Y},0,W,s(0))} \arrow[dl] \arrow[dr] & \mathtt{solve(s(X),Y,0,W,s(0))} & \mathtt{solve(X,0,s(Z),W,s(0))} & & \mathtt{solve(X,s(Y),s(Z),W,s(0))} \\
\mathtt{solve(0,0,0,\boxed{W},s(0))} \arrow[d] & & \mathtt{solve(0,s(Y),0,W,s(0))} & & & \\
\mathtt{solve(0,0,0,0,s(0))} & & & & &
\end{tikzcd}$$

Fig. 3. Definitional tree for term `solve(X,Y,0+s(s(0)),W,s(0))`.

Example 9. Consider Example 2. The term t_4 is inductively sequential since

$$\begin{aligned} match_{t_4}(\mathcal{R}) = \{&\mathtt{solve(X,s(Y),s(Z),W,s(H))}, \mathtt{solve(X,0,s(Z),W,s(H))},\\ &\mathtt{solve(s(X),Y,0,W,s(H))}, \mathtt{solve(0,s(Y),0,W,s(H))},\\ &\mathtt{solve(0,0,0,0,H)}\}\end{aligned}$$

and its associated definitional tree is depicted in Figure 3.

Example 10. On the other hand, consider again Example 2 but term $t' = \mathtt{solve(X,Y,0+s(s(0)),W,H)}$. This term is not inductively sequential since all lhs's for symbol `solve` could be (potentially) matched and a rule partition is necessary, as shown in Figure 2. Then, optimal evaluation is not ensured. For instance, natural narrowing returns positions 3 and 5 as most popular candidates for narrowing, i.e. $\mathtt{m}(t') = \{3, 5\}$ where

$$\begin{array}{ll}
DP_{\mathtt{solve(X,s(Y),s(Z),W,s(H))}}(t') = \{2,3,5\} & DP_{\mathtt{solve(X,0,s(Z),W,s(H))}}(t') = \{2,3,5\}\\
DP_{\mathtt{solve(s(X),Y,0,W,s(H))}}(t') = \{1,3,5\} & DP_{\mathtt{solve(0,s(Y),0,W,s(H))}}(t') = \{1,2,3,5\}\\
DP_{\mathtt{solve(0,0,0,0,H)}}(t') = \{1,2,3,4\} & DP_{\mathtt{solve(s(X),Y,0,W,0)}}(t') = \{1,3,5\}\\
DP_{\mathtt{solve(X,0,s(Z),W,0)}}(t') = \{2,3,5\} & DP_{\mathtt{solve(0,s(Y),Z,0,0)}}(t') = \{1,2,4,5\}\\
DP_{\mathtt{solve(X,s(Y),s(Z),s(W),0)}}(t') = \{2,3,4,5\} &
\end{array}$$

The (computable) notion of an inductively sequential term is based on the idea of a definitional tree whose root pattern is not necessarily[7] of the form $f(x_1, \ldots, x_k)$ where $x_1, \ldots, x_k$ are variables. The following results are consequences of Definition 5. We first introduce the notion of neededness of a narrowing step.

In [17], *root-neededness* of a rewriting step is introduced as the basis to develop *normalizing* and *infinitary normalizing* rewrite strategies for lazy (or call-by-need) rewriting. In [13], it is proved that (almost) orthogonal strongly sequential TRSs amount to reduce root-needed redexes. Therefore, in orthogonal TRSs, every term not in normal form has a needed redex that must be reduced to compute the term's normal form.

Definition 6 (Neededness of a rewriting step). [17] *A position p of a term t is called* root-needed *if in every reduction sequence from t to a head-normal form, either $t|_p$ or a descendant of $t|_p$ is reduced. A rewriting step $t \xrightarrow{p} s$ is* root-needed *if the reduced position p is root-needed.*

[7] This is why we made more flexible the definition of a definitional tree.

In [6], it is shown that neededness of a narrowing step does not stem in a simple way from its counterpart of rewriting since some unifiers may not be strictly unavoidable to achieve a normal form. Intuitively, a narrowing step is needed no matter which unifiers might be used later in the derivation.

Definition 7 (Neededness of a narrowing step). [6] *A narrowing step* $t \rightsquigarrow_{\{p,l\to r,\sigma\}} s$ *is called* outermost-needed *(or needed) if for every substitution* $\eta \geq \sigma$, p *is the position of a root-needed or outermost-needed redex of* $\eta(t)$, *respectively.*

The notion of neededness of a narrowing or reduction step is only meaningful for orthogonal TRSs, since all redexes of a term correspond to non-overlapping rules. However, this notion is well-defined when we consider inductively sequential terms.

Proposition 1. *Let* $\mathcal{R}$ *be a TRS. If* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *is an inductively sequential term, then for all* $p \in \mathcal{P}os_{\mathcal{D}}(t)$, *left-hand sides in* $match_{t|_p}(\mathcal{R})$ *are non-overlapping.*

Now, we investigate a related problem, namely whether natural narrowing (induced by general TRSs) performs only “outermost-needed” narrowing steps when considering inductively sequential terms.

Theorem 1 (Optimality). *Let* $\mathcal{R} = (\mathcal{F}, R)$ *be a left-linear CS and* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *be an inductively sequential term which is not root-stable. Then each natural narrowing step* $t \overset{\mathtt{m}}{\rightsquigarrow}_{\{p,\sigma,l\to r\}} s$ *is outermost-needed.*

The idea behind the previous proposition and theorem is that a narrowing step from an inductively sequential term is outermost-needed whatever narrowing step we can perform later in any derivation starting from it. Note that later narrowing steps may not hold the property of outermost-neededness since we are not restricted to orthogonal TRS. However, in the case that a later step is not outermost-needed, the current step from the inductively sequential term is still unavoidable to obtain a root-stable form.

Example 11. Consider Example 2. All the steps in the narrowing sequences from t_4 are outermost-needed since they are unavoidable to achieve the normal form `False`. In concrete, the initial rewrite step is necessary to achieve the normal form and the instantiation on `Y` is unavoidable if a normal form is desired.

Moreover, we are able to prove that normal forms of $\overset{\mathtt{m}}{\rightsquigarrow}$ are root-stable forms.

Theorem 2 (Correctness). *Let* $\mathcal{R} = (\mathcal{F}, R)$ *be a left-linear TRS. If* $s \in \mathcal{T}(\mathcal{F},\mathcal{X})$ *is a* $\overset{\mathtt{m}}{\rightsquigarrow}$ *-normal form, then* s *is root-stable.*

Indeed, we provide a class of CSs called *inductively sequential preserving* with some interesting normalization properties.

Definition 8 (Inductively sequential preserving). *We say* $\mathcal{R}$ *is an* inductively sequential preserving *CS if for all rule* $l \to r \in \mathcal{R}$, *right-hand side* r *is an inductively sequential term.*

Example 12. The TRS of Example 1 is trivially inductively sequential preserving. Consider now the TRS of Example 2. This TRS is also inductively sequential preserving since rhs `solve(X,0,Z,W,H)` induces sequential evaluation in position 3, rhs `solve(X,Y,0,W,H)` induces sequential evaluation in position 1, and rhs `solve(X,Y,s(Z),W,H)` induces sequential evaluation in position 2 or 5. On the other hand, consider the following CS for the addition function between naturals:

```
 0 + N = N        N + 0 = N       s(M) + N = s(M + N)
```

This TRS is clearly non-inductively sequential preserving since rhs `s(M+N)` is a non-inductively sequential term because no definitional tree can be associated to `M+N` without a rule partition.

Note that inductively sequential CSs are trivially inductively sequential preserving CSs. The following result asserts that natural narrowing steps in an *inductively sequential preserving* CS preserve inductive sequentiality of terms.

Theorem 3. *Let $\mathcal{R} = (\mathcal{F}, R)$ be an inductively sequential preserving CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be an inductively sequential term. If $t \overset{\mathtt{m}}{\rightsquigarrow}_{\{p,\sigma,l\to r\}} s$, then s is inductively sequential.*

In concrete, this last theorem provides the basis for the completeness of natural narrowing, i.e. narrowing of inductively sequential terms within the class of inductively sequential preserving CSs provides root-stable terms.

Theorem 4 (Completeness). *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a left-linear inductively sequential preserving CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be an inductively sequential term which is not root-stable. If $t \rightsquigarrow^* s$ and $root(s) \in \mathcal{C}$ where s is the unique root-stable term in the sequence, then $t \overset{\mathtt{m}}{\rightsquigarrow}^* s$.*

Example 13. Consider again Example 2. Since Example 12 shows TRS of Example 2 is inductively sequential preserving and Example 9 proves term t_4 is inductively sequential, narrowing t_4 through natural narrowing is optimal and provides appropriate root-stable forms.

Finally, we clarify the precise relation between outermost-needed narrowing and natural narrowing.

3.2 (Weakly) Outermost-needed narrowing

The formal definition of outermost-needed narrowing for an inductively sequential CS (i.e. through a definitional tree) can be found in [6] whereas the definition for non-inductively sequential CSs can be found in [5]. The reader should note that definitional trees (as well as the necessary partition) are associated to the TRS at a first stage and remain fixed during execution time.

In strongly sequential TRSs, the order of evaluation is determined by matching automata [14]. In inductively sequential CSs, the order of evaluation is determined by definitional trees and an mapping λ, which implements the strategy by traversing definitional trees as a finite state automata to compute demanded positions to be reduced [3,4,5,6].

Definition 9 (Outermost-needed narrowing). [5,6] *Let $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ with $root(t) = f$ and its definitional tree $\mathcal{T}_f$, $\lambda(t,\mathcal{T}_f)$ is defined as the smallest set satisfying*

$$\lambda(t,\mathcal{T}_f) \ni \begin{cases} (\Lambda,\sigma,l{\to}r) & \textit{if } \pi \in \mathcal{T}_f \textit{ is a leaf, } l \to r \in \mathcal{R},\ \pi \textit{ is a renaming of } l, \\ & \textit{and } \exists\sigma : \sigma = mgu(t,l) \\ (p.q,\theta{\circ}\sigma,l{\to}r) & \textit{if } \pi \in \mathcal{T}_f \textit{ is a branch with inductive position } p, \\ & root(t|_p) = f' \in \mathcal{D},\ \exists\sigma : \sigma = mgu(t,\pi), \\ & \textit{and } (q,\theta,l \to r) \in \lambda(\sigma(t)|_p,\mathcal{T}_{f'}) \end{cases}$$

We say term t narrows by *outermost-needed narrowing* to term s (at position p and using substitution σ), denoted by $t \overset{\mathtt{n}}{\leadsto}_{\{p,l\to r,\sigma\}} s$ if $(p,\sigma,l \to r) \in \lambda(t,\mathcal{T}_{root(t)})$. For non-inductively sequential CSs, a rule partition is necessary and λ is applied to all inductively sequential subsets, where the union of all computed triples is selected.

Example 14. Consider the TRS and term t_4 of Example 2. Consider also definitional trees for symbol `solve` of Figure 2. It is clear that outermost-needed narrowing yields:

$$\begin{aligned}\lambda(\mathtt{solve(X,Y,0+2,W,1)},\mathcal{T}_{\mathtt{solve}}) = \{&(3,\{\mathtt{Y}\mapsto\mathtt{0}\}),(3,\{\mathtt{Y}\mapsto\mathtt{s(Y')}\}),\\ &(3,\{\mathtt{X}\mapsto\mathtt{0},\mathtt{Y}\mapsto\mathtt{0}\}),(3,\{\mathtt{X}\mapsto\mathtt{0},\mathtt{Y}\mapsto\mathtt{s(Y')}\}),\\ &(3,\{\mathtt{X}\mapsto\mathtt{s(X')}\})\}\end{aligned}$$

since `solve` has two definitional trees, positions 1 and 2 are the inductive positions of the root branch pattern of each definitional tree, and subsequent matching or instantiation is made until a branch whose inductive position is 3 is reached.

The following theorem establishes that natural narrowing is conservative w.r.t. outermost-needed narrowing for inductively sequential CSs.

Theorem 5. *Let $\mathcal{R}$ be a left-linear inductively sequential CS and $t,s \in \mathcal{T}(\mathcal{F},\mathcal{X})$ such that t is not root-stable. Then, $t \overset{\mathtt{n}}{\leadsto}_{\{p,\sigma,l\to r\}} s$ if and only if $t \overset{\mathtt{m}}{\leadsto}_{\{p,\sigma,l\to r\}} s$.*

Note that the condition of non root-stability of t is necessary since natural narrowing performs a more exhaustive matching test.

Example 15. Continuing Example 3. According to Definition 3, there is no demanded position for the term $t = \mathtt{10!} \div \mathtt{0}$, i.e. $DP^{\mathtt{m}}_{\mathcal{R}}(t) = \varnothing$. However, if definitional tree (a) of Figure 1 is used by outermost-needed narrowing, it cannot detect that t is a head-normal form and forces the reduction of the first argument since it blindly follows the selected definitional tree.

It is worth noting that if definitional tree (b) of Figure 1 is selected for use, this problem does not happen. However, the reader should note that if we (re)-consider the set of constructor symbols in the program as $\mathcal{C} = \{\mathtt{0},\mathtt{s},\mathtt{pred}\}$ instead of simply $\{\mathtt{0},\mathtt{s}\}$, then term $\mathtt{10!} \div \mathtt{pred(0)}$ is problematic when using definitional tree (a) and term $\mathtt{pred(0)} \div \mathtt{10!}$ is problematic when using definitional tree (b); whereas both are detected as head-normal forms by natural narrowing. Hence, this problem shows that outermost-needed narrowing is dependent on the algorithm for constructing definitional trees.

A simple but informal clarifying idea arises: natural narrowing refines outermost-needed narrowing like a strategy which dynamically calculates the definitional trees associated to each term to reduce. This does not mean that natural narrowing builds (or rebuilds) definitional trees each time a term is going to be evaluated. In fact, no definitional tree is used in the formal definition and this behavior is obtained using only the simple but natural and powerful idea of "most demanded positions".

4 Conclusions

We have provided a suitable refinement of the demandness notion associated to *outermost-needed narrowing* which integrates and also improves the one of the on-demand evaluation presented in [1]. We have also introduced an extension of *outermost-needed narrowing* called *natural narrowing.* We have shown that natural narrowing is a conservative extension of outermost-needed narrowing. In concrete, it preserves optimality for sequential parts of programs in a new class of TRSs called *inductively sequential preserving* and behaves even better than outermost-needed narrowing for the class of inductively sequential CSs in the avoidance of failing computations (see Theorems 1, 2, 3, 5 and Example 3). This paper extends the notion of inductive sequentiality from CSs and symbols to terms, which is the basis for the definition of an inductively sequential preserving CS. We have developed a prototype implementation of natural narrowing (called Natur) which is publicly available at `http://www.dsic.upv.es/users/elp/soft.html`. This prototype is implemented in Haskell (using ghc 5.04.2) and accepts OBJ programs, which have a syntax similar to modern functional (logic) programs, e.g. functional syntax used in Haskell or Curry. The prototype has been used to test the examples presented in this paper

There are still some open problems regarding the application of natural narrowing to non-inductively sequential terms within the general class of constructor systems [4].

Example 16. Consider the well-known parallel-or TRS $\mathcal{R}$ [4,17]:

```
True ∨ X = True      X ∨ True = True      False ∨ False = False
```

This program is almost orthogonal. Consider the term $t = \mathtt{X} \vee \mathtt{Y}$, which is not inductively sequential since no definitional tree can be associated to it. Positions 1 and 2 of t should be narrowed since all the rules could ever be matched. Natural narrowing detects them correctly and narrows both, i.e. $DP^{\mathtt{m}}_{\mathcal{R}}(t) = \{1, 2, 1, 2\}$. However, if we consider the following slightly different TRS $\mathcal{R}'$:

```
True ∨ X = True      X ∨ True = True      False ∨ X = X
```

Only position 1 is narrowed for term t since it is the most demanded position, i.e. $DP^{\mathtt{m}}_{\mathcal{R}'}(t) = \{1, 2, 1\}$, whereas positions 1 and 2 of t should be narrowed.

Acknowledgements. I would like to thank María Alpuente and Salvador Lucas for many helpful remarks on preliminary versions of this work.

References

1. M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving on-demand strategy annotations. In M. Baaz and A. Voronkov, editors, *Proc. of LPAR'02*, LNCS 2514, pages 1–18. Springer-Verlag, 2002.
2. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97*, ACM Sigplan Notices 32(12), pages 151–162. ACM Press, 1997.
3. S. Antoy. Definitional trees. In *Proc. of ALP'92*, LNCS 632, pages 143–157. Springer-Verlag, 1992.
4. S. Antoy. Constructor-based conditional narrowing. In H. Sondergaard, editor, *Proc. of PPDP'01*, pages 199–206. ACM Press, 2001.
5. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of ICLP'97*, pages 138–152. MIT Press, 1997.
6. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
7. S. Antoy and S. Lucas. Demandness in rewriting and narrowing. In M. Falaschi and M. Comini, editors, *Proc. of WFLP'02*, ENTCS 76. Elsevier Sciences, 2002.
8. S. Antoy and A. Middeldorp. A Sequential Strategy. *Theoretical Computer Science*, 165:75–95, 1996.
9. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical report, AIB-2001-09, RWTH Aachen, Germany, 2001.
10. I. Durand and A. Middeldorp. Decidable Call by Need Computations in Term Rewriting. In W. McCune, editor, *Proc. of CADE'97*, LNCS 1249, pages 4–18. Springer Verlag, 1997.
11. W. Fokkink, J. Kamperman, and P. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000.
12. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.
14. G. Huet and J.-J. Lévy. Computations in Orthogonal Term Rewriting Systems, Part I + II. In *Computational logic: Essays in honour of J. Alan Robinson*, pages 395–414 and 415–443. The MIT Press, Cambridge, MA, 1991.
15. S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In H. Sondergaard, editor, *Proc. of PPDP'01*, pages 82–93. ACM Press, 2001.
16. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.
17. A. Middeldorp. Call by need computations to root-stable form. In *Proc. of POPL'97*, pages 94–105. ACM Press, 1997.
18. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: the Language BABEL. *Journal of Logic Programming*, 12(3):191–223, 1992.
19. R. Sekar and I.V.Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 104(1):78–109, 1993.
20. Terese, editor. *Term Rewriting Systems.* Cambridge University Press, Cambridge, 2003.

Functional Logic Programming with Failure and Built-in Equality*

F. J. López-Fraguas and J. Sánchez-Hernández

Dep. Sistemas Informáticos y Programación, Univ. Complutense de Madrid
{fraguas,jaime}@sip.ucm.es

Abstract. Constructive failure has been proposed recently as a programming construct useful for functional logic programming, playing a role similar to that of constructive negation in logic programming. On the other hand, almost any functional logic program requires the use of some kind of equality test between expressions. We face in this work in a rigorous way the interaction of failure and equality (even for non-ground expressions), which is a non trivial issue, requiring in particular the use of disequality conditions at the level of the operational mechanism of constructive failure. As an interesting side product, we develop a novel treatment of equality and disequality in functional logic programming, by giving them a functional status, which is better suited for practice than previous proposals.

1 Introduction

Functional logic programming (see [10]) attempts to amalgamate the best features of functional and logic languages. Some works [19, 20, 15, 17, 16, 18] have addressed the subject of *failure of reduction* as a useful programming construct extending naturally to the functional logic setting the notion of negation as failure [7, 4], of great importance in logic programming.

On the other hand, almost any real program needs to use some kind of equality test between expressions. It is known [5, 14, 1] that a good treatment of equality in presence of non-ground expressions requires the use of disequality constraints. In this paper we address the combination of both issues: failure and equality.

Let us clarify our contribution with respect to previous works: in [15, 17] we extended *CRWL* [8, 9] –a well known semantic framework for functional logic programming– to obtain *CRWLF*, a proof calculus given as a semantic basis for failure in functional logic programming; those papers considered equality and disequality constraints, which were lost in [16], where *CRWLF* was reformulated by using a set oriented syntax to reflect more properly the set-valued nature of expressions in presence of non-deterministic functions typical of *CRWL* and *CRWLF*. The purely semantic, non executable approach of [16] was completed in [18] with a narrowing-like procedure able to operate with failure in presence

* Work partially supported by the Spanish project TIC2002-01167 'MELODIAS'

of non-ground expressions (we call this *constructive failure*, due to its similarity with *constructive negation* [23]); equality and disequality as specific features are again absent in [18].

The combination of constructive failure with a good treatment of equality is the **main contribution** of this paper. Equality is realized by means of a three-valued function (==) returning as possible values *true*, *false* or F, (a specific constructor to express the result of a failed computation). As we argue at the end of the next section, this is better suited to practice than to have independent equality and disequality constraints like in [5, 14, 15, 17].

The organization of the paper is as follows: the next section motivates the interest of our work; Section 3 summarizes the basic concepts about the *S*et *R*eduction *L*ogic of [16] and extends it with the inclusion of the equality function ==; Section 4 is devoted to the operational mechanism, that extends the relation *SNarr* presented in [18] with the inclusion of ==. This extension requires a specialized calculus for ==, the manipulation of disequalities and some other technical notions. We give correctness and completeness results, but proofs are omitted due to lack of space. Finally, in Section 5 we present some conclusions and the future lines of our work.

2 Failure & equality: motivation

We give here a small but not artificial example recalling the interest of constructive failure and showing that the operational treatment of equality of [18] has some drawbacks. We use in this section the concrete syntax of $\mathcal{TOY}$ [13, 1]. The functions `if _ then _` and `if _ then _ else _` are defined as usual. The results indicated in the example correspond, except for some pretty-printing, to the real results obtained with a small prototype implementing the ideas of this work.

The purpose of a computation is to evaluate a possibly non-ground expression, obtaining as answer a pair (S, C) where S is a set of values (constructor terms) and C expresses a condition for the variables in the expression. To consider sets of values instead of single values is the natural thing when dealing with failure and non-deterministic functions, as shown in [15, 17]. When S is a singleton, we sometimes skip the set braces. The condition C, in 'ordinary' functional logic programming, would be simply a substitution. In this paper, as we will see soon, C can have in addition some disequality conditions.

Example: Failure can be used to convert predicates (i.e. `true`-valued functions) into two-valued boolean functions giving values `true` or `false`. The latter are more useful, while the former are simpler to define. As an example, consider the following definition of a predicate `prefix`:

```
prefix [] Ys = true
prefix [X|Xs] [Y|Ys] = if (X==Y) then prefix Xs Ys
```

Here, `==` must be understood as *strict equality* (see e.g. [10]). To obtain from `prefix` a two-valued boolean function `fprefix` is easy using failure:

```
fprefix Xs Ys = if fails (prefix Xs Ys) then false else true
```

The function `fails` is intended to return the value `true` if its argument fails (in finite time) to be reduced to head normal form (variable or constructor rooted expression), and the value `false` if a reduction to hnf exists. Furthermore, failure should be constructive, which means that `fails` should operate soundly in presence of variables[1].

To see what happens with equality, assume first that `==` is seen as any other function, an therefore defined by rules, like in [18] . Assume also, for simplicity, that we are dealing with lists of natural numbers represented by the constructors `0` and `suc`. The rules for `==` would be:

```
    0    == 0 = true           0     == (suc Y) = false
 (suc X) == 0 = false       (suc X) == (suc Y) = X == Y
```

With these rules an expression like `fprefix Xs [Y]` produces infinitely many answers binding `Xs` to lists of natural numbers and `Y` to natural numbers.

In contrast, in this paper `==` has a specific evaluation mechanism involving disequality conditions which can be a part of answers. The virtue of disequalities is that one single disequality can encompass the same information as many substitutions. For `fprefix Xs [Y]` only four answers are obtained: `(true,{Xs=[]})`; `(false,{Xs=[Y,Z|Zs]})`; `(true,{Xs=[Y]})`; `(false,{Xs=[Z|Zs],Y/=Z})`. Notice the disequality condition in the last answer.

Some final remarks about our treatment of equality: in [5, 14] we considered equality and disequality as two independent constraints, to be used in conditional rules. One constraint `e == e'` or `e /= e'` might succeed or fail. To use a two-valued boolean equality function, say `eq`, requires to define it by means of two conditional rules: `X eq Y = true <== X==Y`; `X eq Y = false <== X /= Y`

But this has a penalty in efficiency: to evaluate an equality `e eq e'` when `e,e'` are in fact unequal (thus `e eq e'` should return `false`), the first rule tries to solve the equality, which requires to evaluate `e` and `e'` (and that might be costly); the failure of the first rule will cause backtracking to the second rule, where a new re-evaluation of `e,e'` will be done. The actual implementation of $\mathcal{TOY}$ replaces the above naive treatment of the function `eq` by a more efficient one where one equality is evaluated as far as possible without guessing in advance the value to obtain; this is done in $\mathcal{TOY}$ in a very ad-hoc way lacking any formal justification with respect to the intended semantics of equality. Here we fill the gap in a rigorous way and in the more complex setting of constructive failure, where a third value, `F`, must be considered for `==`.

3 Set Reduction Logic with Failure and ==

In this section we summarize the basic concepts of [16] about the set oriented syntax, set-expressions and the class of *CIS*-programs that we use to build the

[1] From the point of view of implementation, failure is not difficult for ground expressions. For instance, in the system Curry [11] one could use encapsulated search [12]. The problems come with constructive failure, as in the case of logic programs.

*S*et *R*eduction *L*ogic (*SRL* for short) as a semantic basis for constructive failure. In order to extend this proof calculus to deal with equality, we also use some syntactical notions about strict equality, disequality and their failure for constructed terms introduced in [17]. We remark that for the theoretical presentation we use first order notation.

3.1 Technical Preliminaries

We assume a signature $\Sigma = DC_\Sigma \cup FS_\Sigma \cup \{fails, ==\}$, where $DC_\Sigma = \bigcup_{n\in\mathbb{N}} DC^n_\Sigma$ is a set of *constructor* symbols containing at least the usual boolean ones $true$ and $false$, $FS_\Sigma = \bigcup_{n\in\mathbb{N}} FS^n_\Sigma$ is a set of *function* symbols, all of them with associated arity and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$. The functions *fails* (with arity 1) and $==$ (infix and with arity 2) do not belong to $DC \cup FS$, and will be defined by specific rules in the *SRL*-calculus.

We also assume a countable set $\mathcal{V}$ of *variable* symbols. We write $Term_\Sigma$ for the set of (total) *terms* (we say also *expressions*) built over Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset $CTerm_\Sigma$ of (total) constructor terms or (total) *cterms*, which only makes use of DC_Σ and $\mathcal{V}$. The subindex Σ will usually be omitted. Terms intend to represent possibly reducible expressions, while cterms represent data values, not further reducible.

The constant (nullary constructor) symbol F is explicitly used in our terms, so we consider the signature $\Sigma_\mathsf{F} = \Sigma \cup \{\mathsf{F}\}$. This symbol F will be used to express the result of a failed reduction to hnf. The sets $Term_\mathsf{F}$ and $CTerm_\mathsf{F}$ are defined in the natural way. The denotational semantics also uses the constant symbol $\perp$, that plays the role of the undefined value. We define $\Sigma_{\perp,\mathsf{F}} = \Sigma \cup \{\perp, \mathsf{F}\}$; the sets $Term_{\perp,\mathsf{F}}$ and $CTerm_{\perp,\mathsf{F}}$ of (partial) terms and (partial) cterms respectively, are defined in the natural way. Partial cterms represent the result of partially evaluated expressions; thus, they can be seen as approximations to the value of expressions in the denotational semantics. As usual notations we will write $X, Y, Z, ...$ for variables, c, d for constructor symbols, f, g for functions, e for terms and s, t for cterms.

The sets of substitutions $CSubst, CSubst_\mathsf{F}$ and $CSubst_{\perp,\mathsf{F}}$ are defined as mappings from $\mathcal{V}$ into $CTerm, CTerm_\mathsf{F}$ and $CTerm_{\perp,\mathsf{F}}$ respectively. We will write θ, σ, μ for general substitutions and ϵ for the identity substitution. The notation $\theta\sigma$ stands for composition of substitutions. All the considered substitutions are idempotent ($\theta\theta = \theta$). We also write $[X_1/t_1, ..., X_n/t_n]$ for the substitution that maps X_1 into t_1, ..., X_n into t_n.

Given a set of constructor symbols D, we say that the terms t and t' have a *D-clash* if they have different constructor symbols of D at the same position. We say that two tuples of cterms $t_1, ..., t_n$ and $t'_1, ..., t'_n$ have a *D-clash* if for some $i \in \{1, ..., n\}$ the cterms t_i and t'_i have a *D-clash*.

A natural *approximation ordering* $\sqsubseteq$ over $Term_{\perp,\mathsf{F}}$ can be defined as the least partial ordering over $Term_{\perp,\mathsf{F}}$ satisfying the following properties:

- $\perp \sqsubseteq e$ for all $e \in Term_{\perp,\mathsf{F}}$,

- $h(e_1, ..., e_n) \sqsubseteq h(e'_1, ..., e'_n)$, if $e_i \sqsubseteq e'_i$ for all $i \in \{1, ..., n\}$, $h \in DC \cup FS \cup \{fails, ==\} \cup \{\mathsf{F}\}$

The intended meaning of $e \sqsubseteq e'$ is that e is less defined or has less information than e'. Extending $\sqsubseteq$ to sets of terms results in the *Egli-Milner* preordering (see e.g. [21]): given $D, D' \subseteq CTerm_{\perp,\mathsf{F}}$, $D \sqsubseteq D'$ iff for all $t \in D$ there exists $t' \in D'$ with $t \sqsubseteq t'$ and for all $t' \in D'$ there exists $t \in D$ with $t \sqsubseteq t'$.

Next we define the relations $\downarrow$, $\uparrow$, $\not\downarrow$ and $\not\uparrow$, expressing, at the level of cterms, strict equality, disequality, failure of equality and failure of disequality, respectively.

Definition 1. *Relations over $CTerm_{\perp,\mathsf{F}}$*
▶ $t \downarrow t' \Leftrightarrow_{def} t = t', t \in CTerm.$ ▶ $t \uparrow t' \Leftrightarrow_{def}$ *t and t' have a DC-clash.*
▶ $t \not\downarrow t' \Leftrightarrow_{def}$ *t or t' contain* F *as subterm, or they have a DC-clash.*
▶ $\not\uparrow$ *is defined as the least symmetric relation over $CTerm_{\perp,\mathsf{F}}$ satisfying:*
i) $X \not\uparrow X$, *for all* $X \in \mathcal{V}$ *ii)* $\mathsf{F} \not\uparrow t$, *for all* $t \in CTerm_{\perp,\mathsf{F}}$
iii) if $t_1 \not\uparrow t'_1, ..., t_n \not\uparrow t'_n$ *then* $c(t_1, ..., t_n) \not\uparrow c(t'_1, ..., t'_n)$, *for* $c \in DC^n$

These relations will be extended to general expressions by means of the function $==$, whose semantic meaning will be fixed in the proof calculus *SRL* of Section 3.3. Notice that $t \uparrow t' \Rightarrow t \not\downarrow t' \Rightarrow \neg(t \downarrow t')$ and $t \downarrow t' \Rightarrow t \not\uparrow t' \Rightarrow \neg(t \uparrow t')$ but the converse implications do not hold in general. Notice also that $t \not\downarrow t'$ and $t \not\uparrow t'$ can be both true for the same t, t', and in this case neither strict equality, $\downarrow$, nor disequality, $\uparrow$, stand for t, t'. This will be used in the *SRL*-calculus (Table 1) in the rule 12, which states when $==$ fails. Such a situation happens, for instance, with $suc(0)$ and $suc(\mathsf{F})$.

3.2 Set Oriented Syntax: Set-expressions and Programs

A set-expression is a syntactical construction designed for manipulating sets of values. We extend the signature with a new countable set of *indexed variables Γ* that will usually be written as $\alpha, \beta, ...$ A (total) set-expression $\mathcal{S}$ is defined as:

$$\mathcal{S} ::= \{t\} \mid f(\bar{t}) \mid fails(\mathcal{S}_1) \mid t == t' \mid \textstyle\bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2 \mid \mathcal{S}_1 \cup \mathcal{S}_2$$

where $t, t' \in CTerm_{\mathsf{F}}$, $\bar{t} \in CTerm_{\mathsf{F}} \times ... \times CTerm_{\mathsf{F}}$, $f \in FS^n$, $\mathcal{S}_1, \mathcal{S}_2$ are set-expressions and $\alpha \in \Gamma$. We write *SetExp* for the set of (total) set-expressions. The set $SetExp_\perp$ of *partial* set-expressions has the same syntax except that the cterms $t, t', \bar{t}$ can contain $\perp$. With respect to [16] we have added here the construction $t == t'$ to the syntax of set-expressions.

Given a set-expression $\mathcal{S}$ we distinguish two sets of variables in it: the set $PV(\mathcal{S}) \subset \Gamma$ of *produced variables* (those of indexed unions of $\mathcal{S}$), and the remaining $FV(\mathcal{S}) \subset \mathcal{V}$ or *free variables.* In order to avoid variable renaming and simplify further definitions, we always assume that produced variables of a set-expression are indexed only once in the entire set-expression.

We will use substitutions mapping variables of $\mathcal{V} \cup \Gamma$ into terms built up over $\mathcal{V} \cup \Gamma$. When applying a substitution $[\ldots, X_i/t_i, \ldots, \alpha_j/s_j, \ldots]$ to a set-expression $\mathcal{S}$, it is ensured throughout the paper that it does not bind any produced variable of $\mathcal{S}$, i.e. $\{\ldots, \alpha_j, \ldots\} \cap PV(\mathcal{S}) = \emptyset$, and also that no produced variable is captured, i.e. $(\ldots \cup var(t_i) \cup \ldots \cup var(s_j) \cup \ldots) \cap PV(\mathcal{S}) = \emptyset$. It is easy to achieve these conditions with an adequate variable renaming, if necessary, of produced variables in $\mathcal{S}$.

We will also use *set-substitutions* for set-expressions: given $D = \{s_1, ..., s_n\} \subseteq CTerm_{\perp,\mathsf{F}}$ we write $\mathcal{S}[Y/D]$ as a shorthand for the distribution $\mathcal{S}[Y/s_1] \cup ... \cup \mathcal{S}[Y/s_n]$. Extending this notation, we also write $\mathcal{S}[X_1/D_1, ..., X_n/D_n]$ (where $D_1, ..., D_n \subseteq CTerm_{\perp,\mathsf{F}}$) as a shorthand for $(...(\mathcal{S}[X_1/D_1])...)[X_n/D_n]$.

It is easy to transform any expression $e \in Term_{\perp,\mathsf{F}}$ into its corresponding set-expression $\mathcal{S} \in SetExp_{\perp,\mathsf{F}}$ while preserving the semantics with respect to the appropriate proof calculus (see [16] for details). As an example, if c is a constructor symbol and f and g are function symbols, the set-expression corresponding to $f(c, g(X))$ is $\bigcup_{\alpha \in g(X)} f(c, \alpha)$.

Programs consist of rules of the form: $f(t_1, \ldots, t_n) \twoheadrightarrow \mathcal{S}$, where $f \in FS^n$, $t_i \in CTerm$ (notice that F is not allowed to appear in t_i), the tuple $(t_1, \ldots, t_n)$ is linear (each variable occurs only once), $\mathcal{S} \in SetExp$ and $FV(\mathcal{S}) \subseteq var((t_1, \ldots, t_n))$.

Like in [18], we require programs to be *C*omplete *I*nductively *S*equential Programs, or *CIS*-programs for short ([16, 18, 2]). In this kind of programs the heads of the rules must be pairwise non-overlapping and cover all the possible cases of constructor symbols. The interesting point is that for any ground tuple made of terms $t \in CTerm$ there is *exactly one* program rule that can be used for reducing a call $f(\overline{t})$. Notice that this holds for $t \in CTerm$, but not necessarily for $t \in CTerm_{\mathsf{F}}$. If t contains F, which might happen in an intermediate step of a computation, then there could be no applicable rule.

In [3, 16, 18] one can find algorithms to transform general programs into *CIS*-programs while preserving their semantics. This transformation, as well as the transformation to set oriented syntax, can be made transparent to the user in a real implementation, as indeed happens with the small prototype we have developed for this paper.

For instance, the program of the example in Section 2 becomes the following when translated into a *CIS*-program with first order set oriented syntax:

$$prefix([\,], Ys) \twoheadrightarrow \{true\} \qquad prefix([X|Xs], [\,]) \twoheadrightarrow \{\mathsf{F}\}$$

$$prefix([X|Xs], [Y|Ys]) \twoheadrightarrow \bigcup_{\beta \in X==Y} \bigcup_{\gamma \in prefix(Xs,Ys)} ifThen(\beta, \gamma)$$

$$fprefix(Xs, Ys) \twoheadrightarrow \bigcup_{\alpha \in fails(prefix(Xs,Ys))} ifThenElse(\alpha, false, true)$$

The functions `if _ then _` and `if _ then _ else _` are translated as:

$$\begin{array}{ll} ifThen(true, X) \twoheadrightarrow \{X\} & ifThenElse(true, X, Y) \twoheadrightarrow \{X\} \\ ifThen(false, X) \twoheadrightarrow \{\mathsf{F}\} & ifThenElse(false, X, Y) \twoheadrightarrow \{Y\} \end{array}$$

Notice that the original definition of `prefix` is completed in order to cover all the possible cases for the arguments. In particular, the second rule of `prefix` corresponds to a 'missing' case in the original definition, thus giving failure (the value F) in the completed one. Something similar happens with the second rule of *ifThen*.

$$\textbf{(1)}\ \frac{}{\mathcal{S} \lhd \{\bot\}}\quad \mathcal{S} \in \mathit{SetExp}_\bot \qquad\qquad \textbf{(2)}\ \frac{}{\{X\} \lhd \{X\}}\quad X \in \mathcal{V}$$

$$\textbf{(3)}\ \frac{\{t_1\} \lhd \mathcal{C}_1 \quad \{t_n\} \lhd \mathcal{C}_n}{\{c(t_1, ..., t_n)\} \lhd \{c(\overline{t}') \mid \overline{t}' \in \mathcal{C}_1 \times ... \times \mathcal{C}_n\}}\qquad c \in DC \cup \{\mathsf{F}\}$$

$$\textbf{(4)}\ \frac{\mathcal{S} \lhd \mathcal{C}}{f(\overline{t}) \lhd \mathcal{C}}\quad (f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \{R\theta \mid R = (f(\overline{t}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}, \theta \in CSubst_{\bot,\mathsf{F}}\}$$

$$\textbf{(5)}\ \frac{}{f(\overline{t}) \lhd \{\mathsf{F}\}}\quad \text{for all } (f(\overline{s}) \twoheadrightarrow \mathcal{S}') \in \mathcal{P},\ \overline{t} \text{ and } \overline{s} \text{ have a } DC \cup \{\mathsf{F}\}\text{-clash}$$

$$\textbf{(6)}\ \frac{\mathcal{S} \lhd \{\mathsf{F}\}}{\mathit{fails}(\mathcal{S}) \lhd \{\mathit{true}\}} \qquad \textbf{(7)}\ \frac{\mathcal{S} \lhd \mathcal{C}}{\mathit{fails}(\mathcal{S}) \lhd \{\mathit{false}\}}\quad \begin{array}{l}\text{if there is some } t \in \mathcal{C}\\ \text{with } t \neq \bot,\ t \neq \mathsf{F}\end{array}$$

$$\textbf{(8)}\ \frac{\mathcal{S}_1 \lhd \mathcal{C}_1 \qquad \mathcal{S}_2[\alpha/\mathcal{C}_1] \lhd \mathcal{C}}{\bigcup_{\alpha \in \mathcal{S}_1} \mathcal{S}_2 \lhd \mathcal{C}} \qquad \textbf{(9)}\ \frac{\mathcal{S}_1 \lhd \mathcal{C}_1 \qquad \mathcal{S}_2 \lhd \mathcal{C}_2}{\mathcal{S}_1 \cup \mathcal{S}_2 \lhd \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\textbf{(10)}\ \frac{}{t == t' \lhd \{\mathit{true}\}}\quad \text{if } t \downarrow t' \qquad \textbf{(11)}\ \frac{}{t == t' \lhd \{\mathit{false}\}}\quad \text{if } t \uparrow t'$$

$$\textbf{(12)}\ \frac{}{t == t' \lhd \{\mathsf{F}\}}\quad \text{if } t \not\downarrow t' \text{ and } t \not\uparrow t'$$

Table 1. Rules for *SRL*-provability

3.3 The proof calculus *SRL*

Following a well established approach in functional logic programming, we fix the semantics of programs by means of a proof calculus determining which logical statements can be derived from a program. The starting point of this semantic approach was the *CRWL* framework [8, 9], which included a calculus to prove reduction statements of the form $e \to t$, meaning that one of the possible reductions of an expression e results in the (possibly partial) value t. We extended *CRWL* to deal with failure, obtaining in [15, 17] the calculus *CRWLF*; the main insight was to replace single reduction statements by statements $e \lhd \mathcal{C}$, where $\mathcal{C}$ are sets of partial values (called *S*ufficient *A*pproximation *S*ets or *SAS*'s) corresponding to the different possibilities for reducing e.

The calculus *CRWLF* was adapted to *CIS*-programs with set oriented syntax in [16, 18], and the resulting calculus was called *SRL* (for *Set Reduction Logic*). Here we extend *SRL* to cope with equality. The new calculus, commented below, is presented in Table 1.

Rules 1 to 4 are "classical" in *CRWL(F)* [9, 17, 15, 16]. Notice that rule 4 uses a c-instance of a program rule that is unique if it exists (*CIS*-programs ensure it). If such c-instance does not exist, then by rule 5, the corresponding set-expression reduces to $\{\mathsf{F}\}$. As mentioned before when describing *CIS*-programs, this might happen because of the presence of F at some position in $f(\overline{t})$. Rules 6 and 7 establish the meaning of the function *fails*$(\mathcal{S})$ and rules 8 and 9 are natural to understand from classical set theory. Finally, rules 10, 11 and 12 define the meaning of the function $==$ as a a three-valued function: $\{\mathit{true}\}$ for the case of equality, $\{\mathit{false}\}$ for disequality, and $\{\mathsf{F}\}$ if case of failure of both. Notice that given $t, s \in CTerm_{\perp,\mathsf{F}}$ the conjunction of $t \not\downarrow s$ and $t \not\uparrow s$ means that t and s are identical except possibly at the positions (of which there must be at least one) where t or s contain the symbol F.

Given a program $\mathcal{P}$ and $\mathcal{S} \in SetExp_{\perp}$ we write $\mathcal{P} \vdash_{SRL} \mathcal{S} \lhd \mathcal{C}$, or simply $\mathcal{S} \lhd \mathcal{C}$ for sort, if the relation $\mathcal{S} \lhd \mathcal{C}$ is provable with respect to *SRL* and the program $\mathcal{P}$. The denotation of $\mathcal{S}$ (we also call it the *semantics* of $\mathcal{S}$) is defined as $[\![\mathcal{S}]\!] = \{\mathcal{C} \mid \mathcal{S} \lhd \mathcal{C}\}$. Then the denotation of a set-expression is a set of sets of (possible partial) cterms.

4 Operational Procedure: Set Narrowing with Equality

In this section we enlarge the narrowing relation *SNarr* of [18] with the equality function $==$. First, set-expressions are enriched by adding sets of disequalities to them. These disequalities must be manipulated at the operational level, so we introduce a normalization function *solve* and appropriate semantic notions that will be useful later. Then we fix the operational behavior of the equality function $==$ by means of a specific narrowing relation for it, $\underset{\theta}{\rightarrowtail}$, and we give correctness and completeness results for it with respect to *SRL*. Finally, making use of these new capabilities, we integrate the equality function into the general narrowing relation *SNarr*, and we show the corresponding correctness and completeness results, again with respect to *SRL*.

4.1 Disequality Manipulation

In order to introduce the equality function at the operational level we need an explicit manipulation of disequalities. We will work with sets δ of disequalities of the form $t \neq s$, where $t, s \in CTerm$. A first important notion is that of *solution*:

Definition 2. *Given $\delta = \{t_1 \neq s_1, \ldots, t_n \neq s_n\}$ we say that $\sigma \in CSubst_{\perp,\mathsf{F}}$ is a solution for δ, and write $\sigma \in Sol(\delta)$, if $t_1\sigma \uparrow s_1\sigma, \ldots, t_n\sigma \uparrow s_n\sigma$.*

We are particularly interested in sets of *solved forms* of disequalities of the form $X \neq t$ (with $X \not\equiv t$), where the variable X and those of t are all in $\mathcal{V}$, i.e., there are not produced variables in them. We introduce a function *solve* to obtain solved forms for sets of disequalities between cterms. As solved forms are not unique in general, *solve* returns the set of solved forms from a set of disequalities:

- ▶ $solve(\emptyset) = \{\emptyset\}$ ▶ $solve(\{X \neq X\} \cup \delta) = \emptyset$
- ▶ $solve(\{c \neq c\} \cup \delta) = \emptyset$, for any $c \in DC^0$
- ▶ $solve(\{c(\overline{t}) \neq d(\overline{t}')\} \cup \delta) = solve(\delta)$, if $c \neq d$
- ▶ $solve(\{X \neq t\} \cup \delta) = \{\{X \neq t\} \cup \delta' \mid \delta' \in solve(\delta)\}$
- ▶ $solve(\{c(t_1, \ldots, t_n) \neq c(t'_1, \ldots, t'_n)\} \cup \delta) = \{\delta_i \cup \delta' \mid \delta_i \in solve(\{t_i \neq t'_i\}), \delta' \in solve(\delta)\}$

(1) $\emptyset \underset{\epsilon}{\rightarrowtail} true$	**(2)** $X == X, C \underset{\epsilon}{\rightarrowtail} C$ if $X \notin \Gamma$
(3) $X == t, C \underset{[X/t]}{\rightarrowtail} C[X/t]$	if $X \neq t$, $X \notin \Gamma \cup var(t)$, $\Gamma \cap var(t) = \emptyset$ and t does not contain F
(4) $c(t_1, \ldots, t_n) == c(s_1, \ldots, s_n), C \underset{\epsilon}{\rightarrowtail} t_1 == s_1, \ldots, t_n == s_n, C$	
(5) $c(\overline{t}) == d(\overline{s}), C \underset{\epsilon}{\rightarrowtail} false$	
(6) $X == t, C \underset{\epsilon}{\rightarrowtail} false \mid_{X \neq t}$	if $X \neq t$, $X \notin \Gamma$, $\Gamma \cap var(t) = \emptyset$ and t does not contain F
(7) $X == c(t_1, \ldots, t_n), C \underset{[X/c(\overline{Y})]}{\rightarrowtail} (Y_1 == t_1, \ldots, Y_n == t_n, C)[X/c(\overline{Y})]$	if $X \notin \Gamma$ and $var(\overline{t}) \cup \Gamma \neq \emptyset$ or $\overline{t}$ contains F, $\overline{Y}$ fresh variables
(8) $X == c(t_1, \ldots, t_n), C \underset{[X/d(\overline{Y})]}{\rightarrowtail} false$	if $X \notin \Gamma$ and $var(\overline{t}) \cup \Gamma \neq \emptyset$ or $\overline{t}$ contains F, $\overline{Y}$ fresh variables
(9) $\mathsf{F} == t_1, \ldots, \mathsf{F} == t_n \underset{\epsilon}{\rightarrowtail} \mathsf{F}$	

Table 2. Rules for ==

The interesting property about this function is:

Proposition 1. *If* $solve(\delta) = \{\delta_1, \ldots, \delta_n\}$*, then* $Sol(\delta) = Sol(\delta_1) \cup \ldots \cup Sol(\delta_n)$.

Now, the idea is to associate a set of disequalities δ to any set-expression $\mathcal{S}$, for which we use the notation $\mathcal{S}\Box\delta$. For such set-expressions with disequalities we extend the notion of semantics in order to obtain a better way for establishing semantics equivalence between set-expressions. Given $\mathcal{S}\Box\delta$, we are interested in the semantics of $\mathcal{S}$ under total substitutions that also are solutions of δ. With this aim we introduce the notion of *hyper-semantics*:

Definition 3. *The hyper-semantics of a set-expression with disequalities* $\mathcal{S}\Box\delta$ *(with respect to a program* $\mathcal{P}$*) is defined as:* $[\![\mathcal{S}\Box\delta]\!] = \lambda\sigma \in CSubst \cap Sol(\delta).[\![\mathcal{S}\sigma]\!]$

4.2 Operational behavior of ==

The mechanism for evaluating the function == is defined by means of the relation $\underset{\theta}{\rightarrowtail}$ of Table 2. The substitution $\theta \in CSubst$ is the *computed substitution.* This relation works on a set of *constraints* of the form $\{t_1 == s_1, \ldots, t_n == s_n\}$, where $t_1, \ldots, t_n, s_1, \ldots s_n \in CTerm_{\mathsf{F}}$. As an abuse of notation we will write $t_1 == s_1, \ldots, t_n == s_n, C$ for representing the set $\{t_1 == s_1, \ldots, t_n == s_n\} \cup C$, where C is a set of constraints; it is not relevant any ordering on the constraints and the relation == is symmetric. Some comments about the rules of Table 2:

Some comments about the rules:

- rule 2 erase a variable identity, 3 is for binding, 4 for decomposition and 7 is an imitation rule. These are general rules that can be used in intermediate steps of a $\rightarrowtail$-derivation to obtain any result (*true, false* or F);

– the result *true* (equality) can be obtained by applying the general rules and finally rule 1, that stands for the empty set of constraints;
– *false* (disequality) is reached by checking a clash of constructor symbols in rule 5, or by introducing such a clash in rule 8;
– $\mathit{false}|_{X/=u}$ (conditional disequality) can be obtained in rule 6. In this case, the value *false* is conditioned to the disequality $X \neq t$. This is the point where an explicit disequality can be added as part of the solution;
– finally, F is obtained in rule 9 when all the equalities are of the form $\mathsf{F} == t$. In such case neither a equality nor a disequality can be proved.

Apart from the possible values that can return the calculus, it is possible that no rule is applicable at a given step in the derivation. This happens when there are produced variables of Γ that block the evaluation. Notice that the evaluation of an equality $t == s$ will be required from the narrowing relation *SNarr*, where this equality is a part of the full set-expression. So, $t == s$ can contain produced variables indexed in the corresponding set-expression. For example, using the function `prefix` of Section 2, the equality `prefix 0 [0,X]== Y` written as set-expression is $\bigcup_{\alpha \in prefix(0,[0,X])} \alpha == Y$. The equality $\alpha == Y$ cannot be reduced by the rules of $\rightarrowtail$ because α blocks the evaluation, i.e., it is associated to an expression $(prefix(0,[0,X]))$ that requires further evaluation.

We will use the relation $\overset{*}{\underset{\theta}{\rightarrowtail}}$ for 0 or more steps of $\rightarrowtail$, where θ indicates the composition of θ's in individual steps.

In order to simplify the relation *SNarr* of the next section we consider a uniform shape (ω, δ) for the results provided by the rules for $\rightarrowtail$, where ω is the value and δ a set of disequalities (always empty, except for the case of conditional *false*). So, with this format the possible results are: $(\mathit{true}, \emptyset)$, $(\mathit{false}, \emptyset)$, $(\mathit{false}, \{X \neq u\})$ and $(\mathsf{F}, \emptyset)$.

4.3 The operational mechanism

For selecting a redex, *SNarr* uses the notion of *context* defined as:

$$C ::= \mathcal{S} \mid [\,] \mid \mathit{fails}(C_1) \mid \textstyle\bigcup_{X \in C_1} C_2 \mid C_1 \cup C_2$$

where $\mathcal{S}$ is a set-expression, and C_1 and C_2 are contexts.

For defining the operational behavior of *fails* and also for the completeness result of Section 4.3 we need to consider the *information set* $\mathcal{S}^* \subseteq CTerm_{\perp,\mathsf{F}}$ for a set-expression $\mathcal{S}$, defined as:

$$\blacktriangleright\ (f(\bar{t}))^* = (\mathit{fails}(\mathcal{S}))^* = (t == s)^* = \{\perp\} \qquad \blacktriangleright\ (\{t\})^* = \{t\}$$
$$\blacktriangleright\ (\textstyle\bigcup_{\alpha \in \mathcal{S}'} \mathcal{S})^* = (\mathcal{S}[\alpha/\perp])^* \qquad \blacktriangleright\ (\mathcal{S}_1 \cup \mathcal{S}_2)^* = \mathcal{S}_1^* \cup \mathcal{S}_2^*$$

Now, we can define *one-narrowing-step relation:* given a program $\mathcal{P}$, two set-expressions $\mathcal{S}, \mathcal{S}' \in \mathit{SetExp}$, $\theta \in CSubst_{\mathsf{F}}$, and δ, δ' sets of disequalities in

Cntx	$C\ [\mathcal{S}]\Box\delta \underset{\theta}{\leadsto} C\theta\ [\mathcal{S}']\Box\delta'$	if $\mathcal{S}\Box\delta \underset{\theta}{\leadsto} \mathcal{S}'\Box\delta'$
Nrrw$_1$	$f(\bar{t})\Box\delta \underset{\theta\,\|_{var(\bar{t})}}{\leadsto} \mathcal{S}\theta\Box\delta'$	if $(f(\bar{s}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}$, $\theta \in CSubst_{\mathsf{F}}$ is a m.g.u. for $\bar{s}$ and $\bar{t}$ with $Dom(\theta) \cap \Gamma = \emptyset$ and $\delta' \in solve(\delta\theta)$
Nrrw$_2$	$f(\bar{t})\Box\delta \underset{\epsilon}{\leadsto} \{\mathsf{F}\}\Box\delta$	if for every rule $(f(\bar{s}) \twoheadrightarrow \mathcal{S}) \in \mathcal{P}$, $\bar{s}$ and $\bar{t}$ have a $DC \cup \{\mathsf{F}\}$-clash
Nrrw$_3$	$t == s\Box\delta \underset{\theta\,\|_{var(t)\cup var(s)}}{\leadsto} \{\omega\}\Box\delta'$	if $t == s \underset{\theta}{\overset{*}{\rightarrowtail}} (\omega, \delta'')$ and $\delta' \in solve(\delta\theta \cup \delta'')$
Fail$_1$	$fails(\mathcal{S})\Box\delta \underset{\epsilon}{\leadsto} \{true\}\Box\delta$	if $\mathcal{S}^* = \{\mathsf{F}\}$
Fail$_2$	$fails(\mathcal{S})\Box\delta \underset{\epsilon}{\leadsto} \{false\}\Box\delta$	if $\exists t \in \mathcal{S}^*\ t \neq \bot, t \neq \mathsf{F}$
Dist	$\bigcup_{\alpha\in\mathcal{S}_1\cup\mathcal{S}_2} \mathcal{S}_3\Box\delta \underset{\epsilon}{\leadsto} \bigcup_{\alpha\in\mathcal{S}_1} \mathcal{S}_3 \cup \bigcup_{\alpha\in\mathcal{S}_2} \mathcal{S}_3\Box\delta$	
Bind	$\bigcup_{\alpha\in\{t\}} \mathcal{S}\Box\delta \underset{\epsilon}{\leadsto} \mathcal{S}[\alpha/t]\Box\delta$	
Flat	$\bigcup_{\alpha\in\bigcup_{\beta\in\mathcal{S}_1}\mathcal{S}_2} \mathcal{S}_3\Box\delta \underset{\epsilon}{\leadsto} \bigcup_{\beta\in\mathcal{S}_1}\bigcup_{\alpha\in\mathcal{S}_2} \mathcal{S}_3\Box\delta$	
Elim	$\bigcup_{\alpha\in\mathcal{S}'} \mathcal{S}\Box\delta \underset{\epsilon}{\leadsto} \mathcal{S}\Box\delta$	if $\alpha \notin FV(\mathcal{S})$

Table 3. Rules for *SNarr*

solved form, a narrowing step is expressed as $\mathcal{S}\Box\delta \underset{\theta}{\leadsto} \mathcal{S}'\Box\delta'$ where the relation $\mathcal{S}\Box\delta \underset{\theta}{\leadsto} \mathcal{S}'\Box\delta$ is defined in Table 3. In the following we use *SNarr* as the name for this relation.

Rules are essentially those of [18], except for the disequality sets δ and for the rule **Nrrw**$_3$, specific for $==$. This rule performs a subcomputation with the rules for $==$ and integrates the result in the current set-expression. With respect to the other rules: **Cntx** select a redex in the set-expression, **Nrrw**$_1$ evaluates a function call by finding the appropriate (unique if it exists) applicable rule; **Nrrw**$_2$ returns a failure if such rule does not exist; rules **Fail**$_{1,2}$ evaluate the function *fails* according to the information set of the current set-expression. And the rest of rules correspond to easy manipulations from the point of view of sets.

As an example of derivation, consider again the *CIS*-program containing the functions *prefix*, *fprefix*, *ifThen* and *ifThenElse* of Section 3.2. We show one of the four possible derivations for the expression *fprefix*$(Xs, [Y])$ in Table 4 (we write *iT* for *ifThen* and *iTe* for *ifThenElse*). At each step we underline the redex in use, and we point out the rule of *SNarr* used for the step. In the case of **Nrrw**$_1$ we also point out the rule of the program used for the step.

Notice that the computed substitution is found at step (2), by applying the second rule of *prefix* to narrow the redex *prefix*$(Xs, [Y])$. At step (3), the rule **Nrrw**$_3$ throws a subcomputation for the function $==$. It succeeds with a conditional *false* that inserts the disequality $\{Y \neq B\}$ into the computation. The function *fails* is reduced to *true* by means of **Fail**$_1$ at step (7). At the end

(1)	$\underline{\mathit{fprefix}(Xs,[Y])}\Box\emptyset \underset{\epsilon}{\rightsquigarrow}$	(Nrrw$_1$-fprefix)
(2)	$\bigcup_{\alpha\in \mathit{fails}(\underline{\mathit{prefix}(Xs,[Y])})} \mathit{iTe}(\alpha, \mathit{false}, \mathit{true})\Box\emptyset \underset{Xs/[B\mid C]}{\rightsquigarrow}$	(Nrrw$_1$-prefix$_3$)
(3)	$\bigcup_{\alpha\in \mathit{fails}(\bigcup_{\beta\in \underline{B==Y}} \bigcup_{\gamma\in \mathit{prefix}(C,[\ ])} \mathit{iT}(\beta,\gamma))} \mathit{iTe}(\alpha, \mathit{false}, \mathit{true})\Box\emptyset \underset{\epsilon}{\rightsquigarrow}$	(Nrrw$_3$)
	$B == Y \underset{\epsilon}{\rightarrowtail} (\mathit{false}, \{Y \neq B\})$ (by rule 6 of $==$)	
(4)	$\bigcup_{\alpha\in \mathit{fails}(\bigcup_{\underline{\beta\in\{\mathit{false}\}}} \bigcup_{\gamma\in \mathit{prefix}(C,[\ ])} \mathit{iT}(\beta,\gamma))} \mathit{iTe}(\alpha, \mathit{false}, \mathit{true})\Box\{Y \neq B\} \underset{\epsilon}{\rightsquigarrow}$	(Bind)
(5)	$\bigcup_{\alpha\in \mathit{fails}(\bigcup_{\gamma\in \mathit{prefix}(C,[\ ])} \underline{\mathit{iT}(\mathit{false},\gamma)})} \mathit{iTe}(\alpha, \mathit{false}, \mathit{true})\Box\{Y \neq B\} \underset{\epsilon}{\rightsquigarrow}$	(Nrrw$_1$-iT$_2$)
(6)	$\bigcup_{\alpha\in \mathit{fails}(\underline{\bigcup_{\gamma\in \mathit{prefix}(C,[\ ])}\{\mathsf{F}\}})} \mathit{iTe}(\alpha, \mathit{false}, \mathit{true})\Box\{Y \neq B\} \underset{\epsilon}{\rightsquigarrow}$	(Elim)
(7)	$\bigcup_{\alpha\in \underline{\mathit{fails}(\{\mathsf{F}\})}} \mathit{iTe}(\alpha, \mathit{false}, \mathit{true})\Box\{Y \neq B\}$	(Fail$_1$)
(8)	$\bigcup_{\underline{\alpha\in\{\mathit{true}\}}} \mathit{iTe}(\alpha, \mathit{false}, \mathit{true})\Box\{Y \neq B\}$	(Bind)
(9)	$\underline{\mathit{iTe}(\mathit{true}, \mathit{false}, \mathit{true})}\Box\{Y \neq B\}$	(Nrrw$_1$-iTe$_1$)
(10)	$\{\mathit{false}\}\Box\{Y \neq B\}$	
	Final answer: $(\{\mathit{false}\}, \{Xs = [B\mid C], Y \neq B\})$	

Table 4. Derivation for `fprefix Xs [Y]`

we obtain the answer $(\{\mathit{false}\}, \{Xs = [B|C], Y/ = B\})$ as expected. The three remaining answers showed in Section 2 can be obtained in a similar way.

The correctness of *SNarr* with respect to *SRL* is easy to formulate thanks to the notion of hyper-semantics (Definition 3). Essentially it guarantees that the hyper-semantics of a set-expression is preserved under *SNarr* derivations.

Theorem 1 (Correctness of *SNarr*). *Let $\mathcal{S}$, $\mathcal{S}' \in SetExp$, $\theta \in CSubst_{\mathsf{F}}$ and δ,δ' sets of solved disequalities. Then: $\mathcal{S}\Box\delta \overset{*}{\underset{\theta}{\rightsquigarrow}} \mathcal{S}'\Box\delta' \Rightarrow [\![\mathcal{S}\theta\Box\delta']\!] = [\![\mathcal{S}'\Box\delta']\!]$*

The completeness result is quite technical and ensures that the *SAS*'s obtained by *SRL* are captured by the information provided by *SNarr*, that also finds the appropriate substitutions.

Theorem 2 (Completeness of *SNarr*). *Let $\mathcal{S}\Box\delta$ be a set-expression with disequalities and $\theta \in Sol(\delta)$. If $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \triangleleft \mathcal{C}$ then there exists a derivation $\mathcal{S}\Box\delta \overset{*}{\underset{\theta'}{\rightsquigarrow}} \mathcal{S}'\Box\delta'$, introducing a set of fresh variables Π, such that:*

► *$\theta = (\theta'\mu)\mid_{\mathcal{V}-\Pi}$, for some $\mu \in CSubst$* ► *$\mathcal{C} \sqsubseteq (\mathcal{S}'\mu)^*$* ► *$\mu \in Sol(\delta')$*

As a corollary we obtain the following result that essentially says that we have reached our goal of devising an operational procedure for constructive failure.

Corollary 1. *If $\mathcal{P} \vdash_{SRL} \mathcal{S}\theta \triangleleft \{\mathsf{F}\}$, then there exist $\theta', \mu \in CSubst$ such that $\mathcal{S}\Box\emptyset \overset{*}{\underset{\theta'}{\rightsquigarrow}} \{\mathsf{F}\}\Box\delta$, $\theta = \theta'\mu$ and $\mu \in Sol(\delta)$.*

5 Conclusions and Future Work

We have addressed in a rigorous way in this paper the problem of how to realize constructive failure in a functional logic language having a built-in equality function. The motivation was that both failure and equality are important expressive resources in functional logic programs, but there was a lack of a convenient combination of them in previous works. We were guided by the following two aims, hopefully achieved:

- We wanted our work to be technically precise from the theoretical point of view, not only at the semantic description level but also at the operational level, with results relating both levels. For the first level we give a proof calculus fixing the semantics of programs and expressions. Failure is used in programs by means of a two-valued function *fails* giving *true* or *false*, while for equality we use a three-valued function == giving as possible values *true*, *false* or F.
- We wanted failure and equality to be well-behaved from a functional logic programming perspective, which implies the ability to operate in presence of non-ground expressions. Thus, following a usual terminology for negation in logic programming, we expect failure to be *constructive*, and this, when combined with equality, required to consider disequality constraints as a part of the operational procedure, which essentially consists in set-narrowing (in the spirit of [18]) combined with a (novel) evaluation mechanism for equality, all proceeding in an ambient of disequality constraints, which can appear in answers.

The operational procedure, although complex in its description, is amenable for a quite direct implementation. We have a small prototype with which all the examples in the paper have been executed. It includes the program transformation needed to convert programs in user-friendly $\mathcal{TOY}$ syntax into *CIS*-programs with set oriented syntax. As a useful tool for experimentation the prototype includes also the possibility of tracing an execution by automatically generating and compiling a TeX file with the sequence of performed steps. The derivation in Table 4 has been obtained with the aid of this tool.

In the future, we plan to embed the prototype into the $\mathcal{TOY}$ system, and to improve its efficiency.

References

1. M. Abengózar-Carneros et al. $\mathcal{TOY}$: a multiparadigm declarative language, version 2.0. Technical report, Dep. SIP, UCM Madrid, January 2001.
2. S. Antoy. Definitional trees. In *Proc. ALP'92*, pages 143–157, Springer LNCS 632, 1992.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. PPDP'01*, pages 199–206, ACM Press, 2001.
4. K.R. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19&20:9–71, 1994.

5. P. Arenas-Sánchez, A. Gil-Luezas, F. J. López Fraguas, Combining lazy narrowing with disequality constraints. In *Proc. PLILP'94*, pages 385–399, Springer LNCS 844, 1994.
6. D. Chan. Constructive negation based on the completed database. In *Proc. ICSLP'88*, pages 111–125, 1988.
7. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322, Plenum Press, 1978.
8. J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. ESOP'96*, pages 156–172, Springer LNCS 1058, 1996.
9. J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
10. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
11. M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2000.
12. M. Hanus and F. Steiner Controlling search in declarative programs. In *Proc. PLILP/ALP'98*, pages 374–390, Springer LNCS 1490, 1998.
13. F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. RTA'99*, pages 244–247, Springer LNCS 1631, 1999.
14. F.J. López-Fraguas and J. Sánchez-Hernández. Disequalities may help to narrow. *Proc. APPIA-GULP-PRODE*, pages 89–104, 1999.
15. F.J. López-Fraguas and J. Sánchez-Hernández. Proving failure in functional logic programs. In *Proc. CL'00*, pages 179–193, Springer LNAI 1861, 2000.
16. F.J. López-Fraguas and J. Sánchez-Hernández. Functional logic programming with failure: A set-oriented view. In *Proc. LPAR'01*, pages 455–469, Springer LNAI 2250, 2001.
17. F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. To appear in *TPLP*.
18. F.J. López Fraguas and J. Sánchez Hernández. Narrowing failure in functional logic programming. In *Proc. FLOPS'02*, pages 212–227, Springer LNCS 2441, 2002.
19. J.J. Moreno-Navarro. Default rules: An extension of constructive negation for narrowing-based languages. In *Proc. ICLP'94*, pages 535–549, The MIT Press, 1994.
20. J.J. Moreno-Navarro. Extending constructive negation for partial functions in lazy functional-logic languages. In *Proc. ELP'96*, pages 213–227, Springer LNAI 1050, 1996.
21. J.C. Reynolds. *Theories of programming languages.* Cambridge Univ. Press, 1998.
22. P.J. Stuckey. Constructive negation for constraint logic programming. In *Proc. LICS'91*, pages 328–339, 1991.
23. P.J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118:12–33, 1995.

Multiple Evaluation Strategies for the Multiparadigm Programming Language Brooks

Petra Hofstedt and André Metzner

Fakultät IV – Elektrotechnik und Informatik
Technische Universität Berlin, Germany
{ame,ph}@cs.tu-berlin.de

Abstract. This paper presents the functional logic programming language Brooks which inherits from the languages Curry and BABEL, but allows the integration of different narrowing strategies. We describe briefly the abstract machine BEBAM as target for the compilation of Brooks programs together with key points of the translation process. Furthermore differences to other multiparadigm languages are discussed.

1 Introduction

While many programming languages implement only a single paradigm like functional, logic, or object-oriented programming, in recent years the interest in languages with a broader scope, integrating several paradigms simultaneously, has grown. An important subclass of those multiparadigm languages is constituted by functional logic languages which combine the two main declarative paradigms functional and logic programming. They bring together deterministic computations from the functional world and nondeterministic search operations from the logic world.

Extensive research in this area yielded several languages utilizing different evaluation mechanisms, e.g. Escher [Llo99] uses rewriting, BABEL [MNRA92] uses lazy narrowing, and Curry [HAK$^+$02] uses residuation and needed narrowing. However, combinations of different narrowing strategies in a single language appear to be rare. That is understandable, because from the viewpoint of a stand-alone language the gain in the combination of, say, lazy and needed narrowing is not obvious; the latter is usually regarded as superior to the former due to certain optimality properties [AEH00].

But declarative programs together with language evaluation mechanisms can also be considered as constraint solvers [Hof02], which allows for a smooth integration of such host languages into a bigger framework of cooperating solvers. In this context programs are not evaluated as a whole, but in a stepwise fashion under the strategic control of a meta-solver [FHM03]. It is not a priori clear how (meta-)solving strategies and evaluation mechanisms interact. We are currently investigating this topic and expect advantages by avoiding the commitment to a single evaluation mechanism.

Along these lines the need for an experimental host language platform arises. Despite certain advantages an interpreter approach has to offer, performance requirements of the overall system led us to implement an abstract machine with integrated support for innermost, lazy, and needed narrowing. Our work is conceptually based on the machine described by [Loo95] which already supports innermost and lazy narrowing. Therefore one aim of our work was to examine the requirements of needed narrowing and to figure out necessary extensions. As high-level companion for the resulting low-level machine BEBAM we developed the language Brooks that can be seen as a Haskell descendant with logic features or – at least for the features presented in this paper – as a subset of Curry.[1]

The paper is structured as follows: Section 2 gives a brief overview of Brooks and the usage of the language, and it shortly sketches the different evaluation mechanisms. The structure of the narrowing machine BEBAM is treated in Sect. 3. Section 4 is dedicated to the compilation of Brooks programs into BEBAM code. We consider functions to be evaluated using innermost and lazy narrowing in Sect. 4.1 and needed narrowing in Sect. 4.2, resp. Finally we discuss related work and conclude our paper in Sect. 5.

2 Elements of Brooks

This section gives a brief overview of Brooks as it is currently implemented by a prototypical compiler with the abstract machine BEBAM as target. In Sect. 2.1 we consider the elements of a Brooks program and sketch on different evaluation strategies in Sect. 2.2.

2.1 Programs, Data Types, and Function Definitions

A *Brooks program* consists of data type definitions and function definitions with a Haskell-like syntax.

A *data type definition* introduces a new type and a set of data constructors. Types and constructors start with capital letters. For example,

```
data Tree = Leaf Nat | Node Tree Tree
```

defines a data type `Tree` with two variants: a `Tree` is either a `Leaf` with a value of type `Nat` or a `Node` with two children of type `Tree` itself. As usual a constructor is an uninterpreted function.

Brooks has some built-in data types like `Nat` representing natural numbers and `Bool` for boolean values. Inspired by Curry there is furthermore the data type `Constraint` with the single constructor `Success` which comes in handy whenever only positive answers are of interest. This happens especially in the context of conditional rules where guard expressions are checked for satisfiability.

At present, Brooks is a monomorphic language, i.e. neither in data type definitions nor function type declarations universally quantified type variables are allowed. Due to time restrictions we left this aspect out initially, because it

[1] Brooks is the 'B.' in the name of the mathematician Haskell B. Curry and thus stands somewhere between Haskell and Curry.

was not directly relevant to our work and can be added later on in the usual way. Similar reasoning led to the omission of local definitions and lambda abstractions.

A *function* in Brooks consists of a function type declaration and a function definition. There are two different forms of function definitions related to the different evaluation strategies. We discuss the traditional rule-based form below and the other tree-based form in Sect. 2.2. A rule-based function is defined by a sequence of conditional rules, i.e. rules with left-linear constructor patterns and optional `Bool`- or `Constraint`-typed guard expressions. Rules with identical pattern parts but different guards can be bundled together to a rule block which affects code translation and gains efficiency (cf. Sect. 4.1), e.g.

$$\begin{array}{l} f\ \ pat_1 \dots pat_n \mid guard_1 = body_1 \\ f\ \ pat_1 \dots pat_n \mid guard_2 = body_2 \\ f\ \ pat'_1 \dots pat'_n \mid guard'_1 = body'_1 \\ f\ \ pat'_1 \dots pat'_n \mid guard'_2 = body'_2 \end{array} \quad \Longrightarrow \quad \begin{array}{r} f\ \ pat_1 \dots pat_n \mid guard_1 = body_1 \\ \mid guard_2 = body_2 \\ f\ \ pat'_1 \dots pat'_n \mid guard'_1 = body'_1 \\ \mid guard'_2 = body'_2\ . \end{array}$$

For reasons of confluence we require according to [Loo95] that if the left hand sides of variable disjoint variants of two rules for the same function are unifiable with a most general unifier σ then either the right-hand sides must be syntactically identical under σ or the guards must be incompatible, i.e. not simultaneously satisfiable. However, since in general we allow free variables in right-hand sides of rules as well as in guards, keeping confluence is, after all, left in the user's responsibility (cf. [Han95]). Types of free variables must be given explicitly using the `where` construct. Further features of Brooks are the *off-side rule*, *infix operators*, and *currying* along with *higher order functions*. The example program in Fig. 1 can't show every detail but may give a general impression.

```
data Seq = Nil | Nat : Seq           -- Sequences of natural numbers

(=::=) :: Seq -> Seq -> Constraint   -- Equality constraint for sequences
Nil      =::=  Nil                =  Success
(x : xs) =::= (y : ys) | x =:= y  =  xs =::= ys

(++) :: Seq -> Seq -> Seq            -- Concatenation of sequences
Nil      ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

last :: Seq -> Nat                   -- Get "logically" the last element
last xs | (ys ++ (x : Nil)) =::= xs  =  x  where ys :: Seq
                                                 x  :: Nat
```

Fig. 1. Brooks code to (inefficiently) extract the last element from a sequence using a search-oriented technique. The idea is that x is the last element of xs if there exists a sequence ys so that x appended to ys is equal to xs. Note that the equality constraint `(=:=)` for `Nat` is predefined.

2.2 Functions and Evaluation Mechanisms

Our main interest was in the implementation of an abstract machine which supports evaluation by innermost, lazy, and needed narrowing (see [Han94] for an extensive survey of narrowing strategies). *Innermost narrowing* is the simplest of the three strategies and often more efficient than lazy narrowing but is incomplete even if the corresponding term rewriting system is confluent and terminating. It does not allow for infinite data structures and impedes certain elegant programming techniques. Therefore modern implementations of functional-logic programming languages favor *lazy narrowing* or *needed narrowing*.

Needed narrowing is optimal wrt. the length of derivations and the number of computed solutions but in its original form can only be applied to a restricted class of programs, called inductively sequential systems [AEH00]. However, there are extensions of needed narrowing like weakly needed narrowing or parallel needed narrowing which go beyond this limitation [AEH97] and are deployed by Curry. For time reasons they are currently missing from Brooks, but the integration should pose no fundamental problems.

The choice between different evaluation strategies is in principle made via a compile time switch except that for the time being needed narrowing gets a special treatment. It deviates from the other strategies in that the pattern unification process doesn't work in a rule-oriented way but is guided by *definitional trees*. A definitional tree is a particular hierarchical structure to describe the order in which arguments are to be inspected. Although such trees can be generated algorithmically from rule-based function definitions we decided that for the first version of Brooks a manual specification suffices.

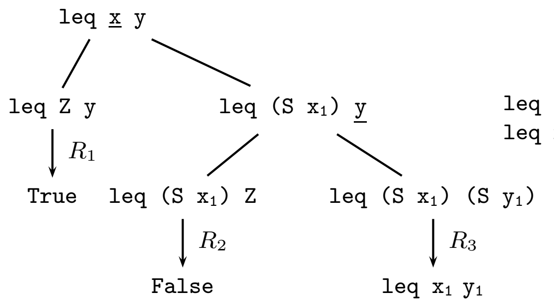
Definitional trees are formally introduced in [Ant92] and have an intuitive structure, so instead of restating definitions we content ourselves with Example 1 which moreover demonstrates the Brooks syntax for definitional trees.

Example 1. The following rule-based definition specifies the boolean function "less than or equal" for natural numbers (of the non-built-in type `Nat'`) built upon the constructors `Z` (zero) and `S` (successor):

```
data Nat' = Z | S Nat'

leq :: Nat' -> Nat' -> Bool
leq Z     y     = True          -- R1
leq (S x) Z     = False         -- R2
leq (S x) (S y) = leq x y       -- R3
```

An according definitional tree is given in Fig. 2 in graphical form and as Brooks code using the `case` construct. The needed narrowing evaluation of a call `leq` t_1 t_2 demands at first the inspection of t_1 because the patterns of all rules have constructor terms at position 1 but not at position 2. Evaluation of t_2 is not needed if t_1 is evaluated to `Z` because `y` in rule R_1 matches any argument. Otherwise the decision between the rules R_2 and R_3 is based on the head constructor of the second argument, i.e. t_2 must be evaluated.



```
leq :: Nat' -> Nat' -> Bool
leq x y = case x of
            Z    -> True
            S x1 ->
              case y of
                Z    -> False
                S y1 -> leq x1 y1
```

Fig. 2. Definitional tree for the function `leq` in graphical form and in Brooks syntax.

3 Structure of the Narrowing Machine

The target for the compilation of Brooks code is an abstract machine which can be regarded as a suitably modified version of the machine described in [Loo95] as execution environment of the language BABEL. Because both languages utilize narrowing techniques as evaluation mechanisms the choice of this machine as a starting point appeared sensible. However, BABEL is restricted to innermost and lazy narrowing and not concerned with needed narrowing, so one task was to identify necessary extensions. The resulting **B**rooks **E**xtended **B**ABEL **A**bstract **M**achine is named BEBAM in the following while BAM designates the original machine.

The (BE)BAM descends from a typical reduction machine commonly used for functional languages augmented with facilities of the Warren Abstract Machine [War83] to support features of logic languages like unification and backtracking. It combines environment-based reduction with graph-based representation of data structures. Furthermore graph nodes are used to store administrative information about suspended (lazy) calculations, partial applications of higher-order functions, variable bindings, etc. However, no graph reduction of expressions takes place.

The machine state is given by the static program store and the seven dynamic elements instruction pointer (ip), graph store (G), graph pointer (gp), data stack (ds), (environment) stack (st), environment pointer (ep), backtrack pointer (bp), and trail (tr): $\langle ip, G, gp, ds, st, ep, bp, tr\rangle \in State$. At any given time the graph store can be seen as a (partial) function which maps graph addresses to graph nodes. With *Graph* as the set of such functions, $State = \mathbb{N} \times Graph \times \mathbb{N} \times \mathbb{N}^* \times \mathbb{N}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}^*$ holds.

Allocation of graph nodes currently happens in the most straightforward way: the graph pointer gp points to the next free graph node and gets continuously adjusted, i.e. is increased by allocations and set back by backtracking which in general causes the graph to shrink. While a choicepoint exists, graph addresses of variables which get bound and of suspensions which are evaluated, are noted in the trail. These bindings and values may depend on information which is only valid in the current evaluation branch and will thus be invalidated by backtracking; the trail allows to undo affected bindings. The data stack is mainly used for purposes of parameter passing. A complete and detailed description of the BEBAM can be found in [Met02]. In the context of this paper the environment stack is of most interest.

On the one hand this stack contains argument blocks for function calls and activation records (environments) which consist of local variables, a pointer to the previously active environment, and a return address for the control flow.[2] On the other hand choicepoints are placed on the stack in order to keep track of the current machine state whenever a nondeterministic branching decision has to be made. That happens when function definitions with multiple rules and/or multiple guards are encountered (see Sect. 4). The backtrack pointer *bp* in the state always points to the currently active choicepoint. With *Env* and *CP* as the sets of all environments and choicepoints, resp., the set of possible stacks can be described as $Stack_{\mathrm{BEBAM}} = (Env \cup (CP^{+} \times Env))^{+}$.

In the BAM the stack looks essentially the same with the small but important difference that the translation of BABEL code never leads to multiple choicepoints stacked on top of each other with no intervening environment, i.e. CP^{+} is only CP in the above formula. In Brooks stacked choicepoints may be caused in two ways: by multiple rule blocks even in the case of innermost/lazy narrowing (see Sect. 4.1) and via nested evaluations in the context of anticipated substitutions in the case of needed narrowing (see Sect. 4.2).

Because in the BAM the environment which was active when the choicepoint was created is always located directly beneath the choicepoint and thus easily found, there is no need to store the environment pointer in the choicepoint. However, in the BEBAM it is generally unknown how many choicepoints have been placed on top of this environment, thus it is wise to store the environment pointer in the choicepoint to facilitate the correct reconstruction of the old state. The other choicepoint contents are the same as in the BAM and include e.g. information about the trail length, the graph pointer, the backtrack pointer, and the backtrack address at creation time. The backtrack mechanisms in the BAM and BEBAM are very similar, so we skip the details here.

However, there is a subtle point concerning local variables which is worth to mention. Backtracking in the BAM always abandons a whole rule and starts afresh with the next one, i.e. bindings of local variables made before backtracking occurs can be safely set back because no local variable can have been in a bound state at choicepoint creation time. But backtracking in the BEBAM is not limited to the rule level if needed narrowing is used, so it is quite likely that some local variables are already in a bound state at choicepoint creation time and others are not but become bound later on. Therefore at first glance it may appear to be necessary to memorize the binding situation in the choicepoint.

[2] An argument block is always associated with an environment, but strictly speaking not part of it. If no choicepoint protects the block, it is removed as soon as the function body is entered in order to avoid space wastage by (now) superfluous data. However, to avoid distraction by minor details we treat these blocks as environment parts in the following discussion.

If the state had to be reconstructed *exactly*, that would in fact be needed, but fortunately it turns out that it is not required at all to reconstruct the binding state of local variables. The structure of definitional trees ensures that when an alternative branch is taken, local variables either are automatically rebound, or are already correctly bound, or are not used anymore.

4 Compilation of Brooks Programs

A Brooks program consists of collections of data type definitions and function definitions (cf. Sect. 2). The latter can be of two different types: functions to be evaluated by innermost or lazy narrowing are defined by rules, and functions to be evaluated by needed narrowing are defined by definitional trees: $\mathit{Function} = \mathit{FunctionByRules} \cup \mathit{FunctionByTree}$. Section 4.1 is concerned with the translation of rule-based functions and introduces various aspects shared by both approaches. Afterwards, Sect. 4.2 focuses on the translation of tree-based functions.

4.1 Innermost and Lazy Narrowing

As long as only innermost and lazy narrowing are considered, Brooks code translation for the BEBAM is similar in spirit to BABEL code translation for the BAM. The most noticeable difference is an additional layer of branching due to support for rule blocks, which is illustrated in Fig. 3 with a simple example.

Nesting of choicepoints is especially advantageous when lazy narrowing is deployed. Backtracking to the start of the next rule involves a potentially expensive reevaluation of just evaluated suspensions if they depend on variables which get bound during unification of arguments and patterns (cf. Sect. 3). This is deplorable, but necessary if the pattern parts differ from rule to rule.

However, inside a rule block the patterns are invariant by definition, i.e. reevaluations would lead to the same results and should therefore be avoided. This is accomplished by a nested choicepoint which is created after successful pattern unification and ensures that backtracking affects only the guard parts until all components of the rule block have been handled.

To facilitate a more precise description of the translation process, we define formally the essence of a rule-based function definition. A function is given by its name and a sequence of rule blocks: $\mathit{FunctionByRules} = \mathcal{D} \times \mathit{Ruleblock}^{+}$, where $\mathcal{D}$ denotes the set of defined function symbols. Each rule block consists of a number of patterns according to the function arity and a sequence of guard-body pairs: $\mathit{Ruleblock} = \mathit{Pattern}^{*} \times (\mathit{Guard} \times \mathit{Expr})^{+}$. The patterns are terms $\mathcal{T}(\mathcal{C}, \mathcal{X})$ over the sets of constructor symbols $\mathcal{C}$ and the variables $\mathcal{X}$. A guard is a (possibly empty) conjunction of boolean or constraint expressions to be checked for satisfiability: $\mathit{Guard} = \mathit{Expr}^{*}$.

The translation of a Brooks function is initiated by *trans_func* (see Fig. 4) which constructs a typical chain of evaluation branches for the different rule blocks. TRY-ME-ELSE creates a new choicepoint with the given argument as

```
data Nat' = Z
          | S Nat'

f :: Nat' -> Nat'
f (S x) | g x = S Z
        | h x = S (S Z)
f Z           = Z

g, h :: Nat' -> Bool
g x = ...
h x = ...
```

```
f :     TRY-ME-ELSE f.2;
          -- pattern unification for (S x)
        TRY-ME-ELSE f.1.2;
          -- check (g x) for satisfiability
          -- body: (S Z)
        RETURN;
f.1.2 : TRUST-ME-ELSE-FAIL;
          -- check (h x) for satisfiability
          -- body: (S (S Z))
        RETURN;
f.2 :   TRUST-ME-ELSE-FAIL;
          -- pattern unification for (Z)
          -- body: (Z)
        RETURN;
```

Fig. 3. Brooks example code on the left which leads to nested choicepoint creation in the machine code on the right. Jump targets are designated with hierarchically structured symbolic labels generated during the translation process.

backtrack address corresponding to an alternative branch where evaluation continues after backtracking. Occurrences of RETRY-ME-ELSE constitute the middle part of the chain and simply modify the backtrack address of the already existing choicepoint. When no more alternatives are available at this level, i.e. at the end of the chain, TRUST-ME-ELSE-FAIL removes the choicepoint.

trans_func hands over the work to *trans_ruleblock* that matches actual arguments against formal parameter patterns by calling *trans_pat* (explained below), removes the arguments from the data stack, and calls *trans_guard_body_pairs* to translate the sequence of guards and associated rule bodies. For this purpose *trans_guard_body_pairs* builds a TRY-ME-ELSE chain quite similar to *trans_func*, leading to the choicepoint nesting discussed above. The details of expression translation are not of specific interest here.

The pattern translation via *trans_pat* (see Fig. 5) is built around the three machine instructions MATCH-VAR, MATCH-CONSTR-ELSE, and INITIATE. The generated unification code follows closely the structure of the pattern in question. MATCH-VAR unconditionally binds a local variable to an argument, and MATCH-CONSTR-ELSE tries to match the specified constructor. Such a match can only be performed against a constructor or an unbound variable, but not a suspension. Therefore, when a lazy strategy is used, INITIATE examines the situation right before the call to MATCH-CONSTR-ELSE and, if it proves necessary, temporarily sets up an environment with the local variables of the suspension and enforces an evaluation to weak head normal form (WHNF).

To give an impression of the machine's inner workings and to emphasize a useful extension of the BEBAM over the BAM, details about the instruction MATCH-CONSTR-ELSE are shown in Fig. 6. Four different cases have to be distinguished depending on the term found on top of the data stack. It can be

```
$trans\_func : FunctionByRules \rightarrow SymbolicInstruction^+$
$trans\_func(\langle f, \langle ruleblock_i \rangle_{i \in \{1,\dots,no\_of\_ruleblocks\}} \rangle) =$
  $f :$                   TRY-ME-ELSE($f.2$)
                          $trans\_ruleblock(f, f.1)(ruleblock_1)$
  $f.2 :$                 RETRY-ME-ELSE($f.3$)
                          $trans\_ruleblock(f, f.2)(ruleblock_2)$
  $\vdots$                $\vdots$
  $f.no\_of\_ruleblocks :$ TRUST-ME-ELSE-FAIL
                          $trans\_ruleblock(f, f.no\_of\_ruleblocks)(ruleblock_{no\_of\_ruleblocks})$
```

Fig. 4. Translation of a function with multiple rule blocks. The type of *trans_ruleblock* is $\mathcal{D} \times Label \rightarrow Ruleblock \rightarrow SymbolicInstruction^+$, where the first two arguments serve technical purposes further down the call chain, e.g. jump label generation and function specific lookup of variable index positions (cf. *trans_pat* in Fig. 5).

```
$trans\_pat : \mathcal{D} \times \mathbb{N} \rightarrow \mathcal{T}(\mathcal{C}, \mathcal{X}) \rightarrow SymbolicInstruction^+$
$trans\_pat(f, else\_addr)(pat) =$
  case $pat$ of
    $x$  with $x \in \mathcal{X} \rightarrow$
      MATCH-VAR($variable\_index^{(f)}(x)$)
    $c\ t_1 \dots t_n$  with $c \in \mathcal{C},\ n \geq 0 \rightarrow$
      case $compilation\_mode$ of
        Innermost/Needed Narrowing $\rightarrow$
          $\epsilon$
        Lazy Narrowing $\rightarrow$
          INITIATE
      MATCH-CONSTR-ELSE($c, n, else\_addr$)
      $trans\_pat(f, 0)(t_1)$
      $\vdots$
      $trans\_pat(f, 0)(t_n)$
```

Fig. 5. Translation of a pattern. The helper function *variable_index* maps local variable symbols to index positions in the environment. The parameter *else_addr* is always zero for innermost and lazy, but not for needed narrowing. Note that the reason why INITIATE seems to be unused for needed narrowing is simply that it has been already generated when *trans_pat* is entered (cf. *trans_tree* in Sect. 4.2).

rooted with the correct constructor which means to inspect the components; it can be an unbound logical variable which means to build a graph representation of the pattern term in a multistep process and to bind the variable; it can be a $\langle\text{HOLE}\rangle$ node which is an intermediate artefact of building a graph representation; or finally it can be a constructor-rooted term which does not match the pattern.

For innermost and lazy narrowing the last situation causes backtracking because the argument *else_addr* is always set to zero in calls to *trans_pat*. If an *else_addr* is given, no backtracking is performed but a direct jump to this address. This behavior is exploited by *trans_tree* in Sect. 4.2 and is the main difference to the otherwise similar instruction MATCH-CONSTR in the BAM which does not know about an *else_addr* and always performs backtracking in case of a mismatch.

```
$\mathcal{E}xec\,[\![\textsf{MATCH-CONSTR-ELSE}(c, n, else\_addr)]\!]\,(ip, G, gp, ds, st, ep, bp, tr) =$
  let $ds \twoheadrightarrow d_0 \cdot ds'$
  in case $G(d_0)$ of
       $\langle\text{CONSTR}, c, b_1 : \ldots : b_n\rangle \to$
         let $ds^* = dereference(G, b_1) : \ldots : dereference(G, b_n) \cdot ds'$
         in $(ip + 1, G, gp, ds^*, st, ep, bp, tr)$
         where
           $dereference : Graph \times \mathbb{N} \rightharpoonup \mathbb{N}$
           $dereference(G, addr) = \ldots$
             {- follow indirections through variable and suspension nodes -}
       $\langle\text{VAR}, ?\rangle \to$
         let $G^* = G[d_0/\langle\text{VAR}, gp\rangle, gp/\langle\text{CONSTR}, c, (gp + 1) : \ldots : (gp + n)\rangle,$
                       $(gp + 1)/\langle\text{HOLE}\rangle, \ldots, (gp + n)/\langle\text{HOLE}\rangle]$
             $ds^* = (gp + 1) : \ldots : (gp + n) \cdot ds'$
         in $(ip + 1, G^*, gp + n + 1, ds^*, st, ep, bp, tr \cdot d_0)$
       $\langle\text{HOLE}\rangle \to$
         let $G^* = G[d_0/\langle\text{CONSTR}, c, gp : \ldots : (gp + n - 1)\rangle,$
                       $gp/\langle\text{HOLE}\rangle, \ldots, (gp + n - 1)/\langle\text{HOLE}\rangle]$
             $ds^* = gp : \ldots : (gp + n - 1) \cdot ds'$
         in $(ip + 1, G^*, gp + n, ds^*, st, ep, bp, tr)$
       otherwise $\to$
         if $else\_addr = 0$ then $backtrack(ip, G, gp, ds, st, ep, bp, tr)$
                            else $(else\_addr, G, gp, ds, st, ep, bp, tr)$
```

Fig. 6. The central operation for pattern unification (some technical details concerning choicepoint handling are omitted). $G[addr/node]$ denotes the graph G modified at address *addr* to contain node *node*. Stacks grow to the left, the trail to the right.

4.2 Needed Narrowing

Independent of the way definitional trees for needed narrowing are obtained, given explicitly or generated algorithmically, function definitions eventually reflect the underlying tree structure: $\mathit{FunctionByTree} = \mathcal{D} \times \mathcal{X}^* \times \mathit{Tree}$, where the set of trees can be described inductively:

$$\langle x, \langle\langle pat_i, tree_i\rangle\rangle_{i\in\{1,\ldots,n\}}\rangle \in \mathit{Tree} \quad \text{with } x \in \mathcal{X},\ pat_1,\ldots,pat_n \in \mathcal{C}\cdot(\mathcal{X})^*, \\ tree_1,\ldots,tree_n \in \mathit{Tree} \text{ for } n \in \mathbb{N}$$

corresponding to the Brooks construct:

`case` x `of` { pat_1 `->` $tree_1$ `;` ... `;` pat_n `->` $tree_n$ }

and

$$\langle\langle guard_i, body_i\rangle\rangle_{i\in\{1,\ldots,n\}} \in \mathit{Tree} \quad \text{with } guard_1,\ldots,guard_n \in \mathit{Guard}, \\ body_1,\ldots,body_n \in \mathit{Expr} \text{ for } n \in \mathbb{N}$$

corresponding to the Brooks fragment:

... `|` $guard_1$ `->` $body_1$
⋮
`|` $guard_n$ `->` $body_n$

The translation is then driven by this structure and realized by *trans_tree*, shown in Fig. 7. The interesting part is the recursive case, where the walk through the tree is still in progress. At first the (graph address of the) relevant argument is loaded onto the data stack with the LOAD instruction which automatically dereferences variable bindings, so that either a constructor-rooted term, or an unbound logical variable, or a suspended evaluation, gets on top of the data stack. Because of an upcoming unification operation with a constructor term, a potential suspension must be evaluated to WHNF which is done by INITIATE (cf. Sect. 4.1).

The instruction SKIP-CONSTR operates in conjunction with the immediately following TRY-ME-ELSE and the MATCH-CONSTR-ELSE which is hidden in *trans_pat* (cf. Fig. 5). The TRY-ME-ELSE chain represents a branching process which may be deterministic or nondeterministic depending on the loaded argument. If it is a constructor-rooted term only a single branch has to be chosen in a deterministic way because all other branches correspond to different constructors, i.e. cannot match. If it is an unbound variable this variable will be instantiated right now according to the first branch, but this is a nondeterministic decision and may be revised later via backtracking to choose a different branch.

If SKIP-CONSTR finds a constructor term on top of the data stack, it skips the next instruction, i.e. reflects the determinism of the situation by avoiding the creation of a choicepoint. By itself that is not enough because the first branch is not necessarily the right one and a regular MATCH-CONSTR would still cause backtracking, albeit affecting the wrong choicepoint. Only the extended

```
$trans\_tree : \mathcal{D} \times Label \to Tree \to SymbolicInstruction^{+}$
$trans\_tree(f, l)(tree) =$
  case $tree$ of
    $\langle\langle guard_i, body_i\rangle\rangle_{i\in\{1,\ldots,n\}} \to$
      $trans\_guard\_body\_pairs(f, l)(tree)$
    $\langle x, \langle\langle pat_i, tree_i\rangle\rangle_{i\in\{1,\ldots,n\}}\rangle \to$
                LOAD($variable\_index^{(f)}(x)$)
                INITIATE
                SKIP-CONSTR
                TRY-ME-ELSE($l.2$)
                $trans\_pat(f, l.2.1)(pat_1)$
                $trans\_tree(f, l.1.1)(tree_1)$
      $l.2$ :     RETRY-ME-ELSE($l.3$)
      $l.2.1$ :   $trans\_pat(f, l.3.1)(pat_2)$
                $trans\_tree(f, l.2.1)(tree_2)$
      ⋮         ⋮
      $l.n$ :     TRUST-ME-ELSE-FAIL
      $l.n.1$ :   $trans\_pat(f, 0)(pat_n)$
                $trans\_tree(f, l.n.1)(tree_n)$
```

Fig. 7. Translation of a definitional tree (some minor technical details are omitted). The function *trans_pat* for pattern translation is shown in Fig. 5.

functionality of MATCH-CONSTR-ELSE makes the approach work (cf. Sect. 4.1 and Fig. 6). Backtracking is not performed in case of a mismatch, but instead the search for the matching alternative is continued by a direct jump to the next MATCH-CONSTR-ELSE.

The net effect can be understood as a kind of *if-then-elseif-*… chain which tries all possible constructors in sequence until the matching one is found. A slight optimization might be the introduction of a more direct *switch*-like construct but the idea would be the same.

Note that the mechanism also behaves as intended if the loaded argument was an unbound logical variable: SKIP-CONSTR does nothing, the choicepoint is created, and the *else_addr* of MATCH-CONSTR-ELSE won't be used because the unification of an unbound variable and a constructor always succeeds.

The presentation of translation schemes for innermost/lazy narrowing and needed narrowing side by side in the framework of a single machine points up an interesting difference in the utilization of the common choicepoint concept. Leaving guards out of account, one can say that needed narrowing uses choicepoints in a very fine-grained way, i.e. each choicepoint corresponds directly to a single variable which will get bound to different constructor terms in the course of computing all solutions. The decision to create a choicepoint is made dynamically at run time when the actual need arises to handle nondeterminism.

In contrast to that, with innermost/lazy narrowing the decision to create a choicepoint is statically made at compile time whenever a function is defined by more than a single rule. It is very well possible that a choicepoint is created although the situation turns out later to be deterministic, rendering the creation pointless with hindsight. The reason for that inefficiency is the coarse-grained usage: in a function call a single choicepoint potentially covers all unbound variables in all arguments simultaneously. In general, it is too costly to determine beforehand if variables will get bound or not, so only after the whole unification process is completed, it becomes clear if the choicepoint was indeed needed.

5 Conclusion and Related Work

With the long-term goal in mind to embed a host language with flexible evaluation strategies into a framework of cooperating constraint solvers, we attempted to explore the possibilities for an implementational integration of different narrowing strategies. As a result we developed the abstract machine BEBAM which currently supports innermost, lazy, and needed narrowing; other narrowing strategies as well as residuation may be added at a later stage.

Most functional logic languages are rather tightly associated with specific evaluation strategies, e.g. Escher is based on rewriting, BABEL is based on lazy narrowing, and Curry is based on residuation and needed narrowing. Consequently implementations of these languages tend to focus on the respective strategies or emphasize additional features.

For example, in [LK99] a stack based graph reduction machine is presented that is designed for lazy narrowing and appears to be structurally roughly comparable to the BEBAM. It additionally handles the concurrent aspects of residuation and has special support for encapsulated search operations, which are the main concern of the paper. However, the given machine description is not detailed enough to evaluate with certainty if innermost and needed narrowing are already supported or would require some modifications of the machine semantics.

The BEBAM can be regarded as an extension of a narrowing machine originally designed for innermost and lazy narrowing in the context of BABEL [Loo95]. The main contribution of the BEBAM as described in this paper is therefore the additional support for needed narrowing while avoiding a complete redesign; the idea was to extend the machine, not to start from scratch. Apart from minor issues three relatively small extensions turned out to suffice for that purpose: choicepoints need a pointer to their associated environment, choicepoint construction must be handled conditionally (SKIP-CONSTR), and pattern unification must not necessarily cause backtracking in case of a mismatch but allow for a direct jump to an alternative branch (MATCH-CONSTR-ELSE).

As companion to the BEBAM we have designed the functional logic language Brooks which has a syntax resembling a restricted form of Haskell but enables the use of logic features. We showed how compilation of Brooks code to BEBAM code can take place utilizing the specific machine facilities. In doing so we focused on the construction of TRY-ME-ELSE chains together with some aspects of pattern unification because in this regard needed narrowing differs most from innermost and lazy narrowing.

The BEBAM and a compiler for Brooks have been prototypically implemented in Haskell as a testbed for the developed ideas and to prove their validity in practice [Met02]. Of course, for serious usage the language needs to be augmented with some convenience features like polymorphism, lambda abstractions, built-in sequences, etc., which were deliberately omitted due to time constraints.

References

[AEH97] S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages. *Proc. of the 14th Int. Conf. on Logic Programming (ICLP'97)*, pages 138–152, July 1997.

[AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[Ant92] S. Antoy. Definitional Trees. In *Proc. of the 3rd Int. Conf. on Algebraic and Logic Programming (ALP'92)*, Springer LNCS 632, pages 143–157, 1992.

[FHM03] St. Frank, P. Hofstedt, and P.R. Mai. Meta-S: A strategy-oriented Meta-Solver Framework. To appear in *Proc. of the 16th Int. FLAIRS Conf.*, AAAI, 2003.

[HAK$^+$02] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, W. Lux, J.J. Moreno-Navarro, and F. Steiner. Curry – An Integrated Functional Logic Language. Language Report, Version 0.7.2 of September 16, 2002. `http://www.informatik.uni-kiel.de/~mh/curry/papers/report.pdf`.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han95] M. Hanus. On extra Variables in (Equational) Logic Programming. In *Proc. of the 12th Int. Conf. on Logic Programming (ICLP'95)*, pages 665–679. MIT Press, 1995.

[Hof02] P. Hofstedt. A General Approach for Building Constraint Languages. In *AI 2002: Advances in Artificial Intelligence*, Springer LNCS 2557, 2002.

[LK99] W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int. Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.

[Llo99] J.W. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, 1999(3), 1999.

[Loo95] R. Loogen. *Integration funktionaler und logischer Programmiersprachen.* R. Oldenbourg Verlag, 1995.

[Met02] A. Metzner. *Eine abstrakte Maschine für die (schrittweise) Abarbeitung funktional-logischer Programme.* Diploma thesis, Technische Universität Berlin, July 2002.

[MNRA92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[War83] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Menlo Park, California, October 1983.

Towards Translating Embedded Curry to C*
(Extended Abstract)

Michael Hanus Klaus Höppner Frank Huch

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,klh,fhu}@informatik.uni-kiel.de

Abstract. This paper deals with a framework to program autonomous robots in the declarative multi-paradigm language Curry. Our goal is to apply a high-level declarative programming language for the programming of embedded systems. For this purpose, we use a specialization of Curry called Embedded Curry. We show the basic ideas of our framework and an implementation that translates Embedded Curry programs into C.

Keywords: functional logic programming, process-oriented programming, embedded systems, domain-specific languages

1 Motivation

Although the advantage of declarative programming languages (e.g., functional, logic, or functional logic languages) for a high-level implementation of software systems is well known, the impact of such languages on many real world applications is quite limited. One reason for this might be the fact that many real-world applications have not only a logical (declarative) component but also demand an appropriate modeling of the dynamic behavior of a system. For instance, embedded systems become more important applications in our daily life than traditional software systems on general purpose computers, but the reactive nature of such systems seems to make it fairly difficult to use declarative languages for their implementation. We believe that this is only partially true since there are many approaches to extend declarative languages with features for reactive programming. In this paper we try to apply one such approach, the extension of the declarative multi-paradigm language Curry [11, 15] with process-oriented features [6, 7], to the programming of concrete embedded systems.

The embedded systems we consider in this paper are Lego Mindstorms robots.[1] Although these are toys intended to introduce children to the construction and programming of robots, they have all typical characteristics of embedded systems. They act autonomously, i.e., without any connection to a powerful host computer, have a limited amount of memory (32 kilobytes for operating system and application programs) and a specialized processor (Hitachi

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD/NSF under grant INT-9981317.

[1] http://mindstorms.lego.com Note that these names are registered trademarks although we do not put trademark symbols at every occurrence of them.

Fig. 1. The RCX, the "heart" of a Mindstorms robot

H8 16 MHz 8-bit microcontroller) which is not powerful compared to current general purpose computers. In order to explain the examples in this paper, we briefly survey the structure of these robots.

The Robotics Invention System (RIS) is a kit to build various kinds of robots. Its heart is the Robotic Command Explorer (RCX, see Fig. 1) containing a microprocessor, ROM, and RAM. To react to the external world, the RCX contains three input ports to which various kinds of sensors (e.g., touch, light, temperature, rotation) can be connected. To influence the external world, the RCX has three output ports for connecting actuators (e.g., motors, lamps). Programs for the RCX are usually developed on standard host computers (PCs, workstations) and cross-compiled into code for the RCX.

The RIS is distributed with a simple visual programming language (RCX code) to simplify program development for children. However, the language is quite limited and, therefore, various attempts have been made to replace the standard program development environment by more advanced systems. A popular representative of these systems is based on replacing the standard RCX firmware by a new operating system, *brickOS*,[2] and writing programs in C with specific libraries and a variant of the compiler `gcc` with a special back end for the RCX controller. The resulting programs are quite efficient and provide full access to the RCX's capabilities.

In this paper we will use the declarative multi-paradigm programming language Curry with synchronization and process-oriented features to program the RCX. The language Curry [11, 15] can be considered as a general purpose declarative programming language since it combines in a seamless way functional, logic, constraint, and concurrent programming paradigms. In order to use it for reactive programming tasks as well, different extensions have been proposed. [12] contains a proposal to extend Curry with a concept of ports (similar concepts exist also for other languages, like Erlang [4], Oz [18], etc) in order to support the high-level implementation of distributed systems. These ideas have been applied in [6] to implement a domain-specific language for process-oriented programming, inspired by the proposal in [7] to combine processes with declarative programming. The target of the latter is the application of Curry for the implementation of reactive and embedded systems. In [14] we have applied this

[2] http://www.brickos.sourceforge.net

framework to a concrete embedded system, the Mindstorms robots described above, together with a simulator. In this paper we present a compiler for a subset of this framework.

The paper is structured as follows. In the next section we sketch the necessary features of Curry. Section 3 presents an example for a Mindstorms robot and shows how to program it in Embedded Curry. Then we survey the possibilities to translate Embedded Curry programs to C in Section 4 and conclude in Section 5 with a discussion of related work.

2 Curry

In this section we survey the elements of Curry which are necessary to understand the examples in this paper. More details about Curry's computation model and a complete description of all language features can be found in [11, 15].

Curry is a multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program[3] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of data types and functions. Functions are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. However, for this paper we will not use the logical features of Curry.

Example 1. The following Curry program defines the data types of Boolean values, the polymorphic type `Maybe`, and a type `SensorMsg` (first three lines). Furthermore, it defined a test and a selector function for `Maybe`:

```
data Bool      = True    | False
data Maybe a   = Nothing | Just a
data SensorMsg = Light Int

isJust :: Maybe a -> Bool
isJust Nothing  = False
isJust (Just _) = True

fromJust :: Maybe a -> a
fromJust (Just x) = x
```

The data type declarations introduce `True` and `False` as constants of type `Bool`, `Nothing` (no value) and `Just` (some value) as the constructors for `Maybe` (`a` is a

[3] Curry has a Haskell-like syntax [17], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

type variable ranging over all types), and a constructor `Light` with an integer argument for the data type `SensorMsg`.

The (optional) type declarations ("`::`") of the functions `isJust` and `fromJust` specify that they take a `Maybe`-value as input and produce a Boolean value or a value of the (unspecified) type `a`.[4]

The operational semantics of Curry, described in detail in [11, 15], is based on an optimal evaluation strategy [2] and can be considered as a conservative extension of lazy functional programming and (concurrent) logic programming.

FlatCurry is an intermediate language that can be used as a common interface for connecting different tools for Curry programs or programs written in other (functional logic) declarative languages (e.g., Toy). In FlatCurry, all functions are defined at top level (i.e., local function declarations in source programs are globalized by lambda lifting). Furthermore, the pattern matching strategy is made explicit by the use of case expressions. Thus, a FlatCurry program basically consists of a list of data type declarations and a list of function definitions. Current Curry implementations like PAKCS [13] use FlatCurry as intermediate language so that the front end can be used with different back ends. The FlatCurry representation of the function `isJust` from Example 1 is the following:

```
isJust :: Maybe a -> Bool
isJust v0 = case v0 of
     Nothing -> False
     Just v1 -> True
```

We use FlatCurry as the source language for our compiler.

3 Programming Autonomous Robots in Embedded Curry

In this section we survey the framework for programming autonomous robots in Curry as proposed in [14]. In this framework we separate the entire programming task into two parts. To control and evaluate the connected sensors of the RCX, we use a synchronous component which generates messages for relevant sensor events (e.g., certain values are reached or exceeded, a value has changed etc.). An Embedded Curry program contains a specification that describes the connected sensors of the robot and the kind of messages that are generated for certain sensor events. The description of the actions to be executed in reaction to the sensor events are described in an asynchronous manner as a *process system.* A process system consists of a set of processes (p_1, p_2,...), a global state and mailbox for sensor messages. The behavior of a process is specified by

- a *pattern matching/condition* (on mailbox and state),
- a sequence of *actions* (to be performed when the condition is satisfied and the process is selected for execution), and

[4] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

– a *process expression* describing the further activities after executing the actions.

A process can be activated depending on the conditions on the global state and mailbox. If a process is activated (e.g., because a particular message arrives in the mailbox), it performs its actions and then the execution continues with the process expression. The check of the condition and the execution of the actions are performed as one atomic step. Process expressions are constructed similarly to process algebras [8] and defined by the following grammar:

p ::= `Terminate`	successful termination
| `Proc (p` $t_1 \ldots t_n$`)`	run process `p` with parameters $t_1 \ldots t_n$
| p_1 `>>>` p_2	sequential composition
| p_1 `<|>` p_2	parallel composition
| p_1 `<+>` p_2	nondeterministic choice

In order to specify processes in Curry following the ideas above, there are data types to define the structure of actions (`Action` *inmsg outmsg state*) and processes (`ProcExp` *inmsg outmsg state*). Furthermore, we define a *guarded process* as a pair of a list of actions and a process term:

```
data GuardedProc inmsg outmsg state =
                    GuardedProc [Action inmsg outmsg state]
                                (ProcExp inmsg outmsg state)
```

For the sake of readability, we define an infix operator to construct guarded processes:

```
acts |> pexp = GuardedProc acts pexp
```

In order to exploit the language features of Curry for the specification of process systems, we consider a *process specification* as a mapping which assigns to each mailbox (list of incoming messages) and global state a guarded process (similarly to Haskell, a `type` definition introduces a type synonym in Curry):

```
type Process inmsg outmsg state =
               [inmsg] -> state -> GuardedProc inmsg outmsg state
```

This definition has the advantage that one can use standard function definitions by pattern matching for the specification of processes, i.e., one can define the behavior of a process `p` with parameters $x_1, \ldots, x_n$ in the following form:

```
ps x1...xn mailbox state                                        (1)
                 | < condition on x1,...,xn, mailbox, state >
                 = [actions] |> process expression
```

Thus, Embedded Curry is Curry plus a library containing the type definitions sketched above and an interpreter for the processes according to the operational semantics [14]. This paper presents a compiler for a subset of Embedded Curry.

As a concrete example, we want to use this framework to program a robot that sorts bricks depending on their color (yellow and blue) into two bins. The robot consists of two more or less independent parts, a robot arm that sorts the

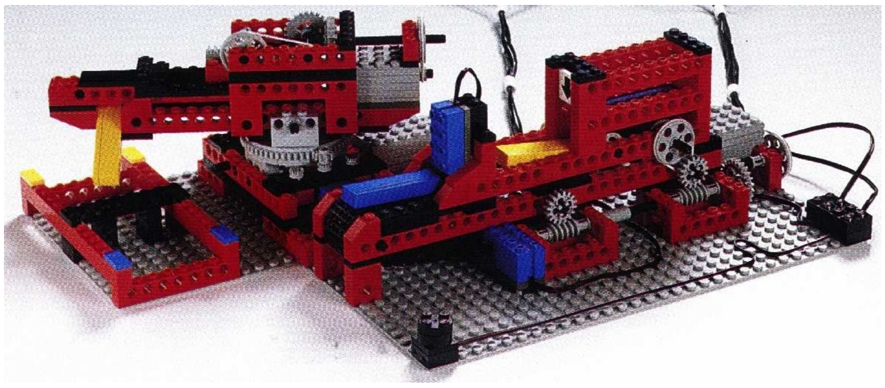


Fig. 2. Example of a robot that sorts bricks

bricks into the bins and a conveyor belt that moves the bricks from a storage to the pick-up area of the arm. On its way, the conveyor belt passes a light sensor. This sensor has two purposes: it detects when the next brick is coming and it recognizes the different colors of the bricks. Fig. 2 shows a picture of this robot. In the following we will call it "sorter".

The synchronous component for the sorter has to control the light sensor which is connected to the sensor port `In_2`. When a brick passes the light sensor, the reflected light of the brick will increase the brightness measured by the sensor. A brick is recognized if the brightness value exceeds the threshold of a blue brick th_{blue} (yellow bricks are even brighter). The specification of the synchronous component is a list of pairs, mapping a sensor port to a sensor specification. This specification is stored in the definition of a distinguished constant with name `sensors`:

```
sensors = [(In_2, LightSensor [OverThresh thblue Light])]
```

`OverThresh` states that a message should be sent if the brightness exceeds the given value. The second argument of `OverThresh` is a function which maps a brightness value (an `Int`) to a message. Here we use the constructor `Light` from Example 1.

Our implementation divides the sorter into two independent processes as well: the conveyor belt and the robotic arm. The communication between these two processes is handled through the global state. When a brick reaches the pick-up area, the belt process changes the global state to `BlueBrick` or `YellowBrick`, respectively, to inform the arm process of the detected brick. After the robotic arm has grasped the brick, the arm process changes the global state back to `Empty` to indicate that the pick-up area is empty again and the conveyor belt can be restarted.

After starting the conveyor belt, the belt process waits until the light sensor detects a brick. This event is indicated by a message `(Light` *br*`)` in the mailbox of the process system (*br* is the brightness value of the brick). Waiting for this message can be expressed by pattern matching on the mailbox. This means the process will only be activated when the first message in the mailbox is of the form (`Light` *i*):

```
waitBrickSpotted ((Light br):_) _ =
  [Deq (Light br)] |> wait t_end >>>
                      atomic [Send (MotorDir Out_C Off),
                              Set (brickType br)] >>>
                      Proc transportBrick
```

Since messages are not automatically deleted, we have to delete the message (`Light` *br*) explicitly after receiving it to prevent from multiple reactions on the same message. Deletion of messages in the mailbox is done by the action (`Deq` *msg*). Since it is performed in the initial action sequence, the matching on the mailbox and deletion of the message are performed in one atomic step.

When a brick is detected, it has not reached the pick-up area yet. Hence, the process waits for a period t_{end} of time until it stops the belt. This can be done by a call of the predefined process expression (`wait` *t*) that suspends for *t* milliseconds. Then the action (`Send` *cmd*) is used to send a command *cmd* to the actuators of the robot. The RCX has three ports for actuators: `Out_A`, `Out_B` and `Out_C`. The motor that controls the conveyor belt is connected to port `Out_C` and the command (`MotorDir Out_C Off`) will stop it. To execute a list of actions inside a process expression, one can use the function `atomic`[5]. To inform the arm process of the brick in the pick-up area, we change the global state to `BlueBrick` or `YellowBrick`. This is done by the action (`Set` *e*) that sets the global state to *e*. The function `brickType` determines the values `BlueBrick` or `YellowBrick` out of its brightness value. Because the conveyor belt should only be restarted when the pick-up area is empty, we have to wait until the global state changes back to `Empty`. This behavior can be expressed by a second process specification that is activated by pattern matching on the global state, starts the belt with the command (`MotorDir Out_C Fwd`) and then calls the first specification:

```
transportBrick _ Empty =
  [Send (MotorDir Out_C Fwd)] |> Proc waitBrickSpotted
```

The arm process is equally simple: the arm stays over the pick-up area until there is a brick ready to be picked up, i.e., the global state changes to either `BlueBrick` or `YellowBrick`. Then it closes the gripper and sets the global state back to `Empty` (this allows the conveyor belt to restart). Finally, it calls the process `putBrick` with the amount of time it takes to turn to the blue or the yellow bin. These times are provided by the function `brickTurnTime`:

```
sortBrick _ brickKind | brickKind /= Empty =
  [] |> closeGripper >>>
        atomic [Set Empty] >>>
        Proc (putBrick (brickTurnTime brickKind))
```

[5] `atomic` is implemented as follows:

```
atomic actions = Proc (\_ _ -> actions |> Terminate)
```

The function `closeGripper` defines a process expression that closes the gripper of the robotic arm.

The process `putBrick` has a parameter `time` that specifies the amount of time it takes to turn to the correct bin. It starts the motor that turns the arm (`Out_A`) and waits for `time` milliseconds before it turns the motor off. The arm should now be placed above the bin and the gripper can be opened to drop the brick. Then the arm turns back for the same amount of time to the pick-up area and restarts the arm process:

```
putBrick time _ _ =
  [Send (MotorDir Out_A Rev)] |>
          wait time >>>
          atomic [Send (MotorDir Out_A Off)] >>>
          openGripper >>>
          atomic [Send (MotorDir Out_A Fwd)] >>>
          wait time >>>
          atomic [Send (MotorDir Out_A Off)] >>>
          Proc sortBrick
```

These two processes can be combined to the whole system in two different ways: sequentially, but then the conveyor belt will not restart until the arm is back at the pick-up area, or, by using two parallel processes. Using the latter, the conveyor belt can be restarted immediately after the brick has been picked up by the arm:

```
go _ _ = [Set Empty] |> Proc transportBrick <|> Proc sortBrick
```

As a result the bricks are sorted faster.

In this example, we use time periods to determine when the bins and the pick-up area are reached. This is the simplest implementation, but different battery charge levels result in different motor speeds so that the time the arm takes to turn to a certain position can differ from one use to another. In most cases it is much better to use rotation sensors to determine the progress of a movement. This can easily be added to the implementation by replacing the wait expressions by processes listening for messages from the rotation sensors. For simplicity of the example, we do not present these details here.

4 Translation

Our goal is to use Embedded Curry to control Lego Mindstorms Robots. Due to the (speed and time) limitations of the RCX, a simple approach, like porting a complete Curry implementation (e.g., PAKCS [13]) to the RCX, will not work. This contrasts with [16] where a functional robot control language is proposed which is executed on top of Haskell running on a powerful Linux system. Our previous implementation of the process extension of Curry [6] is based on an interpreter (written in Curry) for the process expressions following the operational semantics of the process algebra [6, 7, 14]. This implementation is fairly simple (approximately 200 lines of code) but causes too much overhead. Thus, a direct compilation of the process specifications into more primitive code is necessary. As mentioned in Section 1, one of the more flexible operating systems for the RCX is brickOS. It is a POSIX-like operating system offering an appropriate infrastructure like multi-threading, semaphores for synchronization etc. Therefore, using C as an intermediate language and the brickOS compiler as back end is appropriate to implement a Curry compiler for the Lego Mindstorms. Nevertheless, many optimizations are required to map the operational semantics of Curry into features available in brickOS. However, we do not intend to cover all features of Curry (e.g., constraint solving) with our compiler since they are not necessary for our applications.

Our current implementation is restricted to a first-order functional subset of Curry with nonrecursive data declarations. Furthermore, we ignore laziness and generate strict target code. These restrictions were made for the following reasons: lazy evaluation (as well as higher-order partial applications) are memory consumptive. Furthermore, dynamic data structures need garbage collection which can be critical to guarantee constant reaction times. Finally, our experiences with the programming of Mindstorms shows that these restrictions seem to be acceptable in practice although we intend to extend our implementation in the future.

We use FlatCurry as the source code for our translation. To implement the parallel execution of processes, we use processes of the operating system (we will call them OS processes in future). An extra OS process is used for the synchronous component. The implementation of this component can be generated out of the sensor specification given in the Embedded Curry program.

To describe our translation, we start with algebraic data types which are declared as: `data` A `=` $C_0\ A_{0,0}\ \ldots\ A_{0,m_0}$ `|` $\ldots$ `|` $C_n\ A_{n,0}\ \ldots\ A_{n,m_n}$
One can easily map such a declaration to C structures (i.e., records). The structure consists of an enumeration of the constructors (C_0,...,C_n) and a union of all argument structures (structures that can store all the arguments of a certain constructor).

Function declarations of Curry are mapped to C function declarations. Because in FlatCurry all conditions have been resolved into `if`-expressions and pattern matching has been made explicit by the use of case expressions, this can be easily achieved. Case expressions in FlatCurry contain no nested patterns, i.e., all patterns are of the form $Cx_1 \ldots x_n$. Hence, they can be translated to switch statements over the enumeration of the constructors.

4.1 Global State and Mailbox

The global state and mailbox parameters of an Embedded Curry program have a special interpretation that differs from standard Curry parameters. For each check of the guards (i.e., pattern matching and rule conditions) of a process, the current values of the global state and mailbox are used (and not the values at the time of the creation of that process). This behavior can be implemented in C by using global variables for the mailbox and the global state. To guarantee mutual exclusion of the guard checking for all processes, we use a global semaphore. Before checking the guards, the processes waits for this semaphore and releases it afterwards. In the case that one of the guards is satisfied, the semaphore is released only after performing the list of actions of the selected process. The actions are translated accordingly to the operational semantics in [6, 14]. Note that all actions are performed within an initial phase of a process (see definition of `action`). Hence, in the translation all modifications of the state and the mailbox are synchronized by the global semaphore as well.

In Embedded Curry the mailbox is a list of messages. Since the mailbox is extended by the synchronous sensor component, we implement the mailbox as a queue with corresponding operations. Therefore, our translation contains an abstract data type (ADT) definition for the mailbox. For the sake of simplicity, we use a queue implementation of fixed size at the moment but this implementation can be easily replaced by a dynamic queue. To allow pattern matching on the mailbox, one has to implement it similarly to pattern matching on algebraic data types using the selector functions of the ADT mailbox. Although the mailbox can be seen as a recursive data structure, all modifications of the mailbox are made explicit by actions. Hence, no garbage arises during the execution.

4.2 Process Specifications

In Embedded Curry a process specification is a function that maps a mailbox and a state to a pair of actions and process expressions, i.e., it can be easily identified by its type. To generate an executable robot control program, these function declarations are compiled in a different way than other function declarations. Their guards (i.e., pattern matching and rule conditions) have to be checked continously until one of them is satisfied.

We translate a process specification as shown in (1) to the following general form:

```
void ps (x1,...,xn) {
  while (1) {lockState();
             <check guards sequentially. If satisfied, execute process>
             unlockState(); suspend();}}
```

Since the global state and mailbox are global variables, we do not need to pass them as parameters. The arguments x_i represent the local state of the process. To implement the continuous checking of the guards until one guard is satisfied, we use an infinite loop. Inside the loop we first lock the state (by setting the semaphore) in order to guarantee mutual exclusion on the global state and mailbox. Then we sequentially check all guards of the process specification. If none is satisfied, we unlock the state (by releasing the semaphore) and suspend this process until a change to the global state or mailbox occurs.

If one of the guards is satisfied, the corresponding code is executed and the procedure *ps* is finished by `return`. This code includes an unlocking of the state after the actions are performed.

Example 2. The process specification `transportBrick` has the following FlatCurry representation:

```
transportBrick v0 v1 = case v1 of
    Empty -> [Send (MotorDir Out_C Fwd)] |>
             (Proc waitBrickSpotted)
```

This code is translated to the C procedure:

```
void transportBrick () {
  while (1) {lockState();
             switch ((state).kind) {
               case Empty : motor_c_dir(fwd); unlockState();
                            waitBrickSpotted(); return;
               default : unlockState(); suspend();}}}
```

In this example the guard consists of a pattern matching on the global state. If the global state is `Empty`, then the motor connected to port `Out_C` is started and then the process `waitBrickSpotted` is called.

4.3 Tail Call Optimization

Reactive programs, as in embedded systems, are usually non-terminating. In a declarative programming language, non-terminating programs are implemented by recursion. At this point a problem arises because the GNU C compiler we use[6] has no general tail call optimization. If a program contains non-terminating recursion, the execution of the program would result in a stack overflow. However, this problem is only relevant for process specifications. All other function calls should terminate. A workaround for this problem is to execute a tail call of a process outside the process specification. This means that the process specification is terminated before the new process is called, i.e., the stack frame of the old process specification is deleted before the stack frame of the new process is put on the stack.

We realize this by using a small interpreter to execute all process calls of a process system. The information of a process call (i.e., the name of the called process and the parameters of the call) are stored in a data structure and passed to the interpreter procedure. The call information of the tail calls is the result of a process specification. The interpreter stores this information and executes the next call.

Our sorter example consists of five process specifications: `go`, `sortBrick`, `putBrick`, `transportBrick`, and `waitBrickSpotted`. This specification is translated into the following interpreter procedure:

```
int process_system (ProcType np) {
  while (np.next != p_Terminate) {
    switch (np.next) {
      case p_go : np = go(); break;
      case p_transportBrick : np = transportBrick(); break;
      case p_waitBrickSpotted : np = waitBrickSpotted(); break;
      case p_sortBrick : np = sortBrick(); break;
```

[6] gcc version 2.95.3

```
        case p_putBrick : np = putBrick(np.arg.putBrick.arg0);
                          break;}}}
```

The parameter `np` is used to store the information about the next process call. The attribute `next` is the name of the called process and `arg` is a union of all possible parameter combinations. The prefix `p_` is used to prevent name conflicts between functions and data type constructors. Termination of a process is handled as a call of a process `Terminate` and results in a termination of the interpreter procedure.

On the other hand, not all calls of processes must be located in tail positions. Non-tail calls are handled in the following way. Consider a call inside a sequential process expression:

`Proc` $(ps\ e_1\ \ldots\ e_n)$ `>>>` pe

Here we can use the C call stack by the translation:

`process_system(cNext_`ps`(`e_1`,...,`e_n`));` pe'

`cNext_`ps is a function that creates the data structure with the information for the call of the process ps. pe' is the remaining translation of pe.

Process calls in parallel process expressions are a bit more complicated because it is not possible to pass arbitrary parameters to a new OS process. This can be solved by supplying a global variable `nextProcess` were the call information is stored. Thus, a parallel expression containing at least one process call

`Proc` $(ps\ e_1\ \ldots\ e_n)$ `<|>` pe

is translated to:

`nextProcess = cNext_`ps`(`e_1`,...,`e_n`);`
`execi(process_system,0,NULL,PRIO_NORMAL,DEFAULT_STACK_SIZE);`
pe'

`execi` forks a new OS process that will execute the procedure `process_system` without parameters. The procedure then retrieves the call information (i.e., parameters) from `nextProcess`. One global variable `nextProcess` is sufficient for the whole program because it is only needed for the short time of the initialization of the new process. Since multiple processes could be created in parallel, we ensure mutual exclusion on the variable `nextProcess` by an additional semaphore. This means that only one process can be initialized at a time. However, the time needed to initialize a process is too short to cause any problem.

5 Conclusions

We have implemented a compiler for Embedded Curry based on the ideas described above. The compiler translates Embedded Curry programs into C programs which can then be translated into executable code for the RCX. To get an impression of the code size (which is important for embedded systems), the size of the Curry sorter program is 2947 bytes, the FlatCurry code has 24261 bytes, the generated C code has 10407 bytes, and the RCX binary code has

2986 bytes. Our previous Curry interpreter of Embedded Curry produces executables of more than 1 MB (by compilation into Prolog [3]). Since we use C as intermediate language, it is also possible to generate executable code for nearly every platform that has a C compiler. At the moment we do not have a time analysis that would allow us to guarantee certain reaction times. The reactivity depends on the fairness of the scheduling of the brickOS. Nevertheless, our experience showed that the reactivity of the resulting programs was sufficient for our applications.

For embedded system programming, synchronous languages like Esterel [5] or Lustre [9] are often used. Thus, one can also apply such languages to program embedded systems like the Lego Mindstorms robots. Actually, there already exist compilers for those languages into C. Hence, one can use the brickOS compiler to produce RCX code for these languages as well. The translation of the synchronous languages normally produces sequential code by synthesizing the control structure of the object code in the form of an extended finite automaton [10]. This is a major drawback since one does not have much control on the size of the generated code. In some cases only slight modifications in a robot specification can result in a big increase in the size of the generated code. Due to state explosion this could result in too large programs for the limited amount of memory in the RCX. Another proposal to use high-level languages for programming embedded or process-oriented systems is [16]. In contrast to our approach, they propose a functional robot control language which is executed on top of Haskell running on a powerful Linux system.

For future work, we want to enlarge the subset of Curry translatable by our compiler. First, we will investigate possibilities to compile higher-order functions by translating to first-order using partial evaluation [1] or by means of pointers. Further interesting fields of research could be the high-level specification of Embedded Curry programs. This could be integrated in a tool also supporting debugging, testing and formal verification.

References

1. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. *Journal of Functional and Logic Programming*, 2002(1), 2002.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
4. J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1996.
5. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

6. B. Braßel, M. Hanus, and F. Steiner. Embedding processes in a declarative programming language. In *Proc. Workshop on Programming Languages and Foundations of Programming*, pages 61–73. Aachener Informatik Berichte Nr. AIB-2001-11, RWTH Aachen, 2001.
7. R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pages 300–314. Springer LNAI 1861, 2000.
8. W. Fokkink. *Introduction to Process Algebra*. Springer, 2000.
9. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
10. N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 207–218. Springer LNCS 528, 1991.
11. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
12. M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.
13. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pakcs: The portland aachen kiel curry system. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2003.
14. M. Hanus and K. Höppner. Programming autonomous robots in Curry. *Electronic Notes in Theoretical Computer Science*, 76, 2002.
15. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
16. J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Proc. First International Workshop on Practical Aspects of Declarative Languages*, pages 91–105. Springer LNCS 1551, 1999.
17. S.L. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. http://www.haskell.org, 1999.
18. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.

Towards a Mobile Haskell

André Rauber Du Bois[1], Phil Trinder[1], and Hans-Wolfgang Loidl[2]

[1] School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.
`{dubois,trinder}@macs.hw.ac.uk`
[2] Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany
`hwloidl@informatik.uni-muenchen.de`

Abstract. This paper proposes a set of communication primitives for Haskell, to be used in open distributed systems, i.e. systems where multiple executing programs can interact using a predefined protocol. Functions are "first class citizens" in a functional language, hence it would be natural transfer them between programs in a distributed system. However, existing distributed Haskell extensions are limited to closed systems or restrict the set of expressions that can be communicated. The former effectively prohibits large-scale distribution, whereas the latter sacrifices key abstraction constructs. A functional language that allows the communication of functions in an open system can be seen as a *mobile computation* language, hence we call our extension *m*Haskell (Mobile Haskell). We demonstrate how the proposed communication primitives can be used to implement more powerful abstractions, such as remote evaluation, and that common patterns of communication can be encoded as higher order functions or *mobile skeletons*. The design has been validated by constructing a prototype in *Glasgow Distributed Haskell*, and a compiled implementation is under construction.

1 Introduction

The last decade has seen an exponential growth in the use of computer networks, both small and large scale networks, e.g. the Internet. One of the impacts of this growth is that almost all programming languages now have libraries or primitives for distributed programming, including functional languages as Erlang [7], Clean [27], Haskell with Ports [12], Eden [1], and GdH [23].

With a huge number of machines connected world wide, researchers start to think about the possibility of using open global networks as a single computing platform [8, 2], sharing computational power and resources. Applications exploiting these new networks require languages with capabilities for building open, distributed systems. Being a distributed system, constructs for exchanging arbitrary data structures are required. Being an open system, the language must provide ways for programs to connect and communicate with other programs and to discover new resources in the network [26].

In a functional language functions are "first class citizens"; e.g. they can be an element of a data structure, be passed as an argument to a function, or returned as the result. Thus it would be natural to expect a communication library for a functional language to allow the programmer to send functions through a network connection for example. A functional language that allows the communication of functions in an open system can be seen as a *mobile computation* language [9, 3], i.e. one that can control the placement of code and active computations across the network [16]. In particular, a mobile program can transport its state and code to another execution environment in the network, where it resumes execution [19]. This paper presents *m*Haskell (Mobile Haskell), a Haskell extension that allows the implementation of distributed mobile programs.

The paper is structured as follows: The next section describes some Haskell extensions and compares them with the concepts of mobile computation. In section 3 we describe the new primitives for distributed communication. In sections 4 and 5 we show that the basic primitives allow the implementation of more powerful abstractions (such as remote evaluation) and that useful patterns of distributed communication can be packaged as higher order functions or *mobile skeletons*, analogous to *algorithmic skeletons* in parallel computing [5]. To validate our ideas and to test the programs presented here, a prototype of our system was implemented using *Glasgow Distributed Haskell* (GdH) and is described in Section 6. In section 7 we discuss some of the issues that must be considered in a real implementation of the system. Finally we present related works, conclusions and future work.

2 Mobility in Haskell Extensions

This section compares some of the Haskell related languages with the concepts of mobile computation presented in the introduction. Mobility in other functional languages is discussed in section 8.

GpH (*Glasgow Parallel Haskell*) [24] is a conservative parallel extension of Haskell, using the parallel combinator `par` to specify parallel evaluation, and `seq` to specify sequential evaluation. Higher-level coordination is provided using *evaluation strategies*, which are higher-order polymorphic functions that use the `par` and `seq` combinators to specify the degree of evaluation of expressions.

Eden [1] is a parallel extension of Haskell that allows the definition and creation of processes. Eden extends Haskell with *process abstractions* and *process instantiations* which are analogous to function abstraction and function application. Process abstractions specify functional process schemes that represents the behaviour of a process in a purely functional way, and process instantiation is the actual creation of a process and its corresponding communication channels [17]. Eden uses a closed system model with location independence. All values are evaluated to normal form before being sent through a port.

GpH and Eden are simple and powerful extensions to the Haskell language for parallel computing. They both allow remote execution of computation, but the placement of threads is implicit. The programmer uses the `par` combinator in GpH, or process abstractions in Eden, but where and when the data will be shipped is decided by the implementation of the language.

GdH (*Glasgow Distributed Haskell*) is a distributed functional language that combines features of *Glasgow Parallel Haskell* (GpH) [24] and Concurrent Haskell [15]. GdH allows the creation of impure side effecting I/O threads using the `forkIO` command of Concurrent Haskell but it also has a `rforkIO` command for remote thread creation (of type `rforkIO :: IO () -> PEId -> IO ThreadId`). `rforkIO` receives as one of its arguments the identifier of the PE (processing element) in which the thread must be created. Thus, GdH provides the facility of creating threads across machines in a network and each location is made explicit, hence the programmer can exploit distributed resources. Location awareness is introduced in the language with the primitives `myPEId` which returns the identifier of the machine that is running the current thread and `allPEId` which gives a list of all identifiers of all machines in the program. Threads communicate through MVars that are mutable locations that can be shared by threads in a concurrent/distributed system. Communication occurs by reading/writing values from/to these mutable locations. Any type of value can be read/written from/to MVars, including functions and MVars.

GdH seems to be closer to the concepts of mobility presented before. Communication can be implemented using MVars and remote execution of computations is provided with the `revalIO` (remote evaluation) and `rforkIO` primitives. The problem in using GdH for mobile computation is that it is implemented to run on closed systems. After a GdH program starts running, no other PE can join the computation. Another problem is that its implementation relies on a *virtual shared heap* that is shared by all the machines running the computation. The algorithms used to implement this kind of structure might not scale well for large distributed systems like the Internet.

Haskell with ports or *Distributed Haskell* [12] adds to concurrent Haskell monadic operations for communication across machines in a distributed system. Thus local threads can communicate using MVars and remote threads communicate using Ports. Ports can be created, used and merged using the following commands:

```
newPort      :: IO (Port a)
writePort    :: Port a -> a -> IO ()
readPort     :: Port a -> IO a
mergePort    :: Port a -> Port b -> IO (Port (Either a b))
```

A Port is created using `newPort`. For a port to became visible to other machines it must be registered in a separate process called *postoffice* using the `registerPort` command. Once registered it can be found by other PEs using the `lookupPort` operation. Ports allow the communication of first order values including ports. All values are evaluated to normal form before sent to a remote thread.

Haskell with ports is a very interesting model to implement distributed programs in Haskell because it was designed to work on open systems. The only drawback is that the current implementation of the language restricts the values that can be sent through a port, using `writePort`, to the basic types and types that can instantiate the `Show` class. Furthermore, the types of the messages which can be received with `readPort` must be an instance of the `Read` class. The reason for these restrictions is that the values of the messages are converted to strings in order to be sent over the network [12].

3 Proposed Communication Primitives

MVars are used for communication between concurrent threads in Concurrent Haskell and for distributed communication in GdH, but their implementation relies on a closed system.

Haskell with Ports on the other hand, extends Concurrent Haskell with monadic operations for communication between machines in a distributed open system. Thus local threads can still communicate using MVars and remote threads communicate using *ports*.

We propose in this text a set of primitives to be used for distributed communication in Haskell. These primitives are similar to those of Haskell with Ports, but without the restriction of communicating just first-order values. We will call our primitives MChannels, *Mobile Channels* (figure 1), to avoid confusion with Concurrent Haskell channels or Distributed Haskell ports.

```
data MChannel a      -- abstract
type HostName = String
type ChanName = String

newMChannel       :: IO (MChannel a)
writeMChannel     :: MChannel a -> a -> IO ()
readMChannel      :: MChannel a -> IO a
registerMChannel  :: MChannel a -> ChanName -> IO ()
unregisterMChannel:: MChannel a -> IO()
lookupMChannel    :: HostName -> ChanName -> IO (MChannel a)
```

Fig. 1. Mobile Channels

The `newMChannel` function is used to create a mobile channel and the functions `writeMChannel` and `readMChannel` are used to write/read data from/to a channel. The functions `registerMChannel` and `unregisterMChannel` register/unregister channels in a name server. Once registered, a channel can be found by other programs using `lookupMChannel` which retrieves a mobile channel from the name server. A name server is always running on every machine of the system and a channel is always registered in the local name server with the `registerMChannel` function.

When a value is sent through a channel, it is evaluated to *normal form* before communication occurs. The reason for that is because Lazy Evaluation imposes some complications for reasoning about the communication of mobile programs. Consider the following example:

```
let
 (a,b,c) = f x
in
if a then
  writeMChannel ch b
```

Suppose that in the tuple returned by `f x` the first value `a` is a Boolean, the second (`b`) an integer, and the third `c` is a really large data structure (i.e. a big tree). Based on the value of `a` we choose if we want to send the integer `b` (and only `b`) to a remote host. In the example, it seems that the only value being sent is the integer, but because of lazy evaluation that is not what happens. At the point where `writeMChannel` is performed, the value `b` is represented in the heap as the function that selects the second value of a tuple applied to the whole tuple. If `writeMChannel` does not evaluate its argument before communication, the whole value is communicated and is difficult to see that in the Haskell code.

3.1 Sharing Properties

Modern non-strict functional languages usually have their implementation based on a graph reduction machine where a program is represented as a graph and the evaluation of the program is performed by reducing the graph. Programs are represented as graphs to ensure that shared expressions are evaluated only once.

If we try to maintain sharing between nodes in our distributed system this will result in a large number of extra-messages and call-backs to the machines involved in the computation (to request structures that were being evaluated somewhere else or to update this structures) that the programmer of the system did not know about. In a typical mobile application, the client will receive some code from a channel and then the machine can be disconnected from the network while the computation is being executed (consider a handheld or a notebook). If we preserve sharing, is difficult to tell when a machine can be disconnected, because even though the computation is not being executed anymore, the result might be needed by some other application that shared the same graph structure.

The problem is also partially solved by making the primitives strict because then expressions will be evaluated only once and only the result of the evaluation is communicated.

3.2 Discovering Resources

In figure 2, we propose three primitives for resource discovery and registration. All machines running mobile Haskell programs must also run a registration service for resources. The scope for the primitives for resource discovery is only the

```
type ResName = String

registerRes :: a -> ResName -> IO ()
unregisterRes :: ResName -> IO ()
lookupRes :: ResName -> IO (Maybe a)
```

Fig. 2. Primitives for resource discovery

machine where they are called, thus they will only return values registered in the local server. The `registerRes` function takes a name (`ResName`) and a resource (of type `a`) and registers this resource with the name given. `unregisterRes` unregisters a resource associated with a name and `lookupRes` takes a `ResName` and returns a resource registered with that name. To avoid a type clash, if the programmer wants to register resources with different types, she has to define an abstract data type that will hold the different values that can be registered.

A better way to treat these possible type clashes would be to use dynamic types like Clean's *Dynamics* [22], but at the moment there is no complete implementation of it in any of the Haskell compilers.

4 Mobile Examples

4.1 Remote Evaluation

In the *remote evaluation* [28] paradigm, a computation is sent from a host A to a remote host B in order to use the resources available in B. It is straightforward to implement *remote evaluation* using mobile channels. Suppose that every machine running the system also runs a remote evaluation server that can be implemented like this:

```
remoteEvaluationServer myname = do
                          ch <- newMChannel
                          registerMChannel ch myname
                          loop ch
                           where
                              loop ch = do
                                      io <-readMChannel ch
                                      forkIO io
                                      loop ch
```

The `remoteEvaluation` server takes as argument the name of the current machine and creates a channel with this name. After that, it keeps reading values from the channel and forking threads with the IO actions received. Threads are forked using the `forkIO` function of Concurrent Haskell and the `do` notation in the example is a special syntax for monadic computations [14].

If all machines in the system are running this server, we can implement our remote evaluation primitive as follows:

```
rEval :: IO () -> HostName -> IO ()
rEval io host = do
                 ch <- lookupMChannel host host
                 writeMChannel ch io
```

In fact, remote evaluation usually returns a value as the result of the computation. This is the difference between our `rEval` function and the `revalIO` primitive of GdH. The same behaviour could be easily added to `rEval` by creating another channel to return the result back.

4.2 Simple Code mobility

In Figures 3 and 4 we present a simple example using the new primitives. The program starts on a machine called `ushas` (Figure 4) and "migrates" to a machine called `lxtrinder`, using the `rEval` function. When the thread `mobileCode` is spawned on `lxtrinder`, it obtains the load of the machine using a local function called `getLoad`, that was previously registered in the resource server (Figure 3), and then sends the result back to `ushas` where it is printed.

```
getLoad :: IO Int
getLoad = (...)

main = do
         registerRes getLoad "getLoad"
```

Fig. 3. Code on lxtrinder

4.3 Distributed Information Retrieval

Distributed information retrieval applications gather information matching some specified criteria from information sources dispersed in the network. This kind of application has been considered "the killer application" for mobile languages [9].

The mobile program in Figure 6 visits the machines of a network to calculate the total load of the network (see Figure 5). It is an extension to the previous program: Instead of visiting only one machine in the network it visits a collection of machines.

The program visits all machines in the list `listofmachines` and uses the IO action `getLoad` (of type `IO Int`) locally on all machines. This function must be registered in the name server on all the machines in the `listofmachines`.

```
main = do
        mch <- newMChannel
        rEval "lxtrinder.hw.ac.uk" (mobileCode mch)
        load <- readMChannel mch
        print ("Load on lxtrinder: "++ show (load::Int))
        where
        mobileCode mch = do
             func <- lookupRes  "getLoad"
             case func of
              Just gL   -> do
                              load <-gL
                              writeMChannel mch load
              _         -> recoverError

        recoverError = (...)
```

Fig. 4. Code on ushas

5 Patterns of Mobile Computation

The behaviour of "visiting a collection of machines to perform some action and then return a result", as used in Figure 6, is a very common pattern of mobile computation. We can use the facilities provided by functional languages, in particular higher-order functions, to implement a *skeleton* [5] that abstracts this behaviour.

```
visitSke :: MChannel a -> IO a -> (a -> a -> a) -> a ->
                                                [HostName] -> IO ()
visitSke mch action op load []      = writeMChannel mch load
visitSke mch action op load (x:xs)  =
                          reEval x (code mch action op load xs)
      where code mch' action' op' load' list = do
            load2 <- action'
            visitSke mch' action' op' (op' load2 load') list
```

Using the `visitSke` function we can write the previous program like this:

```
visitSke mch action (+) 0 listofmachines

action = do
          func <- lookupRes  "getLoad"
          case func of
            Just gL -> do
                         load <- gL
                         return load
```

If we want to collect the load of the machines to compute the total load afterwards we just need to change the arguments of the skeleton:

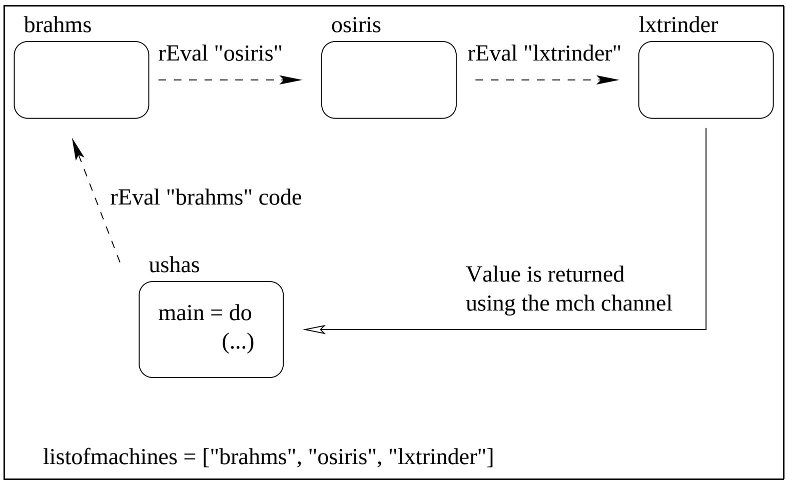


Fig. 5. Distributed Information Retrieval

```
visitSke mch action (++) [] listofmachines
```

6 A Prototype Implementation

To validate the design and to test the examples presented in this text, a prototype of *m*Haskell was implemented using GdH.

In GdH, inter-thread communication is through values that are shared between threads, e.g. an MVar, and there is no other way of establishing a connection between two remote/local threads after their creation.

In our prototype implementation mobile channels are represented as simple MVars and to simulate an open system in GdH we implemented a name server where programs running on different PEs can register their MChannels.

In the example of Figure 7, is possible to see how the GdH implementation of the system works. First an `MChannel` is created on a machine called `ushas` and it is registered with the name `"myC"` in the name server. The name server receives a reference to the MVar created on `ushas` and keeps it in a table together with its name. After the registration process has finished the PE called `osiris` can send a message to the name server asking for the channel that was registered with the name `"myC"` using the `lookupMChannel` command.

How do the primitives find the name server? Threads in GdH can only communicate if they have a reference to a value that is shared between them. We solved the problem by adding an extra parameter to the remote communication primitives that is the channel in which the name server waits for connections. This channel is a *Concurrent Haskell* channel, that is implemented as a linked list of MVars, hence the name server can handle more than one client sending requests at the same time. Programming with this extra parameter can be cum-

```
main = do
      mch <- newMChannel
      mobileCode mch 0 listofmachines
      resp <- readMChannel mch
      print ("Total Load: " ++ show resp)

mobileCode :: MChannel Int -> Int -> [HostName] -> IO ()
mobileCode mch load []      = writeMChannel mch load
mobileCode mch load (x:xs)  = rEval x (code mch load xs)
      where code mch' load' list = do
              func <- lookupRes  "getLoad"
                case func of
                    Just gL       -> do
                                    load2 <- gL
                                    mobileCode mch' (load' + load2) list
```

Fig. 6. Collecting Load Information from several Machines

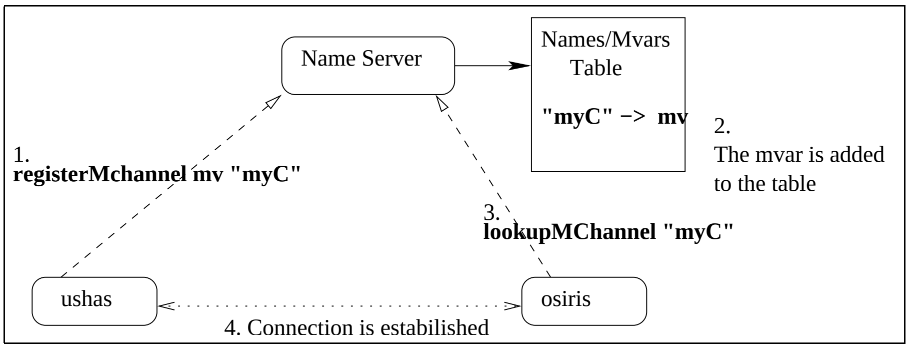

Name Server
Names/Mvars Table
"myC" -> mv
2. The mvar is added to the table
1. registerMchannel mv "myC"
3. lookupMChannel "myC"
ushas
osiris
4. Connection is estabilished

Fig. 7. GdH implementation of Mobile Channels

bersome in some cases but it serves to our purpose of simulating an open system using GdH.

Although we do not need a registration service for resources in GdH, because of the shared heap, we simulate it in the same way as the name server. In fact we only have one name server that accepts registration for both resources and channels but keep them in different tables.

The strict primitives can be partially simulated in our GdH prototype by using *evaluation strategies* [24]:

```
writeMChannel ch v = rnf v ‘seq‘ lazyWriteMChannel ch v
```

Here the strategy `rnf` evaluates its argument `v` to normal form before it is written into the channel `ch`. But this only works if the value being communicated was instantiated in the `NFData` class, which is a Haskell class that describes how to evaluate values.

7 Implementation Design

The GdH prototype allows us to test the concepts and ideas of the new communication primitives but is not suitable for the implementation of real applications. In this section we discuss the main issues that must be considered in a real implementation of the mobile system proposed.

Mobile systems have to abstract over the heterogeneity of large scale distributed systems, allowing machines of different architectures running different operating systems to communicate. This abstraction is usually achieved by compiling programs into architecture independent byte-code. As a platform to build our system, we have chosen the *Glasgow Haskell Compiler* (GHC) [10], a state of art implementation of Haskell. The main reason for choosing GHC is that it supports the execution of byte-code combined with machine code. GHC is both an optimising compiler and a interactive environment called GHCi. GHCi is designed for fast compilation and linking, it generates machine independent byte-code that is linked to the fast native-code available for the basic primitives of the language. Both GHC and GHCi share the same runtime-system, based on the STG-machine [13], that is a graph reduction machine.

In our implementation of the mobile system we are planning to implement routines for packing and unpacking Byte-Code Objects (BCOs), that are GHC's internal representation for its architecture independent byte-code, in the same way that other heap objects are communicated in GpH, GdH and Eden. By packing we mean *serialising* these objects into a form that is suitable for communication. As the basic modules in GHC are compiled into machine code and are present in every standard installation of the compiler, the packing routines have to pack only the machine independent part of the program and link it to the local definitions of the machine dependent part when the code is received and unpacked. This gives us the advantage of having much faster code than using only byte-code. Once packed, the BCO can be communicated using a simple TCP/IP socket connection. All machines running the mobile programs should have the same version of the GHC/GHCi system with an implementation of the primitives for mobility and also have the same binary libraries installed. Programs that communicate functions that are not in the standard libraries must be compiled into byte-code using GHCi.

Programs that will only receive byte-code do not need to have a GHCi installed because the byte-code interpreter is part of GHC's RTS. In fact, if only functions from the standard libraries are used in the mobile programs, there is no need of having GHCi at all in both ends of the communication.

In the future it may be possible to extend the compiler with a *mobility analyses* (maybe based on a non-determinism analyses [21]) that would decide the parts of the program that should be compiled into byte-code and the parts that could be compiled into machine code.

8 Related Work

There are other extensions to functional languages that allow the communication of higher-order values. Kali-Scheme [4] and Erlang [7] are examples of strict untyped languages (Erlang is dynamically typed) that allow the communication of functions. Haskell is a statically typed language hence the communication between nodes can be described as a data type and many mistakes can be caught during the compilation of programs. Other strict typed languages such as Nomadic Pict [29], Facile [18] and Jocaml [6] implement the communication primitives as side effects while we integrate them to the IO monad hence preserving referential transparency.

Curry [11] is a functional logic language that provides communication based on Ports in a similar way to the extension presented in this paper. Another language that is closely related to our system is Famke [27]. Famke is an implementation of threads for the lazy functional language Clean [20] (using monads and continuations), together with an extension for distributed communication using ports. The problem of their approach is that they do not have a real implementation of concurrency, their system provides interleaved execution of atomic actions implemented using continuations.

9 Conclusions and Future Work

In this paper we have presented a set of primitives for communication in a distributed extension of Haskell. Our primitives differ from previous approaches in that they allow communication of arbitrary expressions, including functions, in an open system. We have also demonstrated that our primitives, together with Haskell's higher-order functions, allow the implementation of powerful abstractions, e.g. remote evaluation, and useful patterns of communication can be packaged up as higher order functions or *mobile skeletons*. To validate our design and to test the programs presented in this paper, a prototype of our system was implemented using GdH.

We are currently working on a more robust implementation of the system. We will extend the GHC compiler and its byte-code interpreter GHCi with routines for packing and communicating heap objects as described in section 7. The novelty of the system lies in its ability to communicate the byte-code representation of heap objects generated by GHCi, thus allowing programs running on an heterogeneous network to communicate.

References

1. Silvia Breitinger and Rita Loogen and Yolanda Ortega-Mallén and Ricardo Peña. The Eden Coordination Model for Distributed Memory Systems. In *High-Level Parallel Programming Models and Supportive Environments*, volume 1123, IEEE Press, 1997.

2. Luca Cardelli. Abstractions for Mobile Computation. In *Secure Internet Programming*, pages 51-94, 1999.
3. Luca Cardelli. Mobility and Security. In *Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation*, pages 3-37, Marktoberdorf, Germany, Augus 1999.
4. Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(5):704-739, 1995.
5. Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.
6. Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile Agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.
7. Erlang. http://www.erlang.org/, WWW page, may 2003.
8. Ian Foster, C. Kesselman, and S.Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, 15(3), 2001.
9. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342-361, May 1998.
10. The Glasgow Haskell Compiler. http://www.haskell.org/ghc, WWW page, may, 2003.
11. Michael Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc.of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702, pages 376-395, Springer LNCS, 1999.
12. Frank Huch and Ulrich Norbisrath. Distributed programming in Haskell with Ports. In *Implementation of Functional Languages*, volume 2011, pages 107-121. Springer LNCS, 2000.
13. Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127-202, 1992.
14. Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47-96, IOS Press, 2001.
15. Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 1996.
16. Zeliha Dilsun Kirli. *Mobile Computation with Functions.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2001.
17. Ulrike Klusik, Yolanda Ortega-Mallen, and Ricardo Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages*, volume 1595, pages 103-119, Springer LNCS, 1999.
18. Frederick Colville Knabe. *Language Support for Mobile Agents.* PhD thesis, School of Computer Science, Carnegie mellon University, 1995.
19. Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents *Communications of the ACM*, 3(42):88-89, March 1999.
20. Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent Clean. In Leeuwen Aarts and Rem, editors, *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, volume 505, pages 202-219, Springer LNCS, 1991.

21. R. Peña and C. Segura. A Polynomial Cost Non-Determinism Analysis. In *Implementation of Functional Languages*, volume 2312, pages 121-137, Springer LNCS, 2002.
22. Marco Pil. Dynamic Types and Type Dependent Functions. In *Implementation of Functional Languages*, pages 169-185, Springer LNCS, 1998.
23. R. Pointon, P.W. Trinder, and H-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *Implementation of Functional Languages*, Springer LNCS, 2000.
24. Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23-60, January 1998.
25. Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable parallel implementation of Haskell, In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.
26. P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and Distributed Haskells, *Journal of Functional Programming*, 12(4):469-510, 2002.
27. Arjen van Weelden and Rinus Plasmeijer. Towards a Strongly Typed Functional Operating System. In *Implementation of Functional Languages*, volume 2670, Springer LNCS, 2003.
28. Dennis Volpano. Provably secure programming languages for remote evaluation. *ACM Computing Surveys*, 28(4es):176-176, 1996.
29. Pawel Tomasz Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, Wolfson College, University of Cambridge, 2000.

Refined Definitional Trees and Prolog Implementations of Narrowing

Pascual Julián Iranzo[1]

Departamento de Informática
Universidad de Castilla–La Mancha
Ciudad Real, Spain
Pascual.Julian@uclm.es

Abstract. This paper describes how high level implementations of (needed) narrowing into Prolog can be improved by introducing a refined representation of definitional trees that handles properly the knowledge about the inductive positions of a pattern. We define some generic algorithms that allow us to transform a functional logic program into a set of Prolog clauses which incorporates some refinements that are obtained by *ad hoc* artifices in other similar implementations of functional logic languages. We also present and discuss the advantages of our proposal by means of some simple examples.
Keywords: Functional logic programming, narrowing strategies, implementation of functional logic languages, program transformation.

1 Introduction

Functional logic programming [12] aims to implement programming languages that integrate the best features of both functional programming and logic programming. Most of the approaches to the integration of functional and logic languages consider term rewriting systems as programs and some narrowing strategy as complete operational mechanism. Laziness is a valuable feature of functional logic languages, since it increases the expressive power of this kind of languages: it supports computations with infinite data structures and a modular programming style. Among the different lazy narrowing strategies, needed narrowing [7] has been postulated optimal from several points of view: i) it is correct and complete, with regard to strict equations and constructor substitutions answers, for the class of inductively sequential programs (see, forward, Definition 2); ii) it computes minimal length derivations, if common variables are shared; and iii) no redundant answers are obtained. Some of these optimality properties have also been established for a broader class of term rewriting systems defining non–deterministic functions [4]. Needed narrowing addresses

[1] Supported by CICYT TIC 2001-2705-C03-01, Acción Integrada Hispano-Italiana HI2000-0161, Acción Integrada Hispano-Alemana HA2001-0059, and the Valencian Research Council under grant GV01-424.

computations by means of some structures, namely definitional trees [2], which contain all the information about the program rules. These structures allow us to select a position of the term which is being evaluated and this position points out to a reducible subterm that is "unavoidable" to reduce in order to obtain the result of the computation. It is accepted that the framework for declarative programing based on non–deterministic lazy functions of [17] also uses definitional trees as part of its computational mechanism. In recent years, a great effort has been done to provide the integrated languages with high level implementations of this computational model into Prolog (see for instance [3, 8, 13, 15] and [18]). Most of these implementation systems mainly rely on a two-phase transformation procedure that consists of: a suitable representation structure for the definitional trees associated with a functional logic program; and an algorithm that takes the above representation of definitional trees as an input parameter and translates it into a set of Prolog clauses able to simulate the narrowing strategy being implemented.

This paper investigates how a refined representation of definitional trees can introduce improvements in the quality of the code generated by the transformation scheme we have just described.

The paper is organized as follows: Section 2 recalls some basic notions we use in the rest of the sections. In Section 3 we describe a refined representation of definitional trees and we give an algorithm for building them in the style of [14]. Section 4 discusses how to translate functional logic programs into Prolog, taking advantage of the new representation of definitional trees to improve (needed) narrowing implementations. Section 5 presents some experiments that show the effectiveness of our proposal. Section 6 contains our conclusions. Finally we briefly discuss the lines of future work.

2 Preliminaries

We consider first order expressions or *terms* built from symbols of the set of variables $\mathcal{X}$ and the set of function symbols $\mathcal{F}$ in the usual way. The set of terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We sometimes write $f/n \in \mathcal{F}$ to denote that f is a n–ary function symbol. If t is a term different from a variable, $\mathcal{R}oot(t)$ is the function symbol heading t, also called the *root symbol* of t. A term is *linear* if it does not contain multiple occurrences of the same variable. $\mathcal{V}ar(o)$ is the set of variables occurring in the syntactic object o. We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

A *substitution* σ is a mapping from the set of variables to the set of terms, with finite *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$. We denote the identity substitution by *id*. We define the composition of two substitutions σ and θ, denoted $\sigma \circ \theta$ as usual: $\sigma \circ \theta(x) = \hat{\sigma}(\theta(x))$, where $\hat{\sigma}$ is the extension of σ to the domain of the terms. A *renaming* is a substitution ρ such that there exists the inverse substitution ρ^{-1} and $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $(\exists\sigma)\ s = \sigma(t)$. Two terms t and t' are *variants* if there exists a renaming ρ

such that $t' = \rho(t)$. We say that t is *strictly* more general than s, denoted $t < s$, if $t \leq s$ and t and s are not variants. The quasi–order relation "$\leq$" on terms is often called *subsumption order* and "$<$" is called *strict subsumption order*.

Positions of a term t (also called *occurrences*) are represented by sequences of natural numbers used to address subterms of t. The concatenation of the sequences p and w is denoted by $p.w$. Two positions p and p' of t are *comparable* if $(\exists w)\ p' = p.w$ or $p = p'.w$, otherwise are *disjoint* positions. Given a position p of t, $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the result of replacing the subterm $t|_p$ by the term s. Let $\overline{p_n}$ be a sequence of disjoint positions of a term t, $t[s_1]_{p_1} \ldots [s_n]_{p_n}$ denotes the result of simultaneously replacing each subterm $t|_{p_i}$ by the term s_i, with $i \in \{1, \ldots, n\}$.

2.1 Term rewriting systems

We limit the discussion to unconditional term rewriting systems[1]. A *rewrite rule* is a pair $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The terms l and r are called the *left–hand side* (lhs) and *right–hand side* (rhs) of the rewrite rule, respectively. A *term rewriting system* (TRS) $\mathcal{R}$ is a finite set of rewrite rules.

We are specially interested in TRSs whose associate signature $\mathcal{F}$ can be partitioned into two disjoint sets $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ where $\mathcal{D} = \{\mathcal{R}oot(l) \mid (l \to r) \in \mathcal{R}\}$ and $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions* or *operations*. Terms built from symbols of the set of variables $\mathcal{X}$ and the set of constructors $\mathcal{C}$ are called *constructor terms*. A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{D}$ and $\overline{d_n}$ are constructor terms. A term $f(\overline{x_n})$, where $\overline{x_n}$ are different variables, is called generic pattern. A TRS is said to be *constructor–based* (CB) if the lhs of its rules are patterns. For CB TRSs, a term t is a *head normal form* (hnf) if t is a variable or $\mathcal{R}oot(t) \in \mathcal{C}$.

A TRS is said to be *left–linear* if for each rule $l \to r$ in the TRS, the lhs l is a linear term. We say that a TRS is *non–ambiguous* or *non–overlapping* if it does not contain critical pairs (see [10] for a standard definition of critical pair). Left–linear and non–ambiguous TRSs are called *orthogonal* TRSs.

Inductively sequential TRSs are a proper subclass of CB orthogonal TRSs. The definition of this class of programs make use of the notion of *definitional tree*. For the sake of simplicity and because further complications are irrelevant for our study, in the following definition, we ignore the *exempt* nodes that appear in the original definition of [2] and also the *or*–nodes of [15] used in the implementation of Curry [14]. Note also, that or–nodes lead to parallel definitional trees and thus out of the class of inductively sequential systems.

Definition 1. [Partial definitional tree]
Given a CB TRS $\mathcal{R}$, $\mathcal{P}$ is a partial definitional tree *with pattern π if and only if one of the following cases hold:*

1. *$\mathcal{P} = rule(\pi, l \to r)$, where π is a pattern and $l \to r$ is a rewrite rule in $\mathcal{R}$ such that π is a variant of l.*

[1] This is not a true limitation for the expressiveness of a programming language relaying on this class of term rewriting systems [5].

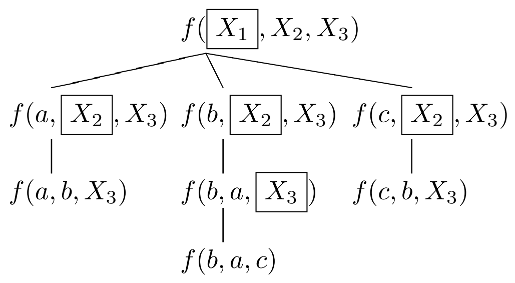


Fig. 1. Definitional tree for the function "f"of Example 1

2. *$\mathcal{P} = branch(\pi, o, \overline{\mathcal{P}_k})$, where π is a pattern, o is a variable position of π (called* inductive position*), $\overline{c_k}$ are different constructors, for some $k > 0$, and for all $i \in \{1, \ldots, k\}$, $\mathcal{P}_i$ is a partial definitional tree with pattern $\pi[c_i(\overline{x_n})]_o$, where n is the arity of c_i and $\overline{x_n}$ are new variables.*

From a declarative point of view, a partial definitional tree $\mathcal{P}$ can be seen as a set of linear patterns partially ordered by the strict subsumption order "$<$" [4]. Given a defined function f/n, a *definitional tree of* f is a partial definitional tree whose pattern is a generic pattern and its leaves contain variants of all the rewrite rules defining f.

Example 1. Given the rules defining the function $f/3$

$$R_1 : f(a, b, X) \to r_1, \qquad R_2 : f(b, a, c) \to r_2, \qquad R_3 : f(c, b, X) \to r_3.$$

a definitional tree of f is:

$$\begin{aligned} &branch(f(X_1, X_2, X_3), 1,\\ &\qquad branch(f(a, X_2, X_3), 2, rule(f(a, b, X_3), R_1)),\\ &\qquad branch(f(b, X_2, X_3), 2, branch(f(b, a, X_3), 2, rule(f(b, a, c), R_2))),\\ &\qquad branch(f(c, X_2, X_3), 2, rule(f(c, b, X_3), R_3))) \end{aligned}$$

Note that there can be more than one definitional tree for a defined function. It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where each node is marked with a pattern and the inductive position in branches is surrounded by a box. Figure 1 illustrates this concept.

Definition 2. [Inductively Sequential TRS]
A defined function f is called inductively sequential *if it has a definitional tree. A rewrite system $\mathcal{R}$ is called inductively sequential if all its defined functions are inductively sequential.*

In this paper we are mainly interested in inductively sequential TRSs (or proper subclasses of them) which are called *programs*.

2.2 Definitional trees and Narrowing Implementations into Prolog

Most of the relevant implementations of functional logic languages, which use needed narrowing as operational mechanism, are based on the compilation of the programs written in these languages into Prolog [8, 13, 15, 16]. These implementation systems may be thought as a translation process that essentially consists in the following:

1. An algorithm to transform the program rules in a functional logic program into a set of definitional trees (See [15] and [14] for some of those algorithms).
2. An algorithm that takes the definitional trees as an input parameter and visits their nodes, generating a Prolog clause for each visited node. Since definitional trees contain all the information about the original program as well as information to guide the (optimal) pattern matching process during the evaluation of expressions, the set of generated Prolog clauses is able to simulate the intended narrowing strategy being implemented.

In the case of functional logic programs with a needed narrowing semantics, a generic algorithm for the translation of definitional trees into a set of clauses is given in [13]. When we apply that algorithm to the definitional tree of function f in Example 1, we obtain the following set of Prolog clauses:

```
% Clause for the root node: it exploits the first inductive position
f(X1, X2, X3, H) :- hnf(X1, HX1), f_1(HX1, X2, X3, H).

% Clauses for the remainder nodes:
f_1(a, X2, X3, H):- hnf(X2, HX2), f_1_a_2(HX2, X3, H).
f_1_a_2(b, X3, H):- hnf(r1, H).

f_1(b, X2, X3, H):- hnf(X2, HX2), f_1_b_2(HX2, X3, H).
f_1_b_2(a, X3, H):- hnf(X3, HX3), f_1_b_2_a_3(HX3, H).
f_1_b_2_a_3(c, H):- hnf(r2, H).

f_1(c, X2, X3, H):- hnf(X2, HX2), f_1_c_2(HX2, X3, H).
f_1_c_2(b, X3, H):- hnf(r3, H).
```

where `hnf(T, H)` is a predicate that is true when `H` is the hnf of a term `T`. For this example, the clauses defining the predicate `hnf` are:

```
% Evaluation to head normal form (hnf).
hnf(T, T) :- var(T), !.
hnf(f(X1, X2, X3), H) :- !, f(X1, X2, X3, H).
hnf(T, T). % otherwise the term T is a hnf;
```

The meaning of these set of clauses is very easy to understand. For evaluating a term $t = f(t_1, t_2, t_3)$ to a hnf, first, it is necessary to evaluate (to a hnf) the substerms of t at the inductive positions of the patterns in the definitional tree associated with f (in the order dictated by that definitional tree — see Figure 1). Hence, for our example: we compute the hnf of t_1 and then the hnf of t_2; if b is the hnf of t_1 and a is the hnf of t_2, we have to compute the hnf of t_3; if the hnf of t_3 is c then the hnf of t will be the hnf of r_2 else the computation fails (see the sixth clause). On the other hand, if the hnf of t_1 is a or c it suffices to evaluate t_2 to a hnf, disregarding t_3, in order to obtain the final value. This evaluation mechanism conforms with the needed narrowing strategy of [7], as it has been formally demonstrated in [1].

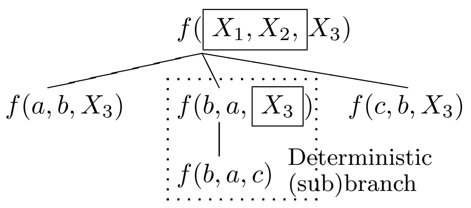


Fig. 2. Refined definitional tree for the function "f"of Example 1

3 A Refined Representation of Definitional Trees

As we have just seen, building definitional trees is the first step of the compilation process in high level implementations of needed narrowing into Prolog. Therefore, providing a suitable representation structure for the definitional trees associated with a functional logic program may be an important task in order to improve those systems. In this section we give a refined representation of definitional trees that saves memory allocation and is the basis for further improvements.

It is noteworthy that the function f of Example 1 has two definitional trees: the one depicted in Figure 1 and a second one obtained by exploiting position 2 of the generic pattern $f(X_1, X_2, X_3)$. Hence, this generic pattern has two inductive positions. We can take advantage of this situation if we "simultaneously" exploit these two positions to obtain the definitional tree depicted in the Figure 2. This new representation cuts the number of nodes of the definitional tree from eight to five nodes. Note also that this kind of representation reduces the number of possible definitional trees associated to a function. Actually, using the new representation, there is only one definitional tree for f.

The main idea of the refinement is as follows: when a pattern has several inductive positions, exploit them altogether. Therefore we need a criterion to detect inductive positions. This criterion exists and it is based on the concept of uniformly demanded position of [15].

Definition 3. [Uniformly demanded position]
Given a pattern π and a TRS $\mathcal{R}$, Let be $\mathcal{R}_\pi = \{l \to r | (l \to r) \in \mathcal{R} \wedge \pi \leq l\}$. A variable position p of the pattern π is said to be: (i) demanded *by a lhs l of a rule in $\mathcal{R}_\pi$ if $\mathcal{R}oot(l|_p) \in \mathcal{C}$. (ii)* uniformly demanded *by $\mathcal{R}_\pi$ if p is demanded for all lhs in $\mathcal{R}_\pi$.*

We write $\mathcal{UDP}os(\pi)$ to denote the set of uniformly demanded positions of the pattern π. The following proposition establishes a necessary condition for a position of a pattern to be an inductive position.

Proposition 1. *Let $\mathcal{R}$ be an inductively sequential TRS and let π be the pattern of a branch node of a definitional tree $\mathcal{P}$ of a function defined in $\mathcal{R}$. If o is an inductive position of π then o is uniformly demanded by $\mathcal{R}_\pi$.*

Proof. We proceed by contradiction. Assume o is not uniformly demanded by $\mathcal{R}_\pi$. Hence, there must exist some $(l \to r) \in \mathcal{R}_\pi$ such that $\mathcal{R}oot(l|_o) = c \in \mathcal{C}$, and some $(l' \to r') \in \mathcal{R}_\pi$ such that $l'|_o \in \mathcal{X}$. Since o is the inductive position of the branch node whose pattern is π, by definition of definitional tree, $\pi < \pi[c(\overline{x_n})]_o \leq l$ and $\pi \leq l'$. Therefore it is impossible to built a partial definitional tree with leaves l and l' by exploiting the position o, which contradicts the hypothesis that o is an inductive position.

Hence, the concept of uniformly demanded position and Proposition 1 give us a syntactic criterion to detect if a variable position of a pattern is an inductive position or not and, therefore, a guideline to built a definitional tree: (i) Given a branch node, select a uniformly demanded position of its pattern; fix it as an inductive position of the branch node and generate the corresponding child nodes. (ii) If the node doesn't have uniformly demanded positions then there two possibilities: the node is a leaf node, if it is a variant of a lhs of the considered TRS, or it is a "failure" node, and it is impossible to build the definitional tree. The following algorithm, in the style of [14], uses this scheme to build a refined partial definitional tree $rpdt(\pi, \mathcal{R}_\pi)$ for a pattern π and rules $\mathcal{R}_\pi = \{l \to r \mid (l \to r) \in \mathcal{R} \wedge \pi \leq l\}$:

1. If $\mathcal{UDP}os(\pi) = \emptyset$ and there is only one rule $(l \to r) \in \mathcal{R}_\pi$ and a renaming ρ such that $\pi = \rho(l)$:

   $$rpdt(\pi, \mathcal{R}_\pi) = rule(\pi, \rho(l) \to \rho(r));$$

2. If $\mathcal{UDP}os(\pi) \neq \emptyset$ and for all $(c_{i_1}, \ldots, c_{i_m}) \in \mathcal{C}_\pi$, $\mathcal{P}_i = rpdt(\pi_i, \mathcal{R}_{\pi_i}) \neq \texttt{fail}$:

   $$rpdt(\pi, \mathcal{R}_\pi) = branch(\pi, \overline{o_m}, \overline{\mathcal{P}_k});$$

   where $\overline{o_m}$ is the sequence of uniformly demanded positions in $\mathcal{UDP}os(\pi)$, $\mathcal{C}_\pi = \{(c_{i_1}, \ldots, c_{i_m}) | (l_i \to r_i) \in \mathcal{R}_\pi \wedge \mathcal{R}oot(l_i|_{o_1}) = c_{i_1} \wedge \ldots \wedge \mathcal{R}oot(l_i|_{o_m}) = c_{i_m}\}$, $k = |\mathcal{C}_\pi| > 0$, $\pi_i = \pi[c_{i_1}(\overline{x_{n_{i_1}}})]_{o_1} \ldots [c_{i_m}(\overline{x_{n_{i_m}}})]_{o_m}$ and $\overline{x_{n_{i_1}}}, \ldots, \overline{x_{n_{i_m}}}$ are new variables.
3. Otherwise, $rpdt(\pi, \mathcal{R}_\pi) = \texttt{fail}$.

Given an inductively sequential TRS $\mathcal{R}$ and a n–ary defined function f in $\mathcal{R}$, the definitional tree of f is $rdt(f, \mathcal{R}) = rpdt(\pi_0, \mathcal{R}_{\pi_0})$ where $\pi_0 = f(\overline{x_n})$. Note that, for an algorithm like the one described in [14] the selection of the inductive positions of the pattern π is non–deterministic, if $\mathcal{UDP}os(\pi) \neq \emptyset$. Therefore, it is possible to build different definitional trees for an inductively sequential function, depending on the inductive position which is selected. On the contrary, our algorithm deterministically produces a single definitional tree for each inductively sequential function. Note also that it matches the more informal algorithm that appears in [14] when, for each branch node, there is only one inductive position.

We illustrate the previous algorithm and last remarks by means of a new example.

Example 2. Given the rules defining the function $f/2$

$$R_1 : f(0,0) \to 0, R_2 : f(s(X),0) \to s(0), R_3 : f(s(X),s(s(Y))) \to f(X,Y).$$

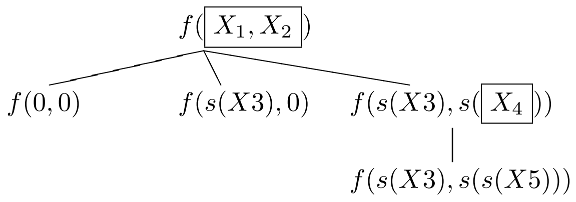


Fig. 3. Refined definitional tree for the function "f"of Example 2

the last algorithm builds the following definitional tree for f:

$$\begin{array}{l} branch(f(X_1,X_2),(1,2),\\ \quad rule(f(0,0),R_1),\\ \quad rule(f(s(X_3),0),R_2),\\ \quad branch(f(s(X_3),s(X_4)),(2.1),rule(f(s(X_3),s(s(X_5))),R_3)) \end{array}$$

which is depicted in Figure 3. The algorithm for generating definitional trees of [14] may build two definitional trees for f (depending on whether position 1 or position 2 is selected as the inductive position of the generic pattern $f(X_1, X_2)$). Both of these trees have seven nodes, while the new representation of Figure 3 reduces the number of nodes of the definitional tree to five nodes.

As it has been proposed in [8], it is possible to obtain a simpler translation scheme of functional logic programs into Prolog if definitional trees are first compiled into *case expressions*. That is, functions are defined by only one rule where the lhs is a generic pattern and the rhs contains case expresions to specify the pattern matching of actual arguments. The use of case expressions doesn't invalidate our argumentation. Thus, we can transform the definitional tree of Example 2 in the following case expression:

$$\begin{array}{lll} f(X_1,X_2) = & \texttt{case}\ (X_1,X_2)\ \texttt{of} & \\ & (0,0) & \to 0\\ & (s(X_3),0) & \to s(0)\\ & (s(X_3),s(X_4)) & \to \texttt{case}\ (X_4)\ \texttt{of}\\ & & \quad s(X_5) \to f(X_3,X_5) \end{array}$$

A case expression, like this, will be evaluated by reducing a tuple of arguments to their hnf and matching them with one of the patterns of the case expression.

4 Improving Narrowing Implementations into Prolog

The refined representation of definitional trees introduced in Section 3 is very close to the standard representation of definitional trees, but it is enough to provide further improvements in the translation of functional logic programs into Prolog.

It is easy to adapt the translation algorithm that appears in [13] to use our refined representation of definitional trees as input. If we apply this slightly different algorithm to the refined definitional tree of Figure 2, we obtain the following set of clauses:

```
% Clause for the root node:
f(X1, X2, X3, H) :- hnf(X1, HX1), hnf(X2, HX2), f_1_2(HX1, HX2, X3, H).

% Clauses for the remainder nodes:
f_1_2(a, b, X3, H):- hnf(r1, H).

f_1_2(b, a, X3, H):- hnf(X3, HX3), f_1_2_b_a(HX3, H).

f_1_2_b_a(c, H):- hnf(r2, H).

f_1_2(c, b, X3, H):- hnf(r3, H).
```

where we have cut the number of clauses with regard to the standard representation into Prolog (of the rules defining function f) presented in Section 2.2. The number of clauses is reduced in the same proportion the number of nodes of the standard definitional tree for f were cut. As we are going to show in the next section, this refined translation technique is able to improve the efficiency of the implementation system.

Note that the analysis of definitional trees provide further opportunities for improving the translation of inductively sequential programs into Prolog. For instance, we can take notice that the definitional tree of function f in Example 1 has a "deterministic" (sub)branch, that is, a (sub)branch whose nodes have only one child (see Figure 2). This knowledge can be used as an heuristic guide for applying unfolding transformation steps selectively. Hence, for the example we are considering, the clauses:

```
f_1_2(b, a, X3, H):- hnf(X3, HX3), f_1_2_b_a(HX3, H).
f_1_2_b_a(c, H):- hnf(r2, H).
```

can be transformed in:

```
f_1_2(b, a, X3, H):- hnf(X3, c), hnf(r2, H).
```

We think this selective unfolding is preferable to the more costly and generalized (post–compilation) unfolding transformation process suggested in [13] and [8].

On the other hand, it is important to note that the kind of improvements we are mainly studying in this work can not be obtained by an unfolding transformation process applied to the set of clauses produced by the standard algorithm of [13]: In fact, it is not possible to obtain the above set of clauses by unfolding transformation of the set of clauses shown in Section 2.2.

5 Experiments

We have made some experiments to verify the effectiveness of our proposal. We have instrumented the Prolog code obtained by the compilation of simple Curry programs by using the `curry2prolog` compiler of Pakcs [9] (an implementation of the multi–paradigm declarative language Curry [14]). We have introduced our translation technique in the remainder Prolog code. The results of the experiments are shown in Table 1. Runtime and memory occupation were measured on a Sun4 Sparc machine, running sicstus v3.8 under SunOS v5.7. The "`Speedup`" column indicates the percentage of execution time saved by our translation technique. The values shown on that column are the percentage of the quantity computed by the formula $(t_1 - t_2)/t_1$, where t_1 and t_2 are the average runtimes, for several executions, of the proposed terms (goals) and Prolog programs obtained when we don't use (t_1) and we use (t_2) our translation technique. The "`G. stack Imp.`" column reports the improvement of memory occupation for the computation. We have measured the percentage of global stack allocation. The amount of memory allocation measured between in each execution remains constant. Most of the benchmark programs are extracted from [14] and the standard prelude for Curry programs with slight modifications. For the benchmark programs `family` and `xor` we evaluate all outcomes. The natural numbers are implemented in Peano notation, using `zero` and `succ` as constructors of the sort. In the `zip` and `zip3` programs the input terms $L1$ and $L2$ are lists of length 9.

Table 1. Runtime speed up and memory usage improvements for some benchmark programs and terms.

`Benchmark`	`Term`	`Speedup`	`G. stack Imp.`
`family`	$grandfather(\_,\_)$	19.9%	0%
`geq`	$geq(100000, 99999)$	4.6%	16.2%
`geq`	$geq(99999, 100000)$	4.3%	16.2%
`xor`	$xor(\_,\_)$	18.5%	0%
`zip`	$zip(L1, L2)$	3.6%	5.5%
`zip3`	$zip3(L1, L2, L2)$	4.5%	10%
	`Average`	9.2%	7.9%

More detailed information about the experiments and benchmark programs can be found in `http://www.inf-cr.uclm.es/www/pjulian/publications.html`.

6 Discussion and Conclusions

Although the results of the preceding section reveals a good behavior of our translation technique, it is difficult to evaluate what may be its impact over the whole system, since the improvements appear when we can detect patterns which have several uniformly demanded positions. For the case of inductively sequential functions without this feature, our translation scheme is conservative and doesn't produce runtime speedups or memory allocation improvements. On the other hand it is noteworthy that, in some cases, the benefits of our translation scheme are obtained in an *ad hoc* way in actual needed narrowing into Prolog implementation systems. For instance, the standard definition of the strict equality used in non–strict functional logic languages is [11, 18]:

$$\begin{aligned} c == c &\to true \\ c(\overline{X_n}) == c(\overline{Y_n}) &\to X_1 == Y_1 \&\& \ldots \&\& X_n == Y_n \end{aligned}$$

where c is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule. There is one of these rules for each constructor that appears in the program we are considering. Clearly, the strict equality has an associate definitional tree whose pattern ($X_1 == X_2$) has two uniformly demanded positions (positions 1 and 2) and, therefore, it can be translated using our technique, that produces a set of Prolog clauses similar to the one obtained by the `curry2prolog` compiler. In fact, the `curry2prolog` compiler translates these rules into the following set of Prolog clauses[2]:

```
hnf(A==B,H):-!,seq(A,B,H).

seq(A,B,H):-hnf(A,F),hnf(B,G),seq_hnf(F,G,H).

seq_hnf(true,true,H):-!,hnf(true,H).
seq_hnf(false,false,H):-!,hnf(true,H).
seq_hnf(c,c,H):-!,hnf(true,H).
seq_hnf(c(A1,...,Z1),c(A2,...,Z2),H):-!,
               hnf(&&(A1==A1,&&(B1==B2,&&(...,&&(Z1==Z2,true)))),H).
```

Thus, the `curry2prolog` compiler produces an optimal representation of the strict equality which is treated as a special system function with an *ad hoc* predefined translation into Prolog, instead of using the standard translation algorithm which is applied for the translation of user defined functions.

Although our contribution, as well as the overall theory of needed evaluation, is interesting for computations that succeed, it is important to say that some problems may arise when a computation does not terminate or fails. For example, given the (partial) function

$$f(a, a) \to a$$

the standard compilation into Prolog is:

```
f(A,B,C)   :- hnf(A,F), f_1(F,B,C).
f_1(a,A,B) :- hnf(A,E), f_1_a_2(E,B).
f_1_a_2(a,a).
```

whilst our translation technique produces:

```
f(A,B,C) :- hnf(A,F), hnf(B,G), f_1(F,G,C).
f_1(a,a,a).
```

Now, if we want to compute the term `f(b, expensive_term)`, the standard implementation detects the failure after the computation of the first argument. On the other hand, the new implementation computes the expensive term (to

[2] Note that, we have simplified the code produced by the `curry2prolog` compiler in order to increase its readability and facilitate the comparison with our proposal.

hnf) for nothing. Of course, the standard implementation has problems too —e.g. if we compute the term `f(expensive_term, b)`, it also computes the expensive term (to hnf)—, but it may have a better behavior on this problem. Thus, in a sequential implementation, the performance of our translation technique may be endanger when subterms, at uniformly demanded positions, are evaluated (to hnf) jointly with an other subterm whose evaluation (to hnf) produces a failure. An alternative to overcome this practical disadvantage is to evaluate these subterms in parallel, introducing monitoring techniques able to detect the failure as soon as possible and then to stop the streams of the computation.

Nevertheless, our work shows that there is a potential for the improvement of actual (needed) narrowing implementation systems: we obtain valuable improvements of execution time and memory allocation when our translation technique is relevant, without an appreciable slowdown in the cases where it is not applicable. Also, our simple translation technique is able to eliminate some *ad hoc* artifices in actual implementations of (needed) narrowing into Prolog, providing a systematic and efficient translation mechanism. Moreover, the ideas we have just developed can be introduced with a modest programming effort in standard implementations of needed narrowing into Prolog (such as the PAKCS [9] implementation of Curry) and in other implementations based on the use of definitional trees (e.g., the implementation of the functional logic language $\mathcal{TOY}$[16]), since they don't modify their basic structures.

7 Future Work

Failing derivations are rather a problematic case where the performance of our translation technique may be endanger. We want to deal with these problem in order to guarantee that slowdowns, with regard to standard implementations of needed narrowing into Prolog, are not produced. Also we like to study how *clause indexing* [19], in the context of Prolog implementation, relates with our work.

On the other hand, we aim to investigate how definitional trees may be used as a guide to introduce selective program transformation techniques.

Acknowledgements

I gratefully acknowledge the anonymous referees for many useful suggestions and Sergio Antoy for clarifying me some aspects of the theoretical/practical dimension of this problem.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Uniform Lazy Narrowing. *Journal of Logic and Computation* 13(2), 2003. Short preliminary version in Proc. of WFLP 2002, available at `http://www.inf-cr.uclm.es/www /pjulian`

2. S. Antoy. Definitional trees. In *Proc. of ALP'92*, volume 632 of *LNCS*, pages 143–157. Springer-Verlag, 1992.
3. S. Antoy. Needed Narrowing in Prolog. Technical Report 96-2, Portland State University, 1996. Full version of extended abstract in [6].
4. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of ALP'97*, volume 1298 of *LNCS*, pages 16–30. Springer-Verlag, 1997.
5. S. Antoy. Constructor-based conditional narrowing. In *Proc. of (PPDP'01).* Springer LNCS, 2001.
6. S. Antoy. Needed Narrowing in Prolog. In *Proc. of PLILP'96*, *LNCS*, pages 473–474. 1996.
7. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
8. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. FroCoS 2000*, pages 171–185. Springer LNCS 1794, 2000.
9. S. Antoy, M. Hanus, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pacs 1.3 : The Portland Aachen Kiel Curry System User Manual. Technical Report Version of December, 4, University of Kiel, Germany, 2000. Available from URL: `http://www.informatik. uni-kiel.de/~pakcs/`
10. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
11. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.
12. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. LOPSTR'95*, pages 252–266. Springer LNCS 1048, 1995.
14. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www. informatik.uni-kiel.de/~curry`, 1999.
15. R. Loogen, F. López-Fraguas, and M. Rodríguez - Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.
16. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
17. J. G. Moreno, M. H. González, F. López-Fraguas, and M. R. Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *Journal of Logic Programming*, 1(40):47–87, 1999.
18. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
19. R. Ramesh and I. V. Ramakrishnan and D. S. Warren. Automata–Driven Indexing of Prolog Clauses. *Journal of Logic Programming*, 23(2):151–202. 1995.

A Narrowing-based Instantiation Rule for Rewriting-based Fold/Unfold Transformations [⋆]

Ginés Moreno

Department of Computer Science, University of Castilla-La Mancha,
02071 Albacete, Spain. gmoreno@info-ab.uclm.es

Abstract. In this paper we show how to transfer some developments done in the field of functional–logic programming (FLP) to a pure functional setting (FP). More exactly, we propose a complete fold/unfold based transformation system for optimizing lazy functional programs. Our main contribution is the definition of a safe instantiation rule which is used to enable effective unfolding steps based on rewriting. Since instantiation has been traditionally considered problematic in FP, we take advantage of previous experiences in the more general setting of FLP where instantiation is naturally embedded into an unfolding rule based on narrowing. Inspired in the so called needed narrowing strategy, our instantiation rule inherits the best properties of this refinement of narrowing. Our proposal optimises previous approaches defined in the literature (that require more transformation effort) by anticipating bindings on unifiers used to instantiate a given program rule and by generating redexes at different positions on instantiated rules in order to enable subsequent unfolding steps. As a consequence, our correct/complete technique avoids redundant rules and preserve the natural structure of programs.

Keywords: Rewriting, Narrowing, Instantiation, Fold/Unfold

1 Introduction

In this paper we are concerned with first order (lazy) functional programs expressed as term rewriting systems (TRS's for short). Our programming language is similar to the one presented in [7, 17] in which programs are written as a set of mutually exclusive recursive equations, but we do not require that the set of equations defining a given function f be exhaustive (i.e., we let f to be undefined for certain elements of its domain). More precisely , we use inductively sequential TRS's, i.e., the mainstream class of pattern-matching functional programs which are based on a case distinction. This is a reasonable class of TRS's not only for modeling pure functional programs but also for representing multiparadigm functional–logic programs that combine the operational methods and advantages of both FP and LP (logic programming) [3]. If the operational principle of FP is based on rewriting, integrated programs are usually executed by

[⋆] This work has been partially supported by CICYT under grant TIC 2001-2705-C03-03, and by the Valencian Research Council under grant GV01-424.

narrowing, a generalization of rewriting that instantiates variables in a given expression and then, it applies a reduction step to a redex of the instantiated expression. Needed narrowing extends the Huet and Lévy's notion of a needed reduction [12], and is the currently best narrowing strategy for first-order (inductively sequential) integrated programs due to its optimality properties w.r.t. the length of derivations and the number of computed solutions and it can be efficiently implemented by pattern matching and unification [4].

The fold/unfold transformation approach was first introduced in [7] to optimize functional programs (by reformulating function definitions) and then used in LP [21] and FLP [2, 16]. This approach is commonly based on the construction, by means of a *strategy*, of a sequence of equivalent programs each obtained from the preceding ones by using an *elementary* transformation rule. The essential rules are *folding* and *unfolding*, i.e., contraction and expansion of subexpressions of a program using the definitions of this program (or of a preceding one). Other rules which have been considered are, for example: instantiation, definition introduction/elimination, and abstraction.

Instantiation is a purely functional transformation rule used for introducing an instance of an existing equation in order to enable subsequent unfolding steps based on rewriting. For instance, if we want to apply an unfolding step on rule $R_1 : \mathtt{f(s(X))} \to \mathtt{s(f(X))}$ by reducing its right hand side (rhs) by using rule R_1 itself, we firstly need to "instantiate" the whole rule with the binding $\{\mathtt{X} \mapsto \mathtt{s(Y)}\}$ obtaining the new (instantiated) rule $R_2 : \mathtt{f(s(s(Y)))} \to \mathtt{s(}\underline{\mathtt{f(s(Y))}}\mathtt{)}$. Now, we can perform a rewriting step[1] on the underlined subterm in the rhs of R_2 using R_1, and then we obtain the desired (unfolded) rule $R_3 : \mathtt{f(s(s(Y)))} \to \mathtt{s(s(f(Y)))}$. In contrast with this naive example, we will see in Section 3 that, in general, it is not easy to decide neither the binding to be applied nor the subterm to be converted in a reducible expression in the considered rule.

The instantiation rule is avoided in LP and FLP transformation systems: the use of SLD-resolution or narrowing (respectively) empowers the fold/unfold system by implicitly embedding the instantiation rule into unfolding by means of unification. Moreover, instantiation is not treated explicitly even in some pure functional approaches, as is the case of [20] where the role of instantiation is assumed to be played by certain distribution laws for case expressions (which are even more problematic than instantiation since they are defined in a less "constructive" way). Nevertheless, the need for a transformation that generates redexes on rules is crucial in a functional setting if we really want to apply unfolding steps based on rewriting.

Similarly to the instantiation process generated implicitly by most unfolding rules for LP ([21, 17]) and FLP ([16]), classical instantiation rules for pure FP ([7, 8, 17]) consider a unique subterm t in a rule R to be converted in a redex and then, R is instantiated with the most general unifiers obtained by unifying t with every rule in the program. Anyway (and as said in [20]), unrestricted instantiation is problematic because it is not even locally equivalence preserving, since it can force premature evaluation of expressions (a problem noted in [6],

[1] Note that this reduction step is unfeasible on the original rule R_1.

and addressed in some detail in [19]) and is better suited to a typed language in which one can ensure that the set of instances is exhaustive. Moreover, the use of mgu's in (the also called "minimal") instantiation rules[2] instead of more precise unifiers may produce sets of non mutually exclusive (i.e., overlapping) rules, which leads to corrupt programs.

All these facts strongly contrast with the transformation methodology for lazy (*call-by-name*, *call-by-need*) FLP based on needed narrowing presented in [2]. The use of needed narrowing inside an unfolding step is able to reduce redexes at different positions in a rule once it as been instantiated by using unifiers which are more precise than the standard most general ones. Moreover, the transformation preserves the original structure (inductive sequentiality) of programs and offers strong correctness results (i.e., it preserves the semantics not only of values, but also of computed answer substitutions). Inspired in all these nice results, we show in this paper how to extrapolate the rules presented in [2] to a pure FP setting and, in particular, we built a safe instantiation rule that transcends the limitations of previous approaches.

The structure of the paper is as follows. After recalling some basic definitions in the next section, we introduce our needed narrowing based instantiation rule in Section 3. Section 4 defines an unfolding rule that reinforces the effects of the previous transformation. By adding new rules for folding, abstracting and introducing new definitions, we obtain a complete transformation system for FP in Section 5. Section 6 illustrates our approach with real examples and finally, Section 7 concludes. More details can be found in [14].

2 Functional Programs *versus* Functional-Logic Programs

For the purposes of this paper, functional and functional–logic programs are undistinguished by syntactic aspects since both can be seen as (inductively sequential) TRS's and they only differ from its corresponding operational semantics which are based on rewriting and narrowing, respectively. In this section we briefly recall some preliminary concepts and notation subjects. We assume familiarity with basic notions from TRS's, FP and FLP [5, 10]. In this work we consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of *defined* functions. The set of *constructor terms* with *variables* is obtained by using symbols from $\mathcal{C}$ and $\mathcal{X}$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. We write $\overline{o_n}$ for the *list* of objects $o_1, \ldots, o_n$. A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n$ are constructor terms (note the difference with the usual notion of pattern in functional programming: a constructor term). A term is *linear* if it does not contain multiple occurrences of one variable. A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with

[2] Specially when trying to generate redexes at different positions on instantiated rules.

$\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. *id* denotes the identity substitution. We write $t \leq t'$ (*subsumption*) iff $t' = \sigma(t)$ for some substitution σ.

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (lhs) and the *right-hand side* (rhs) of the rule, respectively. In the remainder of this paper, functional programs are a subclass of TRS's called inductively sequential TRS's. To provide a precise definition of this class of programs we introduce definitional trees [3]. A *definitional tree* of a finite set of linear patterns S is a non-empty set $\mathcal{P}$ of linear patterns partially ordered by subsumption having the following properties:

Root property: There is a minimum element $pattern(\mathcal{P})$, also called the *pattern* of the definitional tree.

Leaves property: The maximal elements, called the *leaves*, are the elements of S. Non-maximal elements are also called *branches*.

Parent property: If $\pi \in \mathcal{P}$, $\pi \neq pattern(\mathcal{P})$, there exists a unique $\pi' \in \mathcal{P}$, called the *parent* of π (and π is called a child of π'), such that $\pi' < \pi$ and there is no other pattern $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ with $\pi' < \pi'' < \pi$.

Induction property: Given $\pi \in \mathcal{P} \backslash S$, there is a position o of π with $\pi|_o \in \mathcal{X}$ (the *inductive position*), and constructors $c_1, \ldots, c_n \in \mathcal{C}$ with $c_i \neq c_j$ for $i \neq j$, such that, for all $\pi_1, \ldots, \pi_n$ which have the parent π, $\pi_i = \pi[c_i(\overline{x_{n_i}})]_o$ (where $\overline{x_{n_i}}$ are new distinct variables) for all $1 \leq i \leq n$.

The definitional trees are similar to standard matching trees of FP[3].However, differently from left-to-right matching trees used in either Hope, Miranda, or Haskell, definitional trees deal with *dependencies* between arguments of functional patterns. As a good point, optimality is achieved when definitional trees are used (in this sense, they are closer to *matching dags* or *index trees* for TRSs [12, 11]). Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves contain all (and only) the rules used to define f and whose inner nodes contain information to guide the (optimal) pattern matching during the evaluation of expressions. Each inner node contains a *pattern* and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\overline{x_n})$, where $\overline{x_n}$ are different variables.

It is often convenient and simplifies the understanding to provide a graphic representation of definitional trees, where each node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding (underlined) rules. For instance, the definitional tree for the function "$\leqslant$":

$$\begin{array}{rcllr} \texttt{0} & \leqslant & \texttt{N} & \to \texttt{true} & (R_1) \\ \texttt{s(M)} & \leqslant & \texttt{0} & \to \texttt{false} & (R_2) \\ \texttt{s(M)} & \leqslant & \texttt{s(N)} & \to \texttt{M} \leqslant \texttt{N} & (R_3) \end{array}$$

[3] Definitional trees can also be encoded using *case expressions*, another well-known technique to implement pattern matching in FP [18].

can be illustrated as follows:

$$\begin{tikzcd}
 & \boxed{\mathtt{X}} \leqslant \mathtt{Y} \arrow[dl, dash] \arrow[dr, dash] & & \\
\underline{\mathtt{0} \leqslant \mathtt{Y} \rightarrow \mathtt{true}} & & \mathtt{s(X')} \leqslant \boxed{\mathtt{Y}} \arrow[dl, dash] \arrow[dr, dash] & \\
 & \underline{\mathtt{s(X')} \leqslant \mathtt{0} \rightarrow \mathtt{false}} & & \underline{\mathtt{s(X')} \leqslant \mathtt{s(Y')} \rightarrow \mathtt{X'} \leqslant \mathtt{Y'}}
\end{tikzcd}$$

A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential. An inductively sequential TRS can be viewed as a set of definitional trees, each defining a function symbol. There can be more than one definitional tree for an inductively sequential function. In the following we assume that there is a fixed definitional tree for each defined function.

The operational semantics of FP is based on *rewriting*. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l \rightarrow r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. The instantiated lhs $\sigma(l)$ is called a *redex*. $\rightarrow^+$ denotes the transitive closure of $\rightarrow$ and $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$. By giving priority to *innermost* or, alternatively *outermost* redexes, we obtain two different rewriting principles, corresponding to the so called *eager* (*strict* or *call-by-value*) and *lazy* (*non-strict*, *call-by-name* or *call-by-need*) functional programs, respectively.

On the other hand, the operational semantics of integrated languages is usually based on *narrowing*, a combination of variable instantiation and reduction. Formally, $t \leadsto_{p,R,\sigma} s$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \rightarrow_{p,R} s$. Modern functional–logic languages are based on *needed narrowing* and *inductively sequential* programs (a detailed description can be found in [4]).

3 Instantiation

Following [7], the instantiation rule is used in pure functional transformation systems to introduce, or more appropriately, to replace substitution instances of an existing equation in a program. Similarly to the implicit instantiation produced by the narrowing calculus, the goal here is to enable subsequent rewriting steps. This is not an easy task, since there may exist many different alternatives to produce bindings and redexes, and many approaches do not offer correctness results or/and they do not consider instantiation explicitly [7, 20]. In this section we face this problem by defining a safe instantiation rule inspired in the needed narrowing based unfolding transformation for FLP of [2]. This last rule mirrors the needed narrowing calculus by implicitly performing instantiation operations that use more precise bindings than mgu's and generate redexes at different positions on rules.

For the definition of needed narrowing we assume that t is a term rooted with a defined function symbol and $\mathcal{P}$ is a definitional tree with $\mathit{pattern}(\mathcal{P}) = \pi$ such that $\pi \leq t$. We define a function λ from terms and definitional trees to sets of tuples (position, rule, substitution) such that, for all $(p, R, \sigma) \in \lambda(t, \mathcal{P})$, $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*. Function λ is not only useful for defining needed narrowing, but also for defining our instantiation rule. We consider two cases for $\mathcal{P}$[4]:

1. If π is a leaf, then $\lambda(t, \mathcal{P}) = \{(\Lambda, \pi \to r, id)\}$, where $\mathcal{P} = \{\pi\}$, and $\pi \to r$ is a variant of a rewrite rule.
2. If π is a branch, consider the inductive position o of π and some child $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. Let $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ be the definitional tree where all patterns are instances of π_i. Then we consider the following cases for the subterm $t|_o$:

$$\lambda(t,\mathcal{P}) \ni \begin{cases} (p, R, \sigma \circ \tau) & \text{if } t|_o = x \in \mathcal{X}, \tau = \{x \mapsto c_i(\overline{x_n})\}, \text{ and} \\ & (p, R, \sigma) \in \lambda(\tau(t), \mathcal{P}_i); \\ (p, R, \sigma \circ id) & \text{if } t|_o = c_i(\overline{t_n}) \text{ and } (p, R, \sigma) \in \lambda(t, \mathcal{P}_i); \\ (o.p, R, \sigma \circ id) & \text{if } t|_o = f(\overline{t_n}) \text{ for } f \in \mathcal{F}, \text{ and } (p, R, \sigma) \in \lambda(t|_o, \mathcal{P}') \\ & \text{where } \mathcal{P}' \text{ is a definitional tree for } f. \end{cases}$$

Informally speaking, needed narrowing applies a rule, if possible (case 1), or checks the subterm corresponding to the inductive positions of the branch (case 2): if it is a variable, it is instantiated to the constructor of some child; if it is already a constructor, we proceed with the corresponding child; if it is a function, we evaluate it by recursively applying function λ. Thus, the strategy differs from lazy functional languages only in the instantiation of free variables. Note that we compose in each recursive step during the computation of λ the substitution with the local substitution of this step (which can be the identity).

Now, we re-use function λ for defining our instantiation rule. Given the last (inductively sequential) program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, and a rule $(l \to r) \in \mathcal{R}_k$ where r is rooted by a defined function symbol with definitional tree $\mathcal{P}_r$, we may get program $\mathcal{R}_{k+1}$ by application of the instantiation rule as follows:

$$\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{l \to r\} \ \cup \ \{\sigma(l) \to \sigma(r) \mid (p, R, \sigma) \in \lambda(r, \mathcal{P}_r)\}.$$

In order to illustrate our instantiation rule, consider again a program $\mathcal{R}$ with the set of rules defining "$\leqslant$" together with the new rules for `test` and `double`:

$$\begin{array}{rll} \texttt{test(X,Y)} \to \texttt{X} \leqslant \texttt{double(Y)} & & (R_4) \\ \texttt{double(0)} \to 0 & & (R_5) \\ \texttt{double(s(X))} \to \texttt{s(s(double(X)))} & & (R_6) \end{array}$$

[4] This description of a needed narrowing step is slightly different from [4] but it results in the same needed narrowing steps.

Then, function λ computes the following set for the initial term $\mathtt{X} \leqslant \mathtt{double(Y)}$: $\{(\Lambda, R_1, \{\mathtt{X} \mapsto \mathtt{0}\}),\ (2, R_5, \{\mathtt{X} \mapsto \mathtt{s(X')}, \mathtt{Y} \mapsto \mathtt{0}\}),\ (2, R_6, \{\mathtt{X} \mapsto \mathtt{s(X')}, \mathtt{Y} \mapsto \mathtt{s(Y')}\})\}$. Now, by application of the instantiation rule, we replace R_4 in $\mathcal{R}$ by the rules:

$$\begin{array}{rr}
\mathtt{test(0,Y)} \to \mathtt{0} \leqslant \mathtt{double(Y)} & (R_7)\\
\mathtt{test(s(X'),0)} \to \mathtt{s(X')} \leqslant \mathtt{double(0)} & (R_8)\\
\mathtt{test(s(X'),s(Y'))} \to \mathtt{s(X')} \leqslant \mathtt{double(s(Y'))} & (R_9)
\end{array}$$

In contrast with the usual instantiation rule used in pure FP and similarly to the needed narrowing calculus used in FLP, we observe two important properties enjoyed by our transformation rule:

1. It is able to instantiate more than a unique subterm at a time. In fact, subterms at positions Λ and 2 have been considered in the example. Observe that, if, for instance, we only perform the operation on subterm at position Λ, only rule R_7 could be achieved and then, the correctness of the transformation would be lost (i.e., function `test` would remain undefined for other terms different from 0 as first parameter).
2. It uses more specific unifiers than mgu's. This is represented by the extra binding $\mathtt{X} \mapsto \mathtt{s(X')}$ applied in rules R_8 and R_9. Observe that the generation of this extra binding is crucial to preserve the structure (inductive sequentiality) of the transformed program. Otherwise, the lhs's of rules R_8 and R_9 would (erroneously) be $\mathtt{test(X,0)}$ and $\mathtt{test(X,s(Y'))}$ that unify with the lhs of rule R_7.

Experiences in FLP show that the preservation of the program structure is the key point to prove the correctness of a transformation system based on needed narrowing [2, 1]. Since the instantiation rule is not explicitly used in [2], we now formally establish the following result specially formulated for this (purely functional) transformation in an isolated way[5].

Theorem 1. *The application of the instantiation rule to an inductively sequential program generates an inductively sequential program too.*

4 Unfolding

In this section we focus on the counterpart of the instantiation rule, that is, the unfolding transformation which basically consists of replacing a rule by a new one obtained by the application of a rewriting step to a redex in its rhs. This transformation is strongly related to the previous one, since it is usually used in pure FP to generate redexes in the rhs of program rules. In our case, we take advantage once again of the information generated by function λ to proceed with our rewriting based unfolding rule. Hence, given the last (inductively sequential)

[5] This property is formally proved in [1], since instantiation is automatically performed by the (needed narrowing based) unfolding rule presented there.

program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, and a (previously instantiated) rule $(\sigma(l) \to \sigma(r)) \in \mathcal{R}_k$ such that $(p, R, \sigma) \in \lambda(r, \mathcal{P}_r)$, then we may get program $\mathcal{R}_{k+1}$ by application of the unfolding rule as follows:

$$\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{\sigma(l) \to \sigma(r)\} \ \cup \ \{\sigma(l) \to r' \mid \sigma(r) \to_{p,R} r'\}.$$

Observe that, as defined in Section 2, given a term r, if $(p, R, \sigma) \in \lambda(r, \mathcal{P}_r)$ then $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*. Hence, a program obtained by substituting one of its rules, say $l \to r$, by the set $\{\sigma(l) \to r' \mid r \leadsto_{p,R,\sigma} r'\}$, can be considered as the result of transforming it by applying the appropriate instantiation and unfolding steps. In fact, the unfolding rule based on needed narrowing presented in [2] is defined in this way, and for this reason it is said that instantiation is naturally embedded by unfolding in FLP.

Continuing now with our example, we can unfold rule R_7 by rewriting the redex at position Λ of its rhs (i.e., the whole term $\texttt{0} \leqslant \texttt{double(Y)}$) and using rule R_1, obtaining the new rule R_{10}: $\texttt{test(0,Y)} \to \texttt{true}$. Similarly, the unfolding of rules R_8 and R_9 (on subterms at position 2 of its rhs's, with rules R_5 and R_6 respectively) generates:

$$\begin{array}{rl}\texttt{test(s(X'),0)} \to \texttt{s(X')} \leqslant \texttt{0} & (R_{11}) \\ \texttt{test(s(X'),s(Y'))} \to \texttt{s(X')} \leqslant \texttt{s(s(double(Y')))} & (R_{12})\end{array}$$

Finally, rules R_{11} and R_{12} admit empty instantiations and can be unfolded with rules R_2 and R_3 respectively, obtaining:

$$\begin{array}{rl}\texttt{test(s(X'),0)} \to \texttt{false} & (R_{13}) \\ \texttt{test(s(X'),s(Y'))} \to \texttt{X'} \leqslant \texttt{s(double(Y'))} & (R_{14})\end{array}$$

Let us see now the effects that would be produced (in our original program) by the application of unfolding steps preceded by the classical instantiation steps used in traditional functional transformation systems. Starting again with rule R_4, we try to generate a redex in its rhs by instantiation. We have seen in Section 3 that, by applying the binding $\{\texttt{X} \mapsto \texttt{0}\}$ to R_4 in order to enable a subsequent unfolding step with rule R_1, we obtain an incorrect program. Hence, we prefer to act on subterm $\texttt{double(Y)}$ in rule R_4 in order to later reduce its associated redexes with rules R_5 and R_6. This is achieved by applying the ("minimal") bindings $\{\texttt{Y} \mapsto \texttt{0}\}$ and $\{\texttt{Y} \mapsto \texttt{s(Y')}\}$ to R_4:

$$\begin{array}{rl}\texttt{test(X,0)} \to \texttt{X} \leqslant \texttt{double(0)} & (R'_7) \\ \texttt{test(X,s(Y'))} \to \texttt{X} \leqslant \texttt{double(s(Y'))} & (R'_8)\end{array}$$

This process is called "minimal" instantiation in [8], since it uses the mgu's obtained by exhaustively unifying the subterm to be converted in a redex with the lhs's of the rules in the program. The corresponding unfolding steps on R'_7 and R'_8 return:

$$\begin{array}{rl}\texttt{test(X,0)} \to \texttt{X} \leqslant \texttt{0} & (R'_9) \\ \texttt{test(X,s(Y'))} \to \texttt{X} \leqslant \texttt{s(s(double(Y')))} & (R'_{10})\end{array}$$

Now, we can perform two instantiation steps, one for rule R'_9 and one more for rule R'_{10}, using the bindings $\{\mathtt{X} \mapsto \mathtt{0}\}$ and $\{\mathtt{X} \mapsto \mathtt{s(X')}\}$, and obtaining the rules (previously generated by our method) R_{11} and R_{12} together with:

$$\begin{array}{rl} \mathtt{test(0,0)} \to \mathtt{0} \leqslant \mathtt{0} & (R'_{11}) \\ \mathtt{test(0,s(Y'))} \to \mathtt{0} \leqslant \mathtt{s(s(double(Y')))} & (R'_{12}) \end{array}$$

Finally, the unfolding of R'_{11} and R'_{12} returns:

$$\begin{array}{rl} \mathtt{test(0,0)} \to \mathtt{true} & (R'_{13}) \\ \mathtt{test(0,s(Y'))} \to \mathtt{true} & (R'_{14}) \end{array}$$

Observe now that our method, instead of producing these last rules, only generates the unique rule R_{10} that subsumes both ones. Moreover, compared with the traditional transformation process, our method is faster apart from producing smaller (correct) programs. Our transformation rules inherit these nice properties from the optimality of needed narrowing (see [4]) with respect to:

1. the (shortest) length of successful derivations for a given goal which, in our framework, implies that less instantiation and unfolding steps are needed when building a transformation sequence, and
2. the independence of (minimal) computed solutions associated to different successful derivations, which guarantees that no redundant rules are produced during the transformation process.

5 Folding and Related Rules

In order to complete our transformation system, let us now introduce the folding rule, which is a counterpart of the previous transformation, i.e., the compression of a piece of code into an equivalent call. Roughly speaking, in FP the folding operation proceeds in a contrary direction to the usual reduction steps, that is, reduction is performed against a reversed program rule. Now we adapt the folding operation of [2] to FP. Given the last (inductively sequential) program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, and two rules belonging to R_k, say $l \to r$ (the "folded" rule) and $l' \to r'$ (the "folding" rule), such that there exists an operation rooted subterm of r which is an instance of r', i.e., $r|_p = \theta(r')$, we may get program $\mathcal{R}_{k+1}$ by application of the folding rule as follows:

$$\mathcal{R}_{k+1} = \mathcal{R}_k \setminus \{l \to r\} \ \cup \ \{l \to r[\theta(l')]_p\}.$$

For instance, by folding rule $\mathtt{f(s(0))} \to \mathtt{s(f(0))}$ with respect to rule $\mathtt{g(X)} \to \mathtt{f(X)}$, we obtain the new rule $\mathtt{f(s(0))} \to \mathtt{s(g(0))}$. Observe that substitution θ used in our definition is not a unifier but just a matcher. This is similar to many other folding rules for LP and FLP, which have been defined in a similar "functional style" (see, e.g., [17, 21, 2]), and can still produce at a very low cost many powerful optimizations.

In order to increase the optimization power of our folding rule we now introduce a new kind of transformation called definition introduction rule. So, given the last (inductively sequential) program $\mathcal{R}_k$ in a *transformation sequence* $\mathcal{R}_0, \ldots, \mathcal{R}_k$, we may get program $\mathcal{R}_{k+1}$ by adding to $\mathcal{R}_k$ a new rule, called *definition rule* or *eureka*, of the form $f(\overline{x}) \to r$, where f is a *new* function symbol not occurring in the sequence $\mathcal{R}_0, \ldots, \mathcal{R}_k$ and $\mathcal{V}ar(r) = \overline{x}$. Now, we can reformulate our folding rule by imposing that the "folding" rule be an eureka belonging to any program in the transformation sequence whereas the "folded" rule never be an eureka. Note that these new applicability conditions enhances the original definition by relaxing the (strong) condition that the "folding" rule belong to the last program in the sequence, which in general is crucial to achieve effective optimizations [21, 17, 2]. Moreover, the possibility to unsafely fold a rule by itself ("self folding") is disallowed.

Let us illustrate our definitions. Assume a program containing the set of rules defining "+" together with the following rule R_1 : $\texttt{twice(X,Y)} \to \texttt{(X+Y)+(X+Y)}$. Now, we apply the definition introduction rule by adding to the program the following eureka rule R_2: $\texttt{new(Z)} \to \texttt{Z} + \texttt{Z}$. Finally, if we fold R_1 using R_2, we obtain rule R_3: $\texttt{twice(X,Y)} \to \texttt{new(X+Y)}$. Note that the new definition of `twice` enhances the original one, since rules R_2 and R_3 can be seen as a *lambda lifted* version (inspired in the one presented by [20]) of the following rule, which is expressed by means of a local declaration built with the `where` construct typical of functional languages: $\texttt{twice(X,Y)} \to \texttt{Z} + \texttt{Z}$ `where` $\texttt{Z} = \texttt{X} + \texttt{Y}$. This example shows a novel, nice capability of our definition introduction and folding rules: they can appropriately be combined in order to implicitly produce the effects of the so-called *abstraction rule* of [7, 20] (often known as *where–abstraction rule* [17]). The use of this transformation is mandatory in order to obtain the benefits of the powerful tupling strategy [7, 8, 15], as we will see in the following section[6].

The set of rules presented so far is correct and complete (for FP), as formalized in the following theorem, which is an immediate consequence of the strong correctness of the transformation system (for FLP) presented in [2][7].

Theorem 2. *Let $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$ be a transformation sequence, t a term without new function symbols and s a constructor term. Then, $t \to^* s$ in $\mathcal{R}_0$ iff $t \to^* s$ in $\mathcal{R}_n$.*

6 Some examples

In the following, we illustrate the power of our transformation system by tackling some representative examples regarding the optimizations of composition and tupling (more experiments can be found in [14]). Both strategies were originally introduced in [7] for the optimization of pure functional programs (see also [17]).

[6] In [2] we formalize an abstraction rule that allows the abstraction of different expressions simultaneously.

[7] We omit the formal proof here since it directly corresponds to the so called *invariant I1* proved there. Proof details can be found in [1].

By using the composition strategy (or its variants), one may avoid the construction of intermediate data structures that are produced by some function `g` and consumed as inputs by another function `f`, as illustrates the following example where function $\texttt{sum\_prefix(X,Y)}$ (defined in the following program $\mathcal{R}_0$) returns the sum of the `Y` consecutive natural numbers, starting from `X`.

$$\begin{array}{rll@{\quad}rll}
\texttt{sum\_prefix(X,Y)} \to & \texttt{suml(from(X),Y)} & (R_1) & \texttt{from(X)} \to & \texttt{[X|from(s(X))]} & (R_4)\\
\texttt{suml(L,0)} \to & \texttt{0} & (R_2) & \texttt{0+X} \to & \texttt{X} & (R_5)\\
\texttt{suml([H|T],s(X))} \to & \texttt{H+suml(T,X)} & (R_3) & \texttt{s(X)+Y} \to & \texttt{s(X+Y)} & (R_6)
\end{array}$$

Note that function `from` is non-terminating (which does not affect the correctness of the transformation). We can improve the efficiency of $\mathcal{R}_0$ by avoiding the creation and subsequent use of the intermediate, partial list generated by the call to function `from` :

1. Definition introduction: $\texttt{new(X,Y)} \to \texttt{suml(from(X),Y)} \quad (R_7)$
2. Now, since function λ applied to $\texttt{suml(from(X),Y)}$ returns the couple of tuples $(\Lambda, R_2, \{\texttt{Y} \mapsto \texttt{0}\})$ and $(\texttt{1}, R_4, \{\texttt{Y} \mapsto \texttt{s(Y')}\})$, we apply the instantiation rule as follows:
$$\begin{array}{rl}\texttt{new(X,0)} \to \underline{\texttt{suml(from(X),0)}} & (R_8)\\ \texttt{new(X,s(Y'))} \to \texttt{suml(}\underline{\texttt{from(X)}}\texttt{,s(Y'))} & (R_9)\end{array}$$
3. Unfolding of the underlined subtems on rules R_8 and R_9 using R_2 and R_4:
$$\begin{array}{rl}\texttt{new(X,0)} \to \texttt{0} & (R_{10})\\ \texttt{new(X,s(Y'))} \to \texttt{suml([X|from(s(X))],s(Y'))} & (R_{11})\end{array}$$
4. Instantiation of rule R_{11} using the tuple (Λ, R_3, id):
$$\texttt{new(X,s(Y'))} \to \underline{\texttt{suml([X|from(s(X))],s(Y'))}} \; (R_{12})$$
Observe that the unifier generated by the call to function λ is empty and hence, it is not relevant for the instantiation step itself. However, we need the more significant position and rule information to perform the next unfolding step.
5. Unfolding of the underlined subterm in R_{12} using R_3 (note that this is infeasible with an eager strategy):
$$\texttt{new(X,s(Y'))} \to \texttt{X} + \underline{\texttt{suml(from(s(X)),Y')}} \; (R_{13})$$
6. Folding of the underlined subterm in rule R_{13} using the eureka rule R_7:
$$\texttt{new(X,s(Y'))} \to \texttt{X} + \texttt{new(s(X),Y')} \quad (R_{14})$$
7. Folding rule R_1 using R_7: $\texttt{sum\_prefix(X,Y)} \to \texttt{new(X,Y)} \;\; (R_{15})$

Then, the transformed and enhanced final program is formed by rules R_{15}, R_{10} and R_{14} (together with the initial definitions for $+$, `from`, and `suml`). Note that the use of instantiation and unfolding rules based on needed narrowing is essential in the above example. It ensures that no redundant rules are produced and it also allows the transformation to succeed even in the presence of nonterminating functions.

On the other hand, the tupling strategy essentially proceeds by grouping calls with common arguments together so that their results are computed only once. When the strategy succeeds, multiple traversals of data structures can be avoided, thus transforming a nonlinear recursive program into a linear recursive one [17]. The following well-known example illustrates the tupling strategy.

$$\begin{array}{rll} \mathtt{fib(0)} \to \mathtt{s(0)} & & (R_1) \\ \mathtt{fib(s(0))} \to \mathtt{s(0)} & & (R_2) \\ \mathtt{fib(s(s(X)))} \to \mathtt{fib(s(X))} + \mathtt{fib(X)} & & (R_3) \end{array}$$

(together with the rules for addition +). Observe that this program has an exponential complexity that can be reduced to linear by applying the tupling strategy as follows:

1. Definition introduction: $\mathtt{new(X)} \to (\mathtt{fib(s(X))}, \mathtt{fib(X)})$ (R_4)
2. Instantiation of rule R_4 by using the intended tuples $(\mathtt{1}, R_2, \{\mathtt{X} \mapsto \mathtt{0}\})$ and $(\mathtt{1}, R_3, \{\mathtt{X} \mapsto \mathtt{s(X')}\})$:

$$\begin{array}{rll} \mathtt{new(0)} \to (\underline{\mathtt{fib(s(0))}}, \mathtt{fib(0)})) & & (R_5) \\ \mathtt{new(s(X'))} \to (\underline{\mathtt{fib(s(s(X')))}}, \mathtt{fib(s(X'))}) & & (R_6) \end{array}$$

3. Unfolding (the underlined subterms in) rules R_5 and R_6 using R_2 and R_3, respectively, and obtaining the new rules:

$$\begin{array}{rll} \mathtt{new(0)} \to (\mathtt{s(0)}, \mathtt{fib(0)}) & & (R_7) \\ \mathtt{new(s(X'))} \to (\mathtt{fib(s(X'))} + \mathtt{fib(X')}, \mathtt{fib(s(X'))}) & & (R_8) \end{array}$$

   Before continuing with our transformation sequence we firstly reduce subterm $\mathtt{fib(0)}$ in rule R_7 in order to obtain the following rule R_9 which represents a case base definition for `new`: $\mathtt{new(0)} \to (\mathtt{s(0)}, \mathtt{s(0)})$ (R_9). This kind of *normalizing* step that does not require a previous instantiation step is performed automatically by the tupling algorithm described in [15].
4. In order to proceed with the abstraction of rule R_8, we decompose the process in two low level transformation steps as described in Section 5, obtaining the new rules R_{10} and R_{11}:
   – Definition introduction: $\mathtt{new\_aux((Z_1, Z_2))} \to (\mathtt{Z_1} + \mathtt{Z_2}, \mathtt{Z_1})$
   – Folding R_8 w.r.t. R_{10}: $\mathtt{new(s(X'))} \to \mathtt{new\_aux((fib(s(X')), fib(X')))}$
5. Folding R_{11} using R_4: $\mathtt{new(s(X'))} \to \mathtt{new\_aux(new(X'))}$ (R_{12})
6. Abstraction of R_3 obtaining the new rules R_{13} and R_{14}:
   – Definition introduction: $\mathtt{fib\_aux((Z_1, Z_2))} \to \mathtt{Z_1} + \mathtt{Z_2}$
   – Folding R_3 w.r.t. R_{13}: $\mathtt{fib(s(s(X)))} \to \mathtt{fib\_aux((fib(s(X)), fib(X)))}$
7. Folding R_{14} using R_4: $\mathtt{fib(s(s(X)))} \to \mathtt{fib\_aux(new(X))}$ (R_{15})

Now, the transformed program (with linear complexity thanks to the use of the recursive function `new`), is composed by rules R_1, R_2 and R_9 together with:

$$\begin{array}{ll} \mathtt{fib(s(s(X)))} & \to \mathtt{Z_1} + \mathtt{Z_2} \ \mathtt{where}\ (\mathtt{Z_1}, \mathtt{Z_2}) = \mathtt{new(X)} \\ \mathtt{new(s(X))} & \to (\mathtt{Z_1} + \mathtt{Z_2}, \mathtt{Z_1}) \ \mathtt{where}\ (\mathtt{Z_1}, \mathtt{Z_2}) = \mathtt{new(X)} \end{array}$$

where the abstracted and folded rules (R_{13}, R_{15}) and (R_{10}, R_{12}) are expressed by using local declarations for readability.

7 Conclusions

We have defined a safe instantiation rule that combined with other transformation rules for unfolding, folding, abstracting and introducing new definitions, is able to optimize lazy functional programs. The main novelty of our approach is that it is inspired in the kind of instantiation that needed narrowing implicitly produces before reducing a given term. As a nice consequence, our transformation methodology inherits the best properties of needed narrowing and enhances previous pure FP approaches in several senses. For instance, it preserves the structure of transformed programs, it is able to use specific (as opposed to most general) unifiers and it produces redexes at different positions of a given rule after being instantiated with different unifiers, thus minimizing not only the set of transformed rules but also the transformation effort.

Beyond instantiation, the set of rules presented so far has been implemented and proved correct/complete in FLP i.e., all them preserve the semantics of values (corresponding to its functional dimension) and computed answer substitutions (which is associated to its logic counterpart). Since the operational semantics of FLP is based on narrowing, and narrowing subsumes rewriting (i.e., the operational semantics of FP), this directly implies that the whole transformation system specially tailored for FP is also semantics preserving in this context, which guarantees its effective use in real tools. For the future, we plan to extend our transformation system in order to cope with lazy functional programs with different pattern matching semantics, higher-order and sharing.

Acknowledgements. Many thanks to María Alpuente, Moreno Falaschi and Germán Vidal for collaborative work on fold/unfold transformations in FLP and helpful discussions on its extensions.

References

1. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Transformation-based strategies for lazy functional logic programs. Technical Report DSIC-II/23/99, UPV, 1999. Available in URL: `http://www.dsic.upv.es/users/elp/papers.html`.
2. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, pages 147–162. Springer LNCS 1722, 1999.

3. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, New York, 1994. ACM Press.
5. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
6. R.S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21(1):239–250, 1984.
7. R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
8. W. Chin. Towards an Automated Tupling Strategy. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, 1993*, pages 119–132. ACM, New York, 1993.
9. B. Courcelle. Recursive Applicative Program Schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 459–492. Elsevier, Amsterdam, 1990.
10. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
11. M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.
12. G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443. The MIT Press, Cambridge, MA, 1992.
13. L. Kott. Unfold/fold program transformation. In M.Nivat and J.C. Reynolds, editors, *Algebraic methods in semantics*, chapter 12, pages 411–434. Cambridge University Press, 1985.
14. G. Moreno. A Safe Transformation System for Optimizing Functional Programs. Technical Report DIAB-02-07-27, UCLM, 2002. Available in URL: `http://www.info-ab.uclm.es/~personal/gmoreno/gmoreno.htm`.
15. G. Moreno. Automatic Optimization of Multi-Paradigm Declarative Programs. In F. J. Garijo, J. C. Riquelme, and M. Toro, editors, *Proc. of the 8th Ibero–American Conference on Artificial Intelligence, IBERAMIA'2002*, pages 131–140. Springer LNAI 2527, 2002.
16. G. Moreno. Transformation Rules and Strategies for Functional-Logic Programs. *AI Communications, IO Press (Amsterdam)*, 15(2):3, 2002.
17. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
18. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
19. C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language: An approach to instantiation. In K. Davis and R. J. M. Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 133–141, London, UK, 1990. Springer-Verlag.
20. D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.
21. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming, Uppsala, Sweden*, pages 127–139, 1984.

A Memoization Technique for Functional Logic Languages

Salvador España and Vicent Estruch

Departamento de Sistemas Informáticos y Computación-DSIC
Technical University of Valencia, C. de Vera s/n, 46022 Valencia, Spain.
{sespana,vestruch}@dsic.upv.es

Abstract. Declarative multi-paradigm languages combine the main features of functional and logic programming, like laziness, logic variables and non-determinism. Pure functional and pure logic programming have for long time taken advantage of tabling or memoization schemes [15, 23, 6], which motivates the interest in adapting this technique to the integrated paradigm. In this work, we introduce an operational description of narrowing with memoization whose main aim is to provide the basis for a complete sequential implementation. Since the success of memoization critically depends on efficiently indexing terms and accessing the table, we study the adequacy of known term representation techniques. Along this direction, we introduce a novel data structure for representing terms and show how to use this representation in our setting.

1 Introduction

Declarative multi-paradigm languages [12] combine the main features of functional and logic programming. In comparison with functional languages, such integrated languages are more expressive thanks to the ability to perform function inversion and to implement partial data structures by means of logical variables. With respect to logic languages, multi-paradigm languages have a more efficient operational behaviour since functions allow more deterministic evaluations than predicates. The most important operational principle of functional logic languages is narrowing [21], a combination of resolution from logic programming and term reduction from functional programming. Essentially, narrowing instantiates the term variables so that a reduction step is possible, and then reduces the instantiated term.

Several techniques from pure functional and pure logic languages have been extended to implement multi-paradigm languages. In particular, both pure paradigms have exploited memoization[1]. This technique is useful to eliminate redundant computations. The central idea is to memoize sub-computations in a table and reuse their results later. Although this technique entails an overhead which depends on the cost of accessing the table, in some situations a drastic reduction

[1] From now on, we will use this terminology even though other terms (tabling, caching, etc.) are commonly used to refer to the same concept too.

of the asymptotic cost can be obtained. In fact, memoization can be viewed as an automatic technique for applying dynamic programming optimization [8].

The performance advantages of memoization have long been known to the functional programming community [15, 7]. A memoized function remembers the arguments to which it has been applied together with the results it generates on them. Tamaki and Sato [23] proposed an interpretation for logic programming based on memoization; this seminal paper has stimulated a large body of work [6, 19, 17, 5]. In addition to reuse previous computed sub-goals, infinite paths in the search space can be detected, enabling better termination properties (indeed termination can be guaranteed for programs with the *bounded term size property* [6]).

From the aforementioned properties, it becomes natural to adapt memoization to the integrated functional logic paradigm. This adaptation is not straightforward due to some differences between functional evaluation and narrowing. Firstly, in addition to the computed normal form, functional logic programs also produces a computed answer. Secondly, non-determinism of narrowing computations leads to a large, possibly infinite, set of results arising from different sequences of narrowing steps. Previous work attempting to introduce memoization in narrowing [4] focused in finding a finite representation of a (possibly infinite) narrowing space by means of a graph representing the narrowing steps of a goal.

The main motivation of our work is to provide an operational description of a complete (i.e., without backtracking) narrowing mechanism taking advantage of a memoization technique inspired in the graph representation proposed in [4]. This description is intended to be used in a compiler for flat programs [14] into C++, which is currently under development[2]. Since the success of memoization critically depends on efficiently accessing the tables, we face the problem of data structure representation and indexing [19, 20, 18, 22, 9, 10]. In this direction, we introduce a novel data structure for representing terms and show how to use this representation in our setting.

The rest of the paper is organized as follows. In the next section we introduce some foundations in order to describe the theoretical context of our approach. Section 3 presents the main ideas of our proposal and Section 4 presents its operational description. Section 5 describes some characteristics of the implementation under development, focusing in the term representation. Finally, Section 6 sketches planned extensions and offers some conclusions.

2 Foundations

In this section, we recall some definitions and notations about term rewriting systems. We also describe the kernel of a modern functional logic language whose execution model combines lazy evaluation with non-determinism.

Terms are built from a (many-sorted) signature which is divided into two groups: constructors (e.g. a,b,c) and functions or operations (e.g. f,g,h), and

[2] `http://www.dsic.upv.es/users/elp/soft.html`

variables (e.g. u,v,w,x,y,z) as usual. A *constructor term* is a term without operation symbols. We say that a term is *operation-rooted* (resp. *constructor-rooted*) if it has a defined function symbol (resp. a constructor symbol) at the outermost position. *Values* are terms in head normal form, i.e., variables or constructor-rooted terms. The set of variables which occurs in a term t, is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is $\emptyset$ then t is a *ground term.* A *position* p in a term t is represented by a sequence of natural numbers. $t|_p$ specifies a subterm which occurs in a position p in t, and $t[s]_p$ denotes the result of replacing the subterm $t|_p$ by the term s.

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$, if $i \neq j$) and $\sigma(x) = x$ for all other variables x. Substitutions are extended to morphisms on terms as usual.

Recent proposals of multi-paradigm programming consider as source programs inductively sequential systems [3] and a combination of needed narrowing [21] and residuation as operational semantics [13]. A *flat representation* of programs has been introduced in order to express pattern-matching by case expressions [14]. In this work we will consider only the following subset of the flat representation:

$$\begin{array}{lll}
P & ::= D_1 \ldots D_m & \\
D & ::= f(x_1, \ldots, x_n) = e & \\
e & ::= x & \text{(variable)} \\
 & \mid\ c(e_1, \ldots, e_n) & \text{(constructor call)} \\
 & \mid\ f(e_1, \ldots, e_n) & \text{(function call)} \\
 & \mid\ \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(rigid case)} \\
 & \mid\ \mathit{fcase}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(flexible case)} \\
 & \mid\ e_1\ \mathit{or}\ e_2 & \text{(disjunction)} \\
p & ::= c(x_1, \ldots, x_n) & \text{(pattern)}
\end{array}$$

where P denotes a program, D a function definition, p a pattern and e an expression. The general form of a case expression is

$$(f)\mathit{case}\ e\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$$

where e is an expression, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions (possibly containing nested $(f)\mathit{case}$ expressions). The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the corresponding variables of the subexpression e_i. The difference between *case* and *fcase* only shows up when the argument e is a free variable: *case* suspends whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression. Without lost of generality, we will assume some restrictions in flat programs in the following: on the one hand, all $(f)\mathit{case}$ arguments are variables; on the other, *or* and $(f)\mathit{case}$ expressions appear at the outermost positions i.e., they do not appear inside function and constructor calls.

Example 1. The following flat program will be used to illustrate some examples throughout the paper. It defines some functions on natural numbers which are represented by terms built from `Zero` and `Succ`. `coin` nondeterministically computes `Zero` or `Succ(Zero)`. `add` is the arithmetic addition and `leq` defines the relation "less than or equal to".

$$\begin{tikzcd}
\mathit{leq(u,add(u,v))} \arrow[r, "\{u \mapsto \mathit{Zero}\}"] \arrow[d, bend right=60, "\{u \mapsto \mathit{Succ(w)}\}"'] & \mathit{true} \\
\mathit{leq(Succ(w),Succ(add(w,v)))} \arrow[u, bend right=60, "\{w \mapsto u\}"'] &
\end{tikzcd}$$

Fig. 1. Graph representation of the narrowing space of the goal $leq(u, add(u, v))$

```
double(x) = add(x, x)
coin      = Zero or Succ(Zero)
add(x, y) = fcase x of {Zero → y; Succ(m) → Succ(add(m, y))}
leq(x, y) = fcase x of {Zero → True;
                        Succ(m) → fcase y of {Zero → False;
                                              Succ(n) → leq(m, n)}}
```

3 The proposed approach

Previous work aiming at introducing memoization in narrowing [4] focused in finding a finite representation of a (possibly infinite) narrowing space by means of a graph representing the narrowing steps of a goal. Let us consider the example of Figure 1 extracted from [4]. This figure shows the graph representation[3] of the narrowing space of the goal $leq(u, add(u, v))$.

The vertices of this graph are goals i.e., terms being narrowed, and the edges are narrowing steps between these goals. Terms are considered the same vertex if they differ only by a renaming of variables. Solutions can be obtained by composing the set of substitutions which appear in any path in the graph whose origin is the goal vertex and whose destination vertex is a value. The set of computed answers in this example can be represented by $\{u \mapsto Succ^n(Zero)\}$ where $Succ^n$ denotes the composition of $Succ(Succ(\ldots))$ n times; this is called a regular substitution [4] (an extension of regular expressions to the domain of sets of substitutions) and can be obtained with (a modification of) any algorithm that computes the regular expression accepted by a finite state automaton.

Despite the possibility of representing an infinite set of solutions in a finite way, some difficulties have to be considered, from a practical point of view, when an implementation is envisaged. Note that the addition of a new single edge in the graph could change the entire regular substitution. This is the reason why retrieving solutions at different times might produce results more informative than previous ones. In order to overcome this problem, the retrieval of results

[3] For simplicity, we omitted some intermediate steps which basically correspond to $(f)case$ reductions.

might be delayed until the entire graph is constructed. Unfortunately, this is operationally incomplete if the graph is not finite.

As we will see below, our approach is based on the idea of explicitly introduce in the graph the search space of all derived subgoals. This way, some goals should be instantiated with the answers computed for subgoals. If these answers were represented by regular substitutions, we must face the problem of instantiating terms with regular substitutions.

Our proposal is based in a graph representation too. In contrast to [4], we do not use regular substitutions to represent the sets of results. The following points summarizes the proposed approach:

- Every subgoal is explicitly introduced in the graph in order to take more profit of memoization. Therefore, the graph represents no longer the search space of the original goal, but also contains the search space of *all derived subgoals.*
- Search spaces from different goals share common parts in order to further improve memoization. For instance, consider the narrowing space of the goal $double(coin)$ shown in Figure 3. The term $add(coin, coin)$ denoted by `v2` requires subgoal *coin* at position 1 to be reduced, thus a new vertex `v3` associated to *coin* is introduced in the graph. Later, `v2` is reduced to `v3`. Therefore, `v3` has played a double role.
- Computed expressions may be generated at any time. Thus, a dynamic association is needed. Every pending evaluation that needs a goal to be solved is stored in a pending table together with the set of the set of goal's solutions used to continue the evaluation. Obviously, these solutions are computed only once for every goal and this set is shared by all pending evaluations that depends on the same goal.
- Solutions are represented as pairs composed by a value and a computed answer.
- Path finding algorithms [8] are used to obtain the solutions associated to a goal. These algorithms may efficiently detect whether or not a graph updating changes the set of solutions associated to a goal.
- Non-determinism (for instance, variable instantiation in a $(f)case$) is addressed by considering every possible choice and creating an evaluation associated to it. The number of choices is finite for a single step. Since the computation process must pursue only a single branch in a given moment, we maintain a set of *ready* evaluations where they are inserted. Several strategies may be used to select the next evaluation from the set *Ready* to continue. For instance, *breadth-first* or *iterative deepening* does not compromise completeness. Note that despite non-determinism, *only one* graph is considered which memoizes the information of all these branches as soon as they are computed.

4 Operational Description

In this section we introduce a state transition system LNTM, which stands for LNT [14] with Memoization. A LNTM-State belongs to

$$\mathit{Eval} \times \mathit{Ready} \times \mathit{Suspended} \times \mathit{Graph}$$

where the following domains are used to define it:

Eval : is ($\mathit{EvaluationState} \cup \{\mathit{void}\}$)

EvaluationState: is defined by $\mathit{Term} \times \mathit{Expr} \times \mathit{Substitution}$, where *Term* and *Substitution* denote respectively the set of terms and substitutions described in section 2. The first component of *EvaluationState* indicates which goal is being solved. *Expr* differs from terms in that elements of *Expr* may contain *or* operations and $(f)\mathit{case}$ expressions. The special symbol *void* does not belong to the *EvaluationState* domain.

Ready: is the domain $2^{\mathit{EvaluationState}}$

Suspended: set of partial mappings from *EvaluationState* to $2^{\mathit{Substitution} \times \mathit{Term}}$. All suspended *EvaluationState*s have a $(f)\mathit{case}$ expression at the outermost possition and this expression has a function call as argument. This field stores, for every suspended *EvaluationState*, all solutions of its goal argument that had been used to continue its computation.

Graph: is the set of partial mappings from *Term* to $2^{\mathit{Substitution} \times \mathit{Term}}$ a set of pairs (edge label, destination vertex). $\mathit{Vertices}(G)$ will denote $\mathit{Dom}(G)$

The partial mappings Suspended and Graph described above consider terms modulo variable renaming, although variable names are used in the examples to represent substitutions between adjacent terms in the graph.

LNTM-state transitions are described in Figure 2. The following auxiliary functions are used to simplify the description:

ObtainSolution: Searches a path in the graph from a given vertex to a value vertex and composes the path's sequence of substitutions in order to obtain the computed answer. This function is non-deterministic, but it must guarantee that every solution will be sometime found.

UnionMap: This function performs the union of two partial mappings and is used to update the mapping *Suspended* and the *Graph*. $\mathit{Dom}(\mathit{UnionMap}(G, H)) = \mathit{Dom}(G) \cup \mathit{Dom}(H)$. $\mathit{UnionMap}(G, H)(x) = G(x)$ if $x \notin \mathit{Dom}(H)$, and vice-versa. $\mathit{UnionMap}(G, H)(x) = G(x) \cup H(x)$ whenever both exists. For instance, $\mathit{UnionMap}(G, \{t \to \{\}\})$ adds a new vertex t to G.

Unfold: Given a function call $f(t_1, \ldots, t_n)$, *Unfold* creates a new *EvaluationState* $\langle f(t_1, \ldots, t_n), e, \sigma\rangle$ where e is obtained by unfolding $f(t_1, \ldots, t_n)$ and σ is the associated substitution. Example: $\mathit{Unfold}(\texttt{add}(\texttt{coin}, \texttt{coin}))$ is:

$$\begin{array}{lll}\langle \texttt{add}(\texttt{coin}, \texttt{coin}),\ \mathit{fcase}\ \texttt{coin}\ \mathit{of}\ \{ & \texttt{Zero} & \to \texttt{coin}; \\ & \texttt{Succ}(\texttt{m}) & \to \texttt{Succ}(\texttt{add}(\texttt{m}, \texttt{coin})), \{\}\rangle\end{array}$$

Given an initial goal $g(e_1, \ldots, e_n)$, the initial LNTM-state associated to it is the tuple: $\langle void, \{\langle g(e_1, \ldots, e_n), g(e_1, \ldots, e_n), \{\}\rangle\}, \emptyset, \emptyset\rangle$. The calculus proceed until the set *Ready* is empty and the rule (sol) cannot be applied.

To illustrate the description, a trace of *double*(*coin*) is shown in Figure 3.

Let us briefly explain the aforementioned rules. (sel) and (sol) are the only ones which overlap and the only non-deterministic too. They may be applied to a LNTM-state with *void* in the *Eval* field:

(sel) This rule takes an *EvaluationState* element of the set *Ready* following some given strategy (e.g., *breadth-first*, *iterative deepening*, etc.) and puts it in the field *Eval* to be used later by other rules.

(sol) A new solution is searched in the graph for some vertex v corresponding to a goal appeared in the $(f)case$ argument of some suspended *EvaluationState*. A new *EvaluationState* is created from the suspended one by replacing its goal argument by the founded solution. It is stored in the set *Ready* to be selected later. The mapping *Suspended* is updated to reflect the fact that this solution has already been used.

The rest of the rules are non-overlapping and deterministic:

(or) The outermost position in the field *Eval* is an *or* expression. This rule breaks this expression into two who are introduced in the set *Ready* to be selected later.

(val) The field *Eval* must contain a value in order to apply this rule. A new edge in the graph is inserted connecting the goal being solved to the value.

(goal) The outermost position in the field *Eval* is a function call $f(t_1, \ldots, t_n)$. This expression is always a term because of the restrictions imposed to flat programs. An edge is added from the goal being solved to this function call when they are different. If the goal was not in the graph, a new evaluation $Unfold(f(t_1, \ldots, t_n))$ is introduced in the set *Ready* to be selected later.

The three following rules (casec), (casef) and (casev) correspond to the case where an expression of the form $(f)case\ e\ of\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$ is at the outermost position of the field *Eval*.

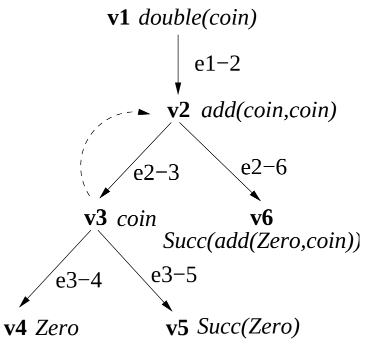
(casec) This rule is used when e in the above expression is constructor rooted (with root c_i). The corresponding pattern $c_i(\overline{x_{n_i}})$ is selected and matched against e. The associated expression e_i replaces the current expression of the field *Eval*. Other fields remain unchanged.

(casef) In this case, e is a function call $f(t_1, \ldots, t_n)$ which must be computed. The current *EvaluationState* of field *Eval* is stored in the mapping *Suspended*. The evaluation $\langle f(t_1, \ldots, t_n), f(t_1, \ldots, t_n), \{\}\rangle$ is also introduced in the set *Ready*.

(casev) This rule is applied to a *fcase* expresion whenever e is an unbound variable. This variable may be bound to every $c_i(\overline{x_{n_i}})$ pattern. Thus, a new evaluation is inserted in the set *Ready* for every possible variable instantiation. This set of *EvaluationState*s is finite and equal to the arity of the *fcase* expression.

Rule	*Eval*	*Ready*	*Suspended*	*Graph*
(sel)	$void$	R	S	G
	$\Longrightarrow e$	$R - \{e\}$	S	G
	$where\ R \neq \emptyset, e \in R$			
(sol)	$void$	R	S	G
	$\Longrightarrow void$	$R \cup \{e\}$	S'	G
	$where\ s = \langle t, (f)case\ f(t_1, \ldots, t_n)\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}, \sigma\rangle$			
	$s \in Dom(S)$			
	$\langle r, \varphi\rangle = ObtainSolution(G, f(t_1, \ldots, t_n))$			
	$f(t_1, \ldots, t_n) \in Vertices(G)$			
	$\langle r, \varphi\rangle \notin S(s)$			
	$e = \langle t, (f)case\ r\ of\ \{p_1 \to \varphi(e_1); \ldots; p_n \to \varphi(e_n)\}, \varphi \circ \sigma\rangle$			
	$S' = UnionMap(S, \{s \to \{\langle r, \varphi\rangle\}\})$			
(or)	$\langle t, e_1\ or\ e_2, \sigma\rangle$	R	S	G
	$\Longrightarrow void$	R'	S	G
	$where\ R' = R \cup \{\langle t, e_1, \sigma\rangle, \langle t, e_2, \sigma\rangle\}$			
(val)	$\langle t, val, \sigma\rangle$	R	S	G
	$\Longrightarrow void$	R	S	G'
	$where$ val is in head normal form			
	$G' = UnionMap(G, \{t \to \{\langle \sigma, val\rangle\}\})$			
(goal)	$\langle t, f(t_1, \ldots, t_n), \sigma\rangle$	R	S	G
	$\Longrightarrow void$	R'	S	G'
	$where$			
	$G' = \begin{cases} UnionMap(G, \{t \to \{\langle \sigma, f(t_1, \ldots, t_n)\rangle\}\}) & \text{if } t \neq f(t_1, \ldots, t_n) \vee \sigma \neq \{\} \\ UnionMap(G, \{t \to \{\}\}) & \text{otherwise} \end{cases}$			
	$R' = \begin{cases} R \cup \{Unfold(f(t_1, \ldots, t_n))\} & \text{if } f(t_1, \ldots, t_n) \notin vertices(G) \\ R & \text{otherwise} \end{cases}$			
(casec)	e	R	S	G
	$\Longrightarrow e'$	R	S	G
	$where\ e = \langle t, (f)case\ c(b_1, \ldots, b_k)\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}, \sigma\rangle$			
	$p_i = c(x_1, \ldots, x_k)$ is the pattern that matches $c(b_1, \ldots, b_k)$			
	$\varphi = \{x_1 \mapsto b_1, \ldots, x_k \mapsto b_k\}$			
	$e' = \langle t, \varphi(e_i), \varphi \circ \sigma\rangle$			
(casef)	e	R	S	G
	$\Longrightarrow void$	R'	S'	G
	$where\ e = \langle t, (f)case\ f(t_1, \ldots, t_n)\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}, \sigma\rangle$			
	$S' = UnionMap(S, \{e \to \{\}\})$			
	$R' = R \cup \{\langle f(t_1, \ldots, t_n), f(t_1, \ldots, t_n), \{\}\rangle\}$			
(casev)	e	R	S	G
	$\Longrightarrow void$	R'	S	G
	$where\ e = \langle t, fcase\ x\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}, \sigma\rangle$			
	$\varphi_i = \{x \mapsto p_i\},\ \forall i = 1, \ldots, n$			
	$R' = R \cup \{\langle t, \varphi_i(e_i), \varphi_i \circ \sigma\rangle : i \in \{1, \ldots, n\}\}$			

Fig. 2. Operational description (LNTM Calculus)



ev1	=	⟨double(coin), double(coin), {}⟩
ev2	=	⟨double(coin), add(coin, coin), {}⟩
ev3	=	⟨add(coin, coin), *fcase* coin *of* {Zero → coin;
		Succ(m) → Succ(add(m, coin)), {}⟩
ev4	=	⟨coin, coin, {}⟩
ev5	=	⟨coin, Zero *or* Succ(Zero), {}⟩
ev6	=	⟨coin, Zero, {}⟩
ev7	=	⟨coin, Succ(Zero), {}⟩
ev8	=	⟨add(coin, coin), *fcase* Zero *of* {Zero → coin;
		Succ(m) → Succ(add(m, coin)), {}⟩
ev9	=	⟨add(coin, coin), coin, {}⟩
ev10	=	⟨add(coin, coin), *fcase* Succ(Zero) *of*
		{Zero → coin; Succ(m) → Succ(add(m, coin)), {}⟩
ev11	=	⟨add(coin, coin), Succ(add(Zero, coin)), {}⟩

Step	Rule	Eval	Ready	Suspended	Graph
		void	{ev1}	∅	∅
1	sel	ev1	∅	∅	∅
2	goal	*void*	{ev2}	∅	⟨{v1}, {}⟩
3	sel	ev2	∅	∅	⟨{v1}, {}⟩
4	goal	*void*	{ev3}	∅	⟨{v1,v2}, {e1-2}⟩
5	sel	ev3	∅	∅	⟨{v1,v2}, {e1-2}⟩
6	casef	*void*	{ev4}	{ev3 → {}}	⟨{v1,v2}, {e1-2}⟩
7	sel	ev4	∅	{ev3 → {}}	⟨{v1,v2}, {e1-2}⟩
8	goal	*void*	{ev5}	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
9	sel	ev5	∅	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
10	or	*void*	{ev6, ev7}	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
11	sel	ev6	{ev7}	{ev3 → {}}	⟨{v1,v2,v3}, {e1-2}⟩
12	val	*void*	{ev7}	{ev3 → {}}	⟨{v1,v2,v3,v4}, {e1-2,e3-4}⟩
13	sel	ev7	∅	{ev3 → {}}	⟨{v1,v2,v3,v4}, {e1-2,e3-4}⟩
14	val	*void*	∅	{ev3 → {}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
15	sol	*void*	{*ev*8}	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
16	sel	*ev*8	∅	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
17	casec	*ev*9	∅	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5}⟩
18	goal	*void*	∅	{ev3 → {Zero}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
19	sol	*void*	{*ev*10}	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
20	sel	*ev*10	∅	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
21	casec	*ev*11	∅	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5}, {e1-2,e3-4,e3-5,e2-3}⟩
22	val	*void*	∅	{ev3 → {Zero, Succ(Zero)}}	⟨{v1,v2,v3,v4,v5,v6}, {e1-2,e3-4,e3-5,e2-3,e2-6}⟩

Fig. 3. Trace of the goal *double*(*coin*). Bottom table corresponds to the sequence of LNTM-states. The graphs in this sequence are represented as pairs of sets: The first component is the set of vertices. The second component is the set of edges. Both sets use the notation of the graph in the top-left part of the figure. Top left is the search space graph; dashed edges represent dependence relations between goals and suspended evaluations in these vertices; edges are no labeled since this example has not variables. Top right table corresponds to *EvaluationState*s used in the trace.

5 Term representation

In this section, we outline some aspects concerning the applicability of the proposed framework. The benefits of memoization are obtained at the expense of space memory and an overhead which depends on the cost of accessing the tables. Memory consumption is a critical issue in our approach, because terms in the graph are never deleted. Moreover, terms are not modified; instead, new terms are created from previous ones. This is the reason why we address the problem of term representation.

Several indexing data structures for term representation have been studied in the context of pure logical and pure functional languages (and also in others contexts like automated theorem proving, see [20] for a survey). Given their tree like structure, representing terms by trees seems to be a natural choice. An extension is to represent terms as *dags* (directed acyclic graphs), like WAM-terms, so that common subterms can be shared. Other representation is *flatterms* which is a linear representation of the preorder traversal of a term. A *discrimination tree* [19] is a trie-like structure where terms are considered linear data structures (like *flatterms*) and common prefixes are shared. *Substitution tree* [11] is an indexing technique where the tree structure is preserved and common parts are shared via variable instantiation.

Most of these data structures are indented to store a large number of terms and support the fast retrieval, for any given query term t, of all terms in the index satisfying a certain relation with t, such as matching, unifiability, or syntactic equality. Nevertheless, in our framework, only syntactic equality modulo variable renaming (i.e., *variant check*) is needed. Other required operations are subterm extraction and term construction by composing previous subterms. We believe that sharing common substructures fulfills these requirements. Tries share parts in a very limited way; substitution trees share common subparts, but it is difficult to create new terms from arbitrary common subterms. This is is the reason why a novel term representation is introduced.

The proposed novel term representation combines *dag* and linear representations. A term is decomposed into two parts: skeleton and binding information. Skeletons are like terms without the binding of variables, they are built from the same many-sorted signature as terms, together with a special 0-ary symbol "$*$" which represents *any* variable. Binding information is represented by the list of variables obtained by a preorder traversal of the term. This information will be represented by a finite sequence of natural numbers $[n_1, \ldots, n_k]$. The skeleton and the binding information of a term can be obtained by means of functions $sk : Term \to Skeleton$ and $vlist : Term \to Binding$ which are defined:

$$sk(t) = \begin{cases} t & t \text{ is a 0-ary functor or constructor} \\ * & t \text{ is a variable} \\ f(sk(t_1), \ldots, sk(t_n)) & t \text{ is } f(t_1, \ldots, t_n) \end{cases}$$

$$vlist(t) = \begin{cases} [\,] & t \text{ is a 0-ary functor or constructor} \\ [t] & t \text{ is a variable} \\ vlist(t_1) {+}{+} \cdots {+}{+} vlist(t_n) & t \text{ is } f(t_1, \ldots, t_n) \end{cases}$$

where we use ++ to denote list concatenation. For instance, given a term $t = add(x, add(x, y))$, we have $sk(t) = add(*, add(*, *))$, and $vlist(t) = [x, x, y]$.

A binding $[n_1, \ldots, n_k]$ is normalized if $n_1 = 1$ and, for every i, such that $n_i > 1$, there exists $j < i$ with $n_j = n_i - 1$. Example: binding $[1, 2, 3, 1, 1]$ is normalized but $[1, 3, 2]$ is not because $n_2 = 3$ and $n_j \neq 2$ for all $j < 2$. Roughly speaking, normalization consist of assigning a new natural number to every new encountered variable in the sequence in order to obtain a canonical representation. That is, for every finite sequence $[s_1, \ldots, s_k]$, there exists a unique normalized binding b and a one-to-one mapping $\varphi : \{s_i : i = 1, \ldots, k\} \to \mathbb{N}$ such that $[\varphi(s_1), \ldots, \varphi(s_k)]$ is normalized. Let $norm : Var \to Binding$ be the function that obtains the normalized binding associated to a list of variables. For instance, $norm([x, y, y, x]) = [1, 2, 2, 1]$, $norm([2, 1]) = [1, 2]$. The above functions allow us to represent a term t by the tuple $\langle sk(t), norm(vlist(t))\rangle$. Given a term t, $numvar(t)$ denotes the number of variables in t. When applied to a skeleton, this function counts the number of "$*$", so $numvar(t) = numvar(sk(t))$.

Skeletons are represented in a global *dag* so that common parts are always shared. Every different skeleton is assigned to a unique (numerical) identifier. These identifiers indexes an array where the root term and child identifiers are stored. Therefore, given a skeleton identifier, subterm extraction can be performed efficiently. The cost of obtaining a subterm at position $n_1 \cdots n_k$ is $O(k)$.

Other important operation is skeleton construction. Given a functor symbol f with arity n, and given n ordered skeletons s_i with identifiers id_i, we want to obtain the skeleton identifier associated to $f(s_1, \ldots, s_n)$. This operation can be done with hashing [8] in $O(n)$. The number of "$*$" is also stored in the skeleton table, this information is needed to relate the skeleton and binding parts of the terms. Bindings are also stored in an array, although they could be organized in a trie too. The length and number of different variables is also stored. Given a sequence of length n, the binding identifier is obtained with hashing in $O(n)$. Binding normalization may also be computed in $O(n)$.

In this way, nodes in the graph are represented by 2 numerical identifiers. Edges of this graph represent substitutions: a mapping from numerical identifiers to pairs of skeleton and binding identifiers. Bindings appearing in substitutions are not normalized, since they are related to the destination node. For example, Figure 4 shows the data structures needed to represent the graph from Figure 1.

The advantages of this representation in terms of memory space are obvious. For example, $n \cdot m$ different terms may be represented with n different skeletons (with the same number k of "$*$"), and m bindings with the same length k. Moreover, this representation solves completely the *variant check* problem.

Memory reduction is not the only criterion to select this representation, it is also necessary to show that the proposed representation suffices to perform all the required operations. Some obvious propositions arise from the constructive definitions of skeletons and bindings:

Proposition 1. *Given a representation $\langle s, b\rangle$ of a term t, the length of b is the same as the number of "$*$" in s i.e. $numvar(sk(t)) = length(vlist(t))$.*

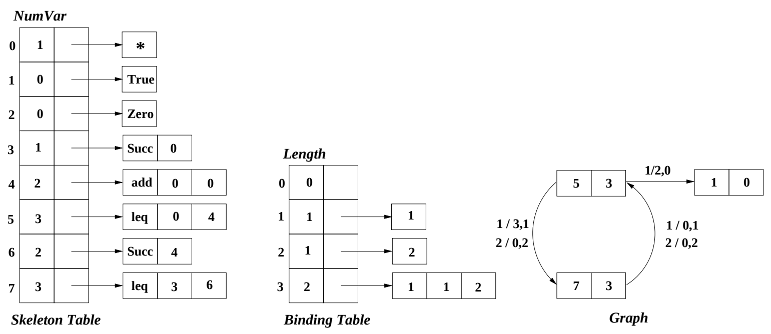


Fig. 4. Data structures representing the graph from Figure 1. The second field of the Skeleton Table stores the root term and the indexes of arguments. These indexes refer to entries in the same table. The second field of the Binding Table is a list of natural numbers whose length is given by the first field. Every vertex in the graph has two values, the skeleton index and the binding index. For instance, vertex 5,3 refer to $\langle leq(*, add(*, *)), [1, 1, 2]\rangle$

Proposition 2. *Let r be a subterm of s at position p i.e. $s|_p = r$. Then $sk(r)$ is a subterm of $sk(s)$ at the same position i.e. $sk(s)|_p = sk(r)$.*

As a consequence of *vlist* definition and propositions 1 and 2:

Proposition 3. *Let s be a term and r a subterm of s at position p, i.e. $s|_p = r$. The binding of r is a normalized interval of the binding of s; moreover this interval can be determined from $sk(s)$ and position p.*

Proposition 4. *Given a representation $\langle s, n_1 \dots n_k\rangle$ of a term t, for every i, $1 \le i \le k$, it is possible to obtain the position p in s such that $t|_p$ is associated to the i^{th} variable occurrence.*

Proposition 5. *If two terms have the same skeleton and normalized bindings, these terms are syntactically equivalent modulo variable renaming.*

The basic operations required in our proposal are:

Subterm extraction This is a direct consequence of Propositions 2 and 3.

Variable instantiation Let $\langle s, b\rangle$ be a representation of a term t and let σ a substitution $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$. $sk(\sigma(t))$ is obtained by replacing in $sk(t)$ the "$*$" associated to x_i (Proposition 4) by $sk(t_i)$. $vlist(\sigma(t))$ is obtained by replacing in $vlist(t)$ the values x_i by $vlist(t_i)$. Since bindings have lost the information about variables, bindings in substitutions are related to bindings in the term and, therefore, they are not normalized. For example, $\sigma(leq(u, add(u, v))) = leq(Succ(x), Succ(add(x, y)))$ where $\sigma = \{u \mapsto s(x), v \mapsto y\}$. The corresponding skeleton binding tuples associated to these terms are

$$\langle leq(*, add(*, *)), [1, 1, 2]\rangle \text{ and } \langle leq(Succ(*), Succ(Add(*, *))), [1, 1, 2]\rangle$$

respectively, where σ can be represented by $\{1 \mapsto \langle Succ(*), [1]\rangle, 2 \mapsto \langle *, [2]\rangle\}$. Note that $\langle *, [2]\rangle$ is not normalized.

Note that these operations may be significantly improved when they are compiled from a flat program.

6 Conclusions and future work

We have presented an operational description of narrowing with memoization whose main aim is to provide the basis for a complete sequential implementation. The narrowing space is viewed as a graph where generated subgoals are explicitly represented. A novel term representation which combines *dag* (directed acylic graph) and linear representations has been proposed which may be used for both representing the graph and performing the narrowing steps.

We should note that sharing substructures in the term representation does not restrict the use of this representation to functional logic programs with sharing [16]. Indeed, the goal *double*(*coin*) has three different results as can be shown in Figure 3 and these three results are obtained with this representation.

A compiler based on these ideas is under development; experimental results are needed to show the practical viability of this memoization technique and the proposed term representation. As a future work, the computed graph may be used for other purposes such as partial evaluation [2] and debugging [1].

References

1. M. Alpuente, F. J. Correa, and M. Falaschi. Debugging Scheme of Functional Logic Programs. In M. Hanus, editor, *Proc. of International Workshop on Functional and (Constraint) Logic Programming, WFLP'01*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
2. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
3. S. Antoy. Definitional Trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.
4. S. Antoy and Z. M. Ariola. Narrowing the narrowing space. In *9th Int'l Symp. on Prog. Lang., Implementations, Logics, and Programs (PLILP'97)*, volume 1292, pages 1–15, Southampton, UK, 9 1997. Springer LNCS.
5. J. Barklund. Tabulation of Functions in Definite Clause Programs. In Manuel Hermenegildo and Jaan Penjam, editors, *Proc. of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, pages 465–466. Springer Verlag, 1994.
6. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.
7. B. Cook and J. Launchbury. Disposable Memo Functions. *ACM SIGPLAN Notices*, 32(8):310–318, 1997.

8. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.
9. B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. *ACM SIGPLAN Notices*, 34(3):97–106, 1999.
10. B. Demoen and K. Sagonas. CAT: The Copying Approach to Tabling. *Lecture Notes in Computer Science, Springer-Verlag*, 1490:21–35, Setp. 1998.
11. P. Graf. Substitution Tree Indexing. Technical Report MPI-I-94-251, Saarbruecken, 1994.
12. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.
14. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
15. J. Hughes. Lazy memo-functions Lazy memo-functions. Proc. of the 2nd Conference on Functional Programming Languages and Computer Architecture (FPCA). *Lecture Notes in Computer Science*, 201:129–146, 1985.
16. F.J. López-Fraguas J.C. González-Moreno, M.T. Hortalá-González and M. Rodríguez-Artalejo. An Approach to Declarative Programming based on a Rewriting Logic. *Journal of Logic Programming*, (40):47–87, 1999.
17. B. Mertens M. Leuschel and K. Sagonas. Preserving Termination of Tabled Logic Programs while Unfolding. In Proc. of LOPSTR'97: Logic Program Synthesis and Transformation, N. Fuchs,Ed. LNCS 1463:189–205, 1997.
18. A. Riazanov R. Nieuwenhuis, T. Hillenbrand and A. Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In Proc of the First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proc. LNCS 2083:257–271, 2001.
19. I. V. Ramakrishnan, P. Rao, K. F. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, 1999.
20. J. Rivero. Data Structures and Algorithms for Automated Deduction with Equality. Phd thesis, Universitat Politecnica de Catalunya, Barcelona, 2000.
21. R. Echahed S. Antoy and M. Hanus. A Needed Narrowing Strategy. In Journal of the ACM (JACM). volume 47, pages 776–822. ACM Press New York, NY, USA, 2000.
22. K. Sagonas and M. Leuschel. Extending Partial Deduction to Tabled Execution: Some Results and Open Issues. *ACM Computing Surveys*, 30(3es), 1998.
23. H. Tamaki and T. Sato. Old Resolution with Tabulation. In *3rd International Conference on Logic Programming*, volume Springer LNCS 225, pages 84–98, 1986.

Integrating Finite Domain Constraints and CLP with Sets

A. Dal Palù[1], A. Dovier[1], E. Pontelli[2], and G. Rossi[3]

[1] Univ. di Udine, Dip. di Matematica e Informatica.
(dalpalu|dovier)@dimi.uniud.it
[2] New Mexico State University, Dept. Computer Science. epontell@cs.nmsu.edu
[3] Univ. di Parma, Dip. di Matematica. gianfr@prmat.math.unipr.it

Abstract. We propose a semantically well-founded combination of the constraint solvers used in the constraint programming languages CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$). This work demonstrates that it is possible to provide efficient executions (through CLP($\mathcal{FD}$) solvers) while maintaining the expressive power and flexibility of the CLP($\mathcal{SET}$) language. We develop a combined constraint solver and we show how static analysis can help in organizing the distribution of constraints to the two constraint solvers.

1 Introduction

Several proposals aimed at developing declarative programming frameworks that incorporate different types of *set-based primitives* have been presented (e.g., [1, 10, 13, 4, 5, 2]). These frameworks provide a high level of data abstraction, where complex algorithms can be encoded in a natural fashion, by directly using the popular language of set theory. These features make this type of languages particularly effective for modeling and rapid prototyping of algorithms. One of the main criticisms moved to these approaches is that the intrinsic computational costs of some set operations (e.g., the NP-completeness of set-unification) make them suited for fast prototyping but not for the development of efficient solutions. On the other hand, various proposals have been made to introduce constraint-based language primitives for dealing with simple forms of sets—i.e., *finite* subsets of a predefined *domain* (typically $\mathbb{Z}$). Constraint-based manipulation of finite domains provides the foundations of the popular CLP($\mathcal{FD}$) framework [15, 3], that has been demonstrated well-suited for the encoding and the fast resolution of combinatorial and optimization problems [11].

The overall goal of this work is to develop a bridge between the expressive power and the high level of abstraction offered by constraint solving over generic hereditarily finite sets—as possible in CLP($\mathcal{SET}$) [5]—and the efficient use of constraint propagation techniques provided by constraint solving over finite domains—as possible in CLP($\mathcal{FD}$). In order to accomplish these results, we extend the CLP($\mathcal{SET}$) framework by introducing integer constants and integer intervals, as a natural extension of the existing constraint handling capabilities—the new language is called CLP($\mathcal{SET}$)$^{\mathsf{int}}$. We develop novel constraint solving algorithms that allow the CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$) solvers to communicate and cooperate in verifying the satisfiability of constraint formulae over the combined structure (finite domains + hereditarily finite sets). We also present static analysis strategies for CLP($\mathcal{SET}$) programs which make it possible to automatically discover situations in which the CLP($\mathcal{FD}$) constraint solver can be used in place of the CLP($\mathcal{SET}$) solver to check constraint satisfiability. Moreover, the analyzer can detect entire fragments of CLP($\mathcal{SET}$) programs that can be translated to equivalent CLP($\mathcal{FD}$) programs and as such executed more efficiently.

This work represents a natural evolution of the previous research on programming with sets, starting from the logic language {log} [4], later developed as a constraint logic programming language (CLP($\mathcal{SET}$) [5]) and endowed with more set-theoretic primitives. The methodology proposed in this paper extends the ideas in CLP($\mathcal{SET}$) in two directions. On one hand, we provide CLP($\mathcal{SET}$) with a methodology to execute programs more efficiently, by relying on the fast propagation mechanisms developed for CLP($\mathcal{FD}$); on the other hand, we allow CLP($\mathcal{SET}$) to become a high-level language to express constraint-based manipulation of CLP($\mathcal{FD}$) domains. A similar idea is behind the work by Flener et al. [6], where the authors compile the language ESRA, that includes set-based primitives, into the constraint-based language OPL. Gervet in [7] introduces set operations over finite domains in the CLP language ECLiPSe [8]. In [17] the authors propose a tuple-based treatment of sets in ECLiPSe and compare it with Gervet's approach. These two approaches, however, do not allow a complete and easy handling of programs involving sets.

2 CLP($\mathcal{SET}$) with Intervals

The signature Σ upon which CLP($\mathcal{SET}$) [5] is based is composed of the set $\mathcal{F}$ of function symbols, that includes $\emptyset$ and $\{\cdot \,|\, \cdot\}$, the set $\Pi_C = \{=, \in, \cup_3, ||, \mathsf{set}\}$ of constraint predicate symbols, and a denumerable set $\mathcal{V}$ of variables. The intuitive semantics of the various symbols is the following: $\emptyset$ represents the empty set, while $\{t \,|\, s\}$ represents the set composed of the elements of the set s plus the element t (i.e., $\{t \,|\, s\} = \{t\} \cup s$). $=$ and $\in$ represent the equality and the membership relations. $\cup_3$ represents the union relation: $\cup_3(r, s, t)$ holds if $t = r \cup s$. The predicate $||$ captures the disjoint relationship between two sets: $s||t$ holds if and only if $s \cap t = \emptyset$. The constraint predicate set is used to enforce the multi-sorted nature of the language; terms are separated into two sorts, one representing set terms (called `Set`) and one representing non-set terms. The predicate set checks whether a term belongs to the sort `Set` (e.g., $\mathsf{set}(\{a, b\})$ holds while $\mathsf{set}(1)$ does not). The semantics of CLP($\mathcal{SET}$) has been described with respect to a predefined structure $\mathcal{A}_{\mathcal{SET}}$ [5].

We will use the notation $\{t_1, t_2, \ldots, t_n \,|\, t\}$ for $\{t_1 \,|\, \{t_2 \,|\, \cdots \{t_n \,|\, t\} \cdots\}\}$ and the notation $\{t_1, t_2, \ldots, t_n\}$ as a shorthand for $\{t_1 \,|\, \{t_2 \,|\, \cdots \{t_n \,|\, \emptyset\} \cdots\}\}$. The *primitive constraints* in CLP($\mathcal{SET}$) are all the positive literals built from symbols in $\Pi_c \cup \mathcal{F} \cup \mathcal{V}$. A *($\mathcal{SET}$-)constraint* is a conjunction of primitive constraints and negation of primitive constraints of CLP($\mathcal{SET}$). Other basic set-theoretic operations (e.g., $\cap$, $\subseteq$) are easily defined as $\mathcal{SET}$-constraints. CLP($\mathcal{SET}$) provides also a syntactic extension called *Restricted Universal Quantifiers (RUQs)*, that allows one to write subgoals of the form $(\forall X_1 \in S_1) \cdots (\forall X_n \in S_n)(C \wedge B)$ where C is a $\mathcal{SET}$-constraint and B is a CLP($\mathcal{SET}$) program goal. The semantics of $(\forall X \in S)G$ is simply $\forall X\,(X \in S \rightarrow G)$. The implementation of RUQs is accomplished via program transformation—generating two recursive CLP($\mathcal{SET}$) clauses using set terms and constraints [4].

In this paper we assume that the constant symbols $0, 1, -1, 2, -2, \ldots$ are in $\mathcal{F}$. Let us call these terms *integer constants* (or, simply, *integers*). Let us also assume that a total order $\prec$ is defined on $T(\mathcal{F})$ (see [5] for details) such that $\ldots - 2 \prec -1 \prec 0 \prec 1 \prec 2 \prec \ldots$. We say that $s \preceq t$ iff $s \prec t$ or $s \equiv t$. When clear from the context, we denote with $x + y$ ($x - y$) the integer constant corresponding to the sum (difference) of the values of the two integer constants x and y. We will also use $<$ and $\leq$ with the usual meaning. These terms are seen as different, uninterpreted, constant terms. Since they are not interpreted, the constraint domain for CLP($\mathcal{SET}$) does not need to be changed.

We also introduce in $\mathcal{F}$ the binary function symbol int. A term $\mathsf{int}(t_i, t_f)$, where t_i and t_f are integer constants, is called an *interval term.* $\mathsf{int}(t_i, t_f)$ is a shorthand for the set term $\{t_i, t_i+1, \ldots, t_f-1, t_f\}$. If $t_f \prec t_i$, then $\mathsf{int}(t_i, t_f)$ is a shorthand for the set term $\emptyset$. We only allow variables or integer constants as arguments of the function symbol int. To start, however, we require the terms t_i, t_f in $\mathsf{int}(t_i, t_f)$ to be non-variable. This restriction will be relaxed in the successive sections. Terms of the form $\mathsf{int}(t_1, t_2)$ with either t_1 or t_2 not integer constants or variables are considered ill-formed terms. Occurrences of ill-formed terms in a constraint cause the CLP($\mathcal{SET}$) solver to return a false result. Moreover, if it is not ill-formed, $\mathsf{set}(\mathsf{int}(t_1, t_2))$ holds. CLP($\mathcal{SET}$) with these extensions is referred to as *CLP($\mathcal{SET}$) with intervals* or, simply, CLP($\mathcal{SET}$)$^{\mathsf{int}}$.

3 CLP($\mathcal{SET}$)$^{\mathsf{int}}$: Constraint Solving modifying $SAT_{\mathcal{SET}}$

CLP($\mathcal{SET}$) provides a sound and complete constraint solver ($SAT_{\mathcal{SET}}$) to verify satisfiability of $\mathcal{SET}$-constraints. Given a constraint C, $SAT_{\mathcal{SET}}(C)$ nondeterministically transforms C either to false (if C is unsatisfiable) or to a finite collection $\{C_1, \ldots, C_k\}$ of constraints in *solved form* [5]. A constraint in solved form is guaranteed to be satisfiable in the considered structure $\mathcal{A}_{\mathcal{SET}}$. Moreover, the disjunction of all the constraints in solved form generated by $SAT_{\mathcal{SET}}(C)$ is *equi-satisfiable* to C. For instance, $SAT_{\mathcal{SET}}(\{1, 2 \,|\, X\} = \{1 \,|\, Y\} \wedge 2 \notin X)$ returns the three solved form constraints:

- $Y = \{2 \,|\, X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$
- $X = \{1 \,|\, N\} \wedge Y = \{2 \,|\, N\} \wedge \mathsf{set}(N) \wedge 2 \notin N$
- $Y = \{1, 2 \,|\, X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$

where N is a new variable—to be treated as an existentially quantified variable. A detailed description of the CLP($\mathcal{SET}$) solver can be found in [5]; a sound and complete implementation of $SAT_{\mathcal{SET}}$ is included in the $\{log\}$-interpreter [12].

Interval terms can be treated simply as syntactic extensions and automatically removed before processing the constraints. The simplest solution is to replace each interval term $\mathsf{int}(t_i, t_f)$ with the corresponding extensional set term

$\{t_i, t_i + 1, \ldots, t_f\}$. Obviously, this is a rather inefficient solution, as it may lead to an explosion in the size of the constraint.

A more effective way to handle constraints involving interval terms is to modify the rewriting procedures of the CLP($\mathcal{SET}$) constraint solver. The modifications should guarantee that atomic constraints involving intervals (e.g., $t \in \mathsf{int}(t_i, t_f)$) have the expected semantics, but without having to expand interval terms into the corresponding set terms. The modified rewriting rules for $\in$ constraints are shown in the figure below. We omit the details of the procedures for the other constraint predicates for space reasons.

(1)	$s \in \emptyset \wedge C'$	$\mapsto$ false
(2)	$r \in \{s \,\vert\, t\} \wedge C'$	$\mapsto$ $C' \wedge r = s$ *or* $C' \wedge r \in t$
(3)	$t \in X \wedge C'$	$\mapsto$ $X = \{t \,\vert\, N\} \wedge \mathsf{set}(N) \wedge C'$
(4.1)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_f \prec t_i, t_i, t_f$ *integers*	$\mapsto$ false
(4.2)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_n \preceq t_f, t_i, t_f, t_n$ *integers*	$\mapsto$ C'
(4.3)	$X \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_f, t_i, t_f$ *integers* *and X is a variable*	$\mapsto$ $C' \wedge X = t_i$ *or* $C' \wedge X \in \mathsf{int}(t_i + 1, t_f)$
(4.4)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_f, t_i, t_f, t_n$ *integers* $t_n \prec t_i$ *or* $t_f \prec t_n$	$\mapsto$ false
(4.5)	$r \in \mathsf{int}(t_i, t_f) \wedge C'$ *r is neither a variable* *nor an integer*	$\mapsto$ false

Although this solution avoids exploding interval terms to set terms and ensures good efficiency in processing each individual membership constraint over intervals, it is still unsatisfactory whenever the constraint involves more than one interval term. For example, consider the $\mathcal{SET}$-constraint: $X \in \mathsf{int}(1, 1000) \wedge X \in \mathsf{int}(1001, 2000)$. According to the above definition of the membership rewriting procedure, solving this constraint will cause the constraint solver to repeatedly (through backtracking) generate at least 1000 integer constants to solve the first or the second constraint, before concluding that the constraint is unsatisfiable. This problem derives from the fact that the constraint solver does not fully exploit the semantic properties of interval constraints.

4 CLP($\mathcal{FD}$): Language and Constraint Solving

The language CLP($\mathcal{FD}$) [15, 3, 11] is an instance of the CLP scheme, and it is particularly suited to encode combinatorial problems. The constraint domain used by CLP($\mathcal{FD}$) contains an alphabet of constraint predicate symbols $\Pi_C = \{\{\in m..n\}, =, \neq, <, \leq, >, \geq\}$ (see, e.g. [9]) and an interpretation structure $\mathcal{A}_{\mathcal{FD}}$, whose interpretation domain is the set $\mathbb{Z}$ of the integer numbers and the interpretation function $(\cdot)^{\mathcal{FD}}$ maps symbols of the signature to elements of $\mathbb{Z}$ and to functions and relations over $\mathbb{Z}$ in the natural way. In particular, the interpretation of the constraint $u \in m..n$ is: $u \in^{\mathcal{FD}} m..n$ holds if and only if $m \leq u \leq n$.[4]

[4] $u \in m..n$ is expressed as `u in m..n` in SICStus and as `u::m..n` in ECLiPSe.

The constraint predicates $\{\in m..n\}$ are used to identify intervals representing the *domains* of the variables in the problem. Additional constraint predicates are often provided in CLP($\mathcal{FD}$) languages, e.g., cumulative constraints and labeling, but for the purpose of this paper we will only focus on the primitive constraints mentioned above. A *$\mathcal{FD}$-constraint* is a conjunction of literals built from symbols in the given signature.

Let $SAT_{\mathcal{FD}}$ be a constraint solver for the $\mathcal{FD}$-constraint domain. Here we consider $SAT_{\mathcal{FD}}$ as a black-box, i.e., a procedure that takes as input a $\mathcal{FD}$-constraint and returns a set of $\mathcal{FD}$-constraints. The resulting constraints cover the same solution space as the input constraint, but they are simplified constraints. The CLP($\mathcal{FD}$) solvers adopted in SICStus [14] and ECLiPSe [8] are incomplete solving procedures; e.g., given the goal

```
| ?- X in 1..2, Y in 1..2, Z in 1..2, X #\= Y, X #\= Z, Y #\= Z.
```

the CLP($\mathcal{FD}$) solvers provided by SICStus and ECLiPSe are not able to find out the inconsistency (actually they leave the given constraint unchanged). A complete solver can be obtained by adding chronological backtracking to force exploration of the solution space. E.g., if we conjoin the goal above with the atom `labeling([X,Y,Z])` (`labeling([], [X,Y,Z])` in SICStus) then the constraint solver will successfully determine the unsatisfiability of the constraint.

Thus, given a constraint C, using CLP($\mathcal{FD}$), we can either obtain a constraint C' implying C, possibly not in solved form and possibly unsatisfiable, or, via labeling, a solved form—possibly ground—solution. This can be compared with the case of CLP($\mathcal{SET}$), where we always obtain a disjunction of solved form satisfiable constraints C'_i (or false if C is unsatisfiable) (see Section 3).

5 CLP($\mathcal{SET}$)$^{\mathsf{int}}$: Combining Solvers

The solution we propose is to exploit the CLP($\mathcal{FD}$) constraint solver within the CLP($\mathcal{SET}$) constraint solver to solve membership constraints over intervals, by mapping the $\in$ constraints of CLP($\mathcal{SET}$) to $\{\in m..n\}$ constraints of CLP($\mathcal{FD}$), as well as equations and disequations over integers, which may occur in a $\mathcal{SET}$-constraint. Extensions that deals with $<$ and $\leq$ are discussed in Section 6. In the specific implementation described here we make use of the constraint solver offered by SICStus Prolog [14], though the proposed technique is general and applicable to other CLP($\mathcal{FD}$) solvers. Let us assume that the two languages make use of the same set of variables and the same representation of integer constants. The function τ that translates atomic $\mathcal{SET}$-constraints to $\mathcal{FD}$-constraints is defined as follows:

$$\tau(c) = \begin{cases} s \in t_i..t_f & \textit{if } c \equiv s \in \mathsf{int}(t_i, t_f) \ \ (\textit{or} \ \ s \in \{t_i, t_i+1, \ldots, t_f\}) \\ s = t & \textit{if } c \equiv s = t \ \textit{or} \ c \equiv s \ \mathsf{is} \ t \\ s \neq t & \textit{if } c \equiv s \neq t \\ \mathsf{true} & \textit{otherwise} \end{cases}$$

where s, t_i, t_f, t are either variables or integer constants. In concrete syntax, the CLP($\mathcal{SET}$) constraint $s \in \mathsf{int}(m, n)$ will correspond to the SICStus CLP($\mathcal{FD}$)

constraint `s in m..n`, whereas `s` $\neq$ `t` and `s` $=$ `t` will correspond to `s #\= t` and `s #= t`, respectively. τ can be naturally extended to conjunctions of primitive constraint. We will also refer to the inverse function τ^{-1}, where $\tau^{-1}(s \in t_i..t_f) = (s \in \mathsf{int}(t_i, t_f))$. The case $\tau^{-1}(s = t)$ is mapped to $(s \text{ is } t)$ only if s is either a variable or an integer constant, and to $(s = t)$ otherwise. It is also possible to extend the function τ to cover some other cases; for example, a CLP($\mathcal{SET}$)-constraint such as $s \in \{t_1, \dots, t_n\}$ where $t_1, \dots, t_n$ are integer constants, $\{t_1, \dots, t_n\} = i_1 \cup \cdots \cup i_k$ and $i_1, \dots, i_k$ are intervals, can be translated as $\mathtt{s\ in\ i_1 \backslash/ \cdots \backslash/ i_k}$ (SICStus Prolog's syntax).

Proposition 1. *Let C be a $\mathcal{SET}$-constraint consisting of a conjunction of literals of the form $\ell \in \mathsf{int}(t_i, t_f), \ell = r, \ell \neq r$, where t_i, t_f are integer constants and ℓ and r are integer constants or variables. Then $\mathcal{FD} \models \exists \tau(C)$ iff $\mathcal{SET} \models \exists C$.*

Proof. (Sketch) Since $\mathcal{SET}$ contains a submodel isomorphic to $\mathcal{FD}$, the $(\rightarrow)$ direction follows trivially. For the $(\leftarrow)$ direction, if σ is s.t. $\mathcal{SET} \models \sigma(C)$, we build σ' s.t. $\mathcal{FD} \models \sigma'(\tau(C))$. If $\sigma(X)$ is not an integer constant, assign $\sigma'(X)$ to a large, unused integer. If $\sigma(X) = \sigma(Y)$, use the same integer for them. $\square$

From a *practical* point of view, we know that, given a constraint C, $SAT_{\mathcal{SET}}(C)$ always detects its satisfiability or unsatisfiability. Conversely, the solvers for CLP($\mathcal{FD}$) are typically incomplete. We can obtain a complete solver by using labeling primitives. Specifically, if C is a $\mathcal{SET}$-constraint as in Proposition 1 and $\mathcal{V}_\in$ is the set of variables occurring in the $\in$-constraints, if $\mathcal{V}_\in = vars(C)$, then the CLP($\mathcal{FD}$) solver of SICStus Prolog returns a ground solution (or detect unsatisfiability) to the constraint $\tau(C) \wedge \mathtt{labeling}([vars(C)])$.

5.1 Integration of $SAT_{\mathcal{SET}}$ and $SAT_{\mathcal{FD}}$

We decompose each CLP($\mathcal{SET}$)$^{\mathsf{int}}$ constraint in three parts: $C^S \wedge C^F \wedge C^C$, where C^S is a constraint that can be handled only by a $\mathcal{SET}$ solver, C^F is a constraint that can be handled only by a $\mathcal{FD}$ solver, and C^C is a constraint that can be handled by both solvers—i.e., C^C is composed of those constraints for which τ produces a result different from true. The C^C part effectively represents a communication conduit between the two constraint solvers. From now on we will replace each constraint with a pair $\langle C, D \rangle$, where $C = C^S \wedge C^C$ and $D = C^F \wedge \tau(C^C)$. The procedure $SAT_{\mathcal{SET}}$ is modified to handle a constraint subdivided into these two components, and to distribute the different components to the proper solver.

The modified rewriting rules for membership constraints are illustrated in the figure in the next page. Whenever a domain constraint $X \in \mathsf{int}(t_i, t_f)$ is encountered in C, the solver passes the corresponding $\mathcal{FD}$-constraint $\tau(c)$ to the $SAT_{\mathcal{FD}}$ procedure, which in turn will check the satisfiability of the constraint $\tau(c) \wedge D$. If $\tau(c) \wedge D$ turns out to be unsatisfiable, then also $SAT_{\mathcal{SET}}$ will terminate with a false result, otherwise, $SAT_{\mathcal{SET}}$ will continue. Note that $SAT_{\mathcal{FD}}$ may return an improved constraint D' (e.g., by narrowing intervals through node and arc consistency); in this case $SAT_{\mathcal{SET}}$ will be reactivated on $\langle C \wedge \tau^{-1}(D'), D'\rangle$. Moreover, the component $\tau^{-1}(D')$ returned to the $\mathcal{SET}$ solver may contain different types of constraints—i.e., simplified domain, equality and inequality constraints.

(1)	$\langle s \in \emptyset \wedge C', D\rangle \}$	$\mapsto$ false
(2)	$\langle r \in \{s \mid t\} \wedge C', D\rangle \}$	$\mapsto$ $\langle C' \wedge r = s, D \wedge \tau(r = s)\rangle$ *or* $\langle C' \wedge r \in t, D \wedge \tau(r \in t)\rangle$
(3)	$\langle t \in X \wedge C', D\rangle \}$	$\mapsto$ $\langle X = \{t \mid N\} \wedge \mathsf{set}(N) \wedge C', D\rangle$
(4.1)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ *r is neither a variable nor an integer* t_i, t_f *integers* $\}$	$\mapsto$ false
(4.2)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ t_i, t_f *integers* $SAT_{\mathcal{FD}}(D \wedge \tau(r \in \mathsf{int}(t_i, t_f))) = \mathsf{false}$ $\}$	$\mapsto$ false
(4.3)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ t_i, t_f *integers* $SAT_{\mathcal{FD}}(D \wedge \tau(r \in \mathsf{int}(t_i, t_f))) = D'$ $\}$	$\mapsto$ $\langle C' \wedge \tau^{-1}(D'), D'\rangle$

Similar modifications can be applied to the procedures that handle the other primitive constraints, that we omit due to lack of space. Observe that each new constraint that is generated during $SAT_{\mathcal{SET}}$ computation should be translated with τ and made available to $SAT_{\mathcal{FD}}$ in the second part of the constraint. For example, each time we process constraints of the type $X = i$ with i integer constant, then the equation $X = i$ has to be provided to CLP($\mathcal{FD}$) as well. Observe, moreover, that $\mathcal{FD}$-constraints could be simply accumulated in the CLP($\mathcal{FD}$) constraint store, without solving them during the $SAT_{\mathcal{SET}}$ computation. The CLP($\mathcal{FD}$) constraint store could be processed by $SAT_{\mathcal{FD}}$ at the end, after the termination of $SAT_{\mathcal{SET}}$. Interleaving $SAT_{\mathcal{FD}}$ and $SAT_{\mathcal{SET}}$, as done in the rewriting rules discussed above, provides significant pruning of the search space.

5.2 The Global Constraint Solver

The global constraint solver $SAT_{\mathcal{SET}+\mathcal{FD}}$ is the same CLP($\mathcal{SET}$) constraint solver $SAT_{\mathcal{SET}}$ [5], in which CLP($\mathcal{SET}$) rewriting rules are replaced by the new rules defined in the previous section. These rules allow us to exploit the $SAT_{\mathcal{FD}}$ constraint solver whenever it is possible (and convenient).

```
$SAT_{\mathcal{SET}+\mathcal{FD}}(\langle C, D\rangle)$ :
  set_infer($C$);
  repeat
      $\langle C', D'\rangle := \langle C, D\rangle$;
      set_check($C$);
      STEP($\langle C, D\rangle$);
  until $\langle C, D\rangle = \langle C', D'\rangle$
```

$SAT_{\mathcal{SET}+\mathcal{FD}}$ uses the procedure set_check to check the set constraints and the procedure set_infer to add set constraints for those variables that are required to belong to the sort Set; set_check and set_infer are the same procedures already used in CLP($\mathcal{SET}$), suitably updated in order to account for interval terms. The STEP procedure is the core part of $SAT_{\mathcal{SET}+\mathcal{FD}}$: it calls the rewriting procedures on the constraint $\langle C, D\rangle$ in a predetermined order (to ensure termination—see [5]). At the end of the execution of the STEP procedure, non-solved primitive constraints may still occur in $\langle C, D\rangle$; therefore, the execution of STEP has to be iterated until a fixed-point is reached—i.e., the constraint cannot be simplified any further.

As mentioned in Section 2, if C is satisfiable, then $SAT_{\mathcal{SET}}$ non-deterministically returns satisfiable constraints in solved form. This is not the case for the $SAT_{\mathcal{SET}+\mathcal{FD}}$

solver. Let $\langle C', D' \rangle$ be one of the pairs returned by the procedure $SAT_{\mathcal{SET}+\mathcal{FD}}$. $C' \wedge \tau^{-1}(D')$ is no longer guaranteed to be satisfiable in $\mathcal{SET}$. For example, a possible unsatisfiable output for $SAT_{\mathcal{SET}+\mathcal{FD}}$ is:

$\big\langle \mathtt{X} \notin \mathtt{S}\ , \mathtt{X} \neq \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Z} \wedge \mathtt{Y} \neq \mathtt{Z} \wedge \mathtt{X} \in 1..2 \wedge \mathtt{Y} \in 1..2 \wedge \mathtt{Z} \in 1..2 \big\rangle$.

On the other hand, observe that $SAT_{\mathcal{SET}+\mathcal{FD}}$ still preserves the set of solutions of the original constraint, thanks to Proposition 1. A complete solver can be obtained by adding chronological backtracking to force exploration of the solution space, e.g., by using labeling predicates. However, requiring direct execution of a complete labeling within $SAT_{\mathcal{SET}+\mathcal{FD}}$ each time $SAT_{\mathcal{FD}}$ is called would cause, in practice, the loss of most of the advantages of using the $SAT_{\mathcal{FD}}$ solver—it will effectively lead to a translation of each interval into an extensional set. Also, our approach is to maintain a behavior as close as possible to that of the traditional $SAT_{\mathcal{SET}}$ solver; thus we do not want to require the user to explicitly request an enumeration of the solutions.

The solution we propose is to modify the global solver as follows. Given a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ constraint C, the resolution of C is performed according to the following steps:

step 1. The $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver is initially applied to the constraint $\langle C, \mathsf{true} \rangle$; this will lead either to false (in this case we stop) or to the constraint $\langle C', D' \rangle$.

step 2. Let D' be the $\mathcal{FD}$ part of the constraint obtained from the last call to $SAT_{\mathcal{SET}+\mathcal{FD}}$. A constraint of the form $X \in t_1..t_2$ is selected from D' and the solutions $R_1, \ldots, R_k$ are collected from the execution of $SAT_{\mathcal{FD}}(D' \wedge \mathsf{labeling}([X]))$. Each R_i, in particular, will contain a constraint of the type $X = r_i$ for some $r_i \in t_1..t_2$. The execution of $SAT_{\mathcal{SET}+\mathcal{FD}}$ is restarted from each constraint $\langle C' \wedge \tau^{-1}(R_i), R_i \rangle$. The step is repeated until there are no constraints of the form $X \in t_1..t_2$ in D'.

Additional improvements to the resolution process can be achieved through a refinement of the sorts system used by CLP($\mathcal{SET}$)$^{\mathsf{int}}$. Currently the system recognizes two sorts, `Set` and `Ker`, describing respectively set terms and non-set terms. The sort `Ker` requires now a further refinement, by distinguishing integer constants (`Int`) from all other non-set terms (`Ker`). Enforcing the sorts requires the introduction of additional constraints: $\mathsf{integer}(s)$ and $\mathsf{ker}(s)$ are satisfiable only by terms belonging to the sort `Int` and `Ker`, respectively. During the rewriting process, each time a constraint of the type $s \in \mathsf{int}(t_i, t_f)$ is generated, the following constraint is introduced: $\mathsf{integer}(s) \wedge \mathsf{integer}(t_i) \wedge \mathsf{integer}(t_f)$. Observe that if one of the endpoints of the interval (i.e., t_i or t_f) is a variable, then the new constraint will guarantee that the variable can be assigned only integer values. Constraints $\mathsf{integer}(X)$ are also generated via static program analysis, as shown in Section 7.

6 Encoding New Operators

CLP($\mathcal{FD}$) allows us to properly and efficiently deal with constraints of the form $\ell \in t_i..t_f, \ell = r, \ell \neq r, \ell < r, \ell \leq r$. Moreover, arithmetic operations can be used in terms ℓ and r. The capability of dealing with arithmetic operations and with the natural ordering between integers can be used to improve the capabilities of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ as well. We could deal with this new constraints directly in CLP($\mathcal{SET}$), since with a set-based encoding of integer terms, $<$ can be directly encoded by $\in$. However, this approach is highly inefficient, and, moreover, it turns out to be not adequate to deal with more general algebraic expressions like $X = Y + 3 \wedge X - Z \leq Y + 1$, since arithmetic function symbols are not interpreted in CLP($\mathcal{SET}$).

A more effective solution can be achieved by avoiding any direct treatment of $<$ and $\leq$ within $SAT_{\mathcal{SET}}$, and simply relying on $SAT_{\mathcal{FD}}$ to handle this type of constraints. Let us assume that the language of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ is extended as follows: $\Pi_c = \{=, \in, \cup_3, \|, \mathsf{set}, \mathsf{integer}, \leq\}$ and that $\mathcal{F}$ includes the traditional arithmetic function symbols (e.g., $+$, $-$). Let us call *integer terms* those terms built using arithmetic operators, integer constants and variables. Integer terms are interpreted on the subset of the domain of $\mathcal{SET}$ containing integer constants (i.e., on terms of sort `Int`). Interpretation of arithmetic function symbols is the expected one. Their sort will be $\langle\{\mathtt{Int}\}, \{\mathtt{Int}\}, \{\mathtt{Int}\}\rangle$.

Integer terms can occur as arguments of primitive constraints based on $=$, $\neq$, and $\leq$. Rewriting rules for these constraints are modified so that as soon as they recognize that one of the arguments of the predicate symbol at hand is an integer term, solution of the constraint is committed to $SAT_{\mathcal{FD}}$. The translation function τ is modified to allow s, t_i, t_f, t to be either variables or integer terms (not only constants), and such that $\tau(s \leq t) = s \leq t$, where s, t are either integer terms or variables. The rewriting rules for $\leq$ constraints used in $SAT_{\mathcal{SET}+\mathcal{FD}}$ are defined as follows:

(1)	$\langle s \leq t \wedge C, D\rangle$ *s or t not integers or* $SAT_{\mathcal{FD}}(D \wedge \tau(s \leq t)) = \mathsf{false}$	$\mapsto \mathsf{false}$
(2)	$\langle s \leq t \wedge C, D\rangle$ *s, t integers or variables* $SAT_{\mathcal{FD}}(D \wedge \tau(s \leq t)) = D'$	$\mapsto \langle C \wedge \tau^{-1}(D'), D'\rangle$

The definitions for the $=$ and $\neq$ constraints are updated accordingly. We assume that terms built using arithmetic operators but involving also non-integer terms are considered ill-formed terms (i.e., not respecting the sorts). For instance $X + f(a)$ is an ill-formed term. In CLP($\mathcal{SET}$)$^{\mathsf{int}}$ the presence of ill-formed syntactic objects is detected within the constraint solver procedure by exploiting sort constraints, set, $\mathsf{integer}$, and ker, automatically generated by the solver itself or provided by the user as part of the input constraint. Whenever the solver detects an inconsistency among sort constraints it terminates with a false result.

The presence of $<$ and $\leq$ allows us also to introduce new expressive constraints in $\mathcal{SET}$. For example, it becomes possible to provide a general cardinality constraint size in the context of CLP($\mathcal{SET}$): the constraint $\mathsf{size}(s, t)$ is satisfied if s is a set, t is an integer, and t corresponds to the cardinality of s. These constraints can be handled using new rewriting rules that we omit due to lack of space.

As an example of how constraints on integer terms can be conveniently exploited in CLP($\mathcal{SET}$)$^{\mathsf{int}}$ consider the problem of finding solutions for a system of linear equations. The following code shows a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ implementation for this problem, while the table shows experimental results obtained running the code using $SAT_{\mathcal{SET}}$ and using $SAT_{\mathcal{SET}+\mathcal{FD}}$.

```
start(X, Y, Z, L, H) :-
    X ∈ int(L, H)∧
    Y ∈ int(L, H)∧
    Z ∈ int(L, H)∧
    X1 = 1 + X∧
    X1 = 2 * Y + Z∧
    X2 = Z − Y ∧ X2 = 3∧
    X3 = X + Y ∧ X3 = 5 + Z.
```

	Experimental results						
L	-10	-20	-30	-10^3	-10^4	-10^5	-10^6
H	10	20	30	10^3	10^4	10^5	10^6
$SAT_{\mathcal{SET}}$	41s	8m 31s	38m	∞	∞	∞	∞
$SAT_{\mathcal{SET}+\mathcal{FD}}$	0s	0s	0s	0.13s	1.6s	19.5s	3m 58s

Solving the problem using $SAT_{\mathcal{SET}}$ is feasible but with unacceptable computation time. Conversely, the solution using the $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver is obtained in time comparable to that obtained using directly CLP($\mathcal{FD}$).

As another example, consider the well-known combinatorial problem of solving the SEND + MORE = MONEY puzzle (i.e., solve the equation by assigning a distinct digit between 0 and 9 to each letter). The following code shows a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ possible solution for this problem.

```
solve_puzzle(S, E, N, D, M, O, R, Y) :-
        {S, E, N, D, M, O, R, Y} ⊆ int(0, 9) ∧
        size({S, E, N, D, M, O, R, Y}) = 8 ∧
        M * 10000 + O * 1000 + N * 100 + E * 10 + Y  =
                    S * 1000 + E * 100 + N * 10 + D +
                    M * 1000 + O * 100 + R * 10 + E.
```

Running the program using $SAT_{\mathcal{SET}}$ results in unacceptable computational time, while using $SAT_{\mathcal{SET}+\mathcal{FD}}$ we get the solution in 0.01 seconds.

As a final remark, observe that effective solutions have been proposed to handle *cardinality* constraints in CLP($\mathcal{FD}$) [16]. This concept has a semantics related to the size of a set, but it is not the general cardinality notion. Precisely, given n variables $X_1, \ldots, X_n$ with domains $D_1, \ldots, D_n$, a cardinality constraint for a domain element d is a bound k on the number of X_i that can be simoultaneously instantiated to d. It is therefore a constraint associated to the availability of a fixed number k of resources of a certain kind d.

7 Static Analysis for Nested Sets Identification

It is possible to further improve communication between the two constraint solvers by *statically* extracting selected properties from programs. We introduce a static analysis schema for CLP($\mathcal{SET}$)$^{\mathsf{int}}$ programs, checking if the term s is a *n*ested *set* of *i*ntegers. We define $\mathsf{ni\_set}(s)$ as the measure of the depth of nesting of the sets. The intuitive semantics is the following:

- $\mathsf{ni\_set}(s) = 0$ if s in an integer constant;
- $\mathsf{ni\_set}(s) = n + 1$ if s is a set and for each $t \in s$ it holds that $\mathsf{ni\_set}(t) = n$.

(1)	$\emptyset\ \}$	$\mapsto$ $\mathsf{ni\_set}(\emptyset) := \bot$
(2)	$f(t_1,\ldots,t_k)$ $f \not\equiv \{\cdot\mid\cdot\}, f \not\equiv \mathsf{int}$, *not an integer constant* $\}$	$\mapsto$ $\mathsf{ni\_set}(f(t_1,\ldots,t_k)) := \top$
(3)	*an integer constant* $t\ \}$	$\mapsto$ $\mathsf{ni\_set}(t) := 0$
(4)	$\mathsf{int}(t_i,t_f)$ t_i, t_f *are integers and* $t_i \le t_f\ \}$	$\mapsto$ $\mathsf{ni\_set}(\mathsf{int}(t_i,t_f)) := 1$
(5)	$\{t \mid s\}\ \}$	$\mapsto$ $\mathsf{ni\_set}(\{t \mid s\}) := (\mathsf{ni\_set}(t)+1) \sqcup \mathsf{ni\_set}(s)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(\{t \mid s\}) - 1$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(\{t \mid s\})$
(6)	$s = t\ \}$	$\mapsto$ $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(s)$
(7)	$t \in s\ \}$	$\mapsto$ $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(t) \sqcup (\mathsf{ni\_set}(s) - 1))$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup (\mathsf{ni\_set}(t) + 1)$
(8)	$\cup_3(r,s,t)\ \}$	$\mapsto$ $\mathsf{ni\_set}(r) := \mathsf{ni\_set}(r) \sqcup \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t)$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(r)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(r)$

Fig. 1. Abstract Interpretation steps to compute ni_set

Observe that $\mathsf{ni\_set}(s) > 0$ implies that the constraint $\mathsf{set}(s)$ holds. If $\mathsf{ni\_set}(s) = 1$ then s is a flat set of integers (e.g., $\{1,2,3,5\}$). Moreover, we allow two additional possible values for $\mathsf{ni\_set}(s)$: $\bot$, when no information is currently available for s, and $\top$, when s is not a nested set of integers. The values for $\mathsf{ni\_set}(s)$ are elements of the lattice $\mathcal{L} = \langle\{\bot,\top,0,1,2,\ldots\},\prec\rangle$ represented in Figure 8. The join operator $\sqcup$ is naturally defined on $\mathcal{L}$. We introduce two more operations on the lattice $\mathcal{L}$, respectively denoted by "+" and "−", defined in the table below, where m, n, p denote integers, $+_{\mathbb{Z}}$ and $-_{\mathbb{Z}}$ represent the standard addition and subtraction between integers, and x is a generic element of $\mathcal{L}$:

$\bot + \bot = \bot,\ \bot + n = n + \bot = \bot,\ \top + x = x + \top = \top,\ m + n = m +_{\mathbb{Z}} n$
$\top - x = \top, m - n = m -_{\mathbb{Z}} n\ \ (m \ge n), m - n = \top\ \ (m < n),$ $\bot - \bot = \bot - n = \bot, \bot - \top = \top$

The process of deriving the values of $\mathsf{ni\_set}(\cdot)$ starts by initializing $\mathsf{ni\_set}(X)$ to $\bot$ for each variable X occurring in the CLP($\mathcal{SET}$)$^{\mathsf{int}}$ goal G under consideration. The rules in Figure 1 are then applied on the terms and subterms of G and constraint literals until a fixpoint is reached. The case $s||t$ is missing since no information concerning the elements of s and t can be obtained by the fact that they are disjoint.

It is immediate to see that the analysis is correct and that terminates in polynomial time w.r.t. the size of G. It can be further refined, by extracting additional information from the arithmetic operations. For example, from $A+B$ one can infer that $\mathsf{ni\_set}(A) = \mathsf{ni\_set}(A) \sqcup 0$ (the same for B). Similarly, the presence of interval definitions with variables as endpoints, as in $X \in \mathsf{ni\_set}(1,N)$ allows us to infer: $\mathsf{ni\_set}(N) = \mathsf{ni\_set}(N) \sqcup 0$.

For example, consider the constraint: $\mathtt{M} \in \mathsf{int}(\mathtt{1},\mathtt{10}), \mathtt{N} = \{\{\mathtt{M}\},\mathtt{M}\}, \cup_3(\{\mathtt{M}\},\emptyset,\mathtt{S})$. The analysis returns $\mathsf{ni\_set}(M) = 0, \mathsf{ni\_set}(S) = 1, \mathsf{ni\_set}(N) = \top$.

8 Analysis of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ Programs

The set of rules presented in the previous section can be used to analyze a complete CLP($\mathcal{SET}$)$^{\mathsf{int}}$ program. Let us assume w.l.o.g. that a program P is composed of a set of clauses of the form $p(X_1,\ldots,X_m) \mathrel{:\!-} C, B$ where $X_1,\ldots,X_m$

are distinct variables, C is a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ constraint, and B is a conjunction of program atoms of the form $q(Y_1,\ldots,Y_k)$, where $Y_1,\ldots,Y_k$ are distinct variables.

During the analysis, each variable is assigned an abstract value that represents its properties. Since the variables are managed with a global scope, we avoid possible name clashes that can affect the analysis, renaming the variables in such a way that each of them occurs only in one clause. We also force certain collisions that may help the analysis task. Consider the set of n clauses defining p: the variables are renamed such that all these clauses have identical heads. Thus, given $p(X_1,\ldots,X_k) \mathrel{:-} C_1, B_1, \ldots, p(Y_1,\ldots,Y_k) \mathrel{:-} C_n, B_n$, they are renamed to $p(N_1,\ldots,N_k) \mathrel{:-} C_1[X_i/N_i], B_1[X_i/N_i] \ldots p(N_1,\ldots,N_k) \mathrel{:-} C_n[Y_i/N_i], B_n[Y_i/N_i]$, where N_i are new variables. The analysis gathers more information if we consider the program together with its input . For instance, assume that the goal G contains the m variables $X_1,\ldots,X_m$ and that $X_1,\ldots,X_n$ $(n < m)$ are associated to integer numbers. Then, choosing some integer numbers $s_1,\ldots,s_n$, we can make available the information to the analysis by adding to the program P the new clause $g(X_1,\ldots,X_n,X_{n+1},\ldots,X_m) \mathrel{:-} X_1 = s_1 \wedge \ldots \wedge X_n = s_n, G$.

The analysis algorithm proceeds as follows. At the beginning, all the program variables are initialized with $\perp$ from the lattice $\mathcal{L}$. The program analysis performs the fixpoint described in the previous section to determine the $\mathsf{ni\_set}$ values, by repeatedly executing the following two steps for each clause in the program:

• *Local analysis:* The analysis described in Section 7 is applied to C.

• *Propagation of information to other clauses:* For each $q(A_1,\ldots,A_k)$ in B, the information related to formal (X_i) and actual (A_i) parameters is updated as follows: $\mathsf{ni\_set}(X_i) := \mathsf{ni\_set}(X_i) \sqcup \mathsf{ni\_set}(A_i)$; $\quad \mathsf{ni\_set}(A_i) := \mathsf{ni\_set}(X_i)$;

Domain Analysis. The static analysis scheme can be also used to infer domain information for those variables X such that $\mathsf{ni\_set}(X) = 0$. In this case, the elements of the abstraction lattice $\mathcal{D}$ are integer intervals. The order in the lattice is provided by the relation $[x,y] \leq [x',y']$ iff $x' \leq x$ and $y \leq y'$. The join operation $\sqcup$ of two intervals is: $[x,y] \sqcup [x',y'] = [\min(x,x'), \max(y,y')]$. The analysis rules are similar to those in Figure 1. As an example, consider the CLP($\mathcal{SET}$) program:

$a(X,Y) \mathrel{:-} X \in \mathsf{int}(1,3) \wedge X \neq Y \wedge p(Y). \quad p(Z) \mathrel{:-} Z = 3. \quad p(Z) \mathrel{:-} Z = 5.$
The analysis determines the following properties: $\mathsf{ni\_set}(X) = 0, \mathsf{ni\_set}(Y) = 0, \mathsf{ni\_set}(Z) = 0, X \in \mathsf{int}(1,3), Y \in \mathsf{int}(3,5), Z \in \mathsf{int}(3,5)$. Without this information, a constraint such as $\mathtt{X} \neq \mathtt{Y}$ would be maintained in the $\mathcal{SET}$ part of the constraint, while, after the analysis, it is possible to pass it to $SAT_{\mathcal{FD}}$, and to use X and Y as arguments of a $\mathsf{labeling}$. Thus, this analysis helps in preparing the domain constraints to be sent to CLP($\mathcal{FD}$).

Application of the Static Analysis Framework. The results of the static analysis can generate new constraints. For all variables X such that $\mathsf{ni\_set}(X) = 0$ we add the constraints $\mathsf{integer}(X)$ and $X \in \mathtt{dom}(X)$.

Moreover, in some cases it is possible to directly compile a fragment of a CLP($\mathcal{SET}$) program P into CLP($\mathcal{FD}$). This can be done, for example, when for all term s in P, $\mathsf{ni\_set}(s) \in \{0,1\}$. Consider the following program, used to find subsets S of $\{1, \dots, n\}$ such that, for each pair of (not necessarily distinct) numbers in S, their sum is not in S (simplification of Shur numbers problem):

$$\begin{array}{l} \mathit{shur}(N, S) :- \\ S \subseteq \mathsf{int}(1, N) \ \wedge \\ (\forall X \in S)(\forall Y \in S)(\forall Z \in S)(Z \neq X + Y). \end{array}$$

The $\forall$ are used to encode RUQs (see Section 2) . The static analysis process is applied to the program obtained from the removal of the RUQs, and produces $\mathsf{ni\_set}(N) = \mathsf{ni\_set}(X) = \mathsf{ni\_set}(Z) = \mathsf{ni\_set}(T) = 0, \mathsf{ni\_set}(S) = 1$. These results can be used to directly derive a SICStus CLP($\mathcal{FD}$) program. The two programs have been executed for different values of N, using a PC, 1GHz, using SICStus Prolog 3.10. Precisely, in {log} (implementation of CLP($\mathcal{SET}$) [12]) the goal is `Q={A:shur(N,A)}` and in SICStus (execution of the program automatically derived from CLP($\mathcal{SET}$)$^{\mathsf{int}}$ using static analysis) `setof(A,shur(N,A),Q)`. In Figure 8 we report the running times.

N	5	6	7	8	16	24
{log}	11.7s	54s	3m.57s	∞	∞	∞
CLP($\mathcal{FD}$)	0.01s	0.02s	0.04s	0.08s	6.5s	3m.48s
Speedup	1200	2700	6000	∞	∞	∞

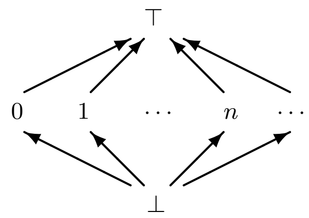


Fig. 2. Computational results and the Lattice $\mathcal{L}$

9 Conclusions

In this paper we presented an extension of the constraint logic programming language on hereditarily finite sets CLP($\mathcal{SET}$). The extension provides the ability to combine manipulation of domain variables (as in CLP($\mathcal{FD}$)) with the ability to manipulate the domain themselves, seen as finite sets. On one hand, this extends the expressive power of CLP($\mathcal{FD}$), allowing domains to play the role of first-class citizens of the language. On the other hand, the ability to automatically determine domain/domain variables allows the system to effectively distribute constraints between two constraints solvers (the one for CLP($\mathcal{SET}$) and the one for CLP($\mathcal{FD}$)), thus using the most efficient resolution mechanisms to solve each constraint. Static analysis mechanisms have been developed to support the execution of the extended constraint solving system. The extensions described in this paper have been implemented and will be soon available in the current distribution of the {log} system [12].

The work presented provides a number of new avenues for further generalizations. We plan to extend the results to sets of integer (or elements chosen from a finite universe) elements and to sets of sets of integer elements. These situations are precisely identified by the program analysis technique, when the outputs for a set s are $\mathsf{ni\_set}(s) = 1$ and $\mathsf{ni\_set}(s) = 2$, respectively. Several algorithms are based on operations on these two kinds of sets. For instance graphs related problems deal with sets of nodes and sets of sets of nodes (edges and/or partitions). Similarly, cooperations with constraint solvers on real numbers, as in $CLP(\mathcal{R})$, will be investigated.

Acknowledgements. We thank Sara Manzoli for useful discussions. The work is partially supported by MIUR project: *Automatic Aggregate—and number—Reasoning for Computing.*

References

1. P. Arenas-Sánchez and M. Rodríguez-Artalejo. A General Framework for Lazy Functional Logic, Programming with Algebraic Polymorphic Types. *Theory and Practice of Logic Programming*, 2(1):185–245, 2001.
2. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.
3. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *J. of Logic Programming*, 27:185–226, 1996.
4. A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *J. of Logic Programming*, 28(1):1–44, 1996.
5. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and Constraint Logic Programming. *ACM TOPLAS*, 22(5):861–931, 2000.
6. P. Flener, B. Hnich, and Z. Hiziltan. Compiling high-level type constructor in constraint programming. In I.V. Ramakrishnan, editor, *PADL2001*, LNCS 1990, pages 229–244. Springer-Verlag, Berlin, 2001.
7. C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1:191–246, 1997.
8. IC Parc. *The ECLiPSe Constraint Logic Programming System.* London. `www.icparc.ic.ac.uk/eclipse/`.
9. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *J. of Logic Programming*, 19–20:503–581, 1994.
10. B. Jayaraman. Implementation of Subset-Equational Programs. *J. of Logic Programming*, 12(4):299–324, 1992.
11. K. Marriott and P. J. Stuckey. *Programming with Constraints: an Introduction.* The MIT Press, Cambridge, Mass., 1998.
12. G. Rossi. *The {log} Constraint Logic Programming Language.* `http://prmat.math.unipr.it/~gianfr/setlog.Home.html`.
13. O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of Set Terms in the Logic Data Language (LDL). *J. of Logic Programming*, 12(1/2):89–119, 1992.
14. Swedish Institute of Computer Science. *The SICStus Prolog Home Page.* `www.sics.se`.
15. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
16. P. Van Hentenryck and Y. Deville. The Cardinality Operator: a New Logical Connective for Constraint Logic Programming. In K. Furukawa, editor, *International Conference on Logic Programming*, pages 745–759. MIT Press, 1991.
17. T. Yakhno and E. Petrov. Extensional Set Library for ECLiPSe. In D. Bjørner, M. Broj, and A. Zamulin, editors, *PSI'99*, LNCS 1755, pages 434–444. Springer-Verlag, Berlin, 2000.

Constructive Intensional Negation: a Practical Implementation (extended abstract)

Susana Muñoz, Julio Mariño, and Juan José Moreno-Navarro

Universidad Politécnica de Madrid [⋆]

Abstract. While negation in Logic Programming (LP) is a very active area of research, it was not included in the first Prolog systems. Amazingly, it is still true because there is no Prolog system implementing a sound and complete negation capability. One of the most promising techniques in the literature is intensional negation, following a transformational approach: for each positive predicate p its negative counterpart $intneg(p)$ is generated. However, from the practical point of view intensional negation cannot be considered a successful approach because no implementation is given. The reason is that neither universally quantified goals can be computed nor many practical issues are addressed. In this paper, we describe our efficient variant of the transformation of the intensional negation, called *Constructive Intensional Negation* providing some formal results as well as discussing a concrete implementation.

Keywords: Negation, Constraint Logic Programming, Program Transformation, Logic Programming Implementation, Constructive Negation.

1 Introduction

Kowalski and Colmerauer's decision on the elements of first-order logic supported in LP was based on the availability of implementation techniques and efficiency considerations. Among those important aspects not included from the beginning we can mention *evaluable functions*, *negation* and *higher order features*. All of them have revealed themselves as important for the expressiveness of Prolog as a programming language, but, while in the case of evaluable functions and higher order features considerable effort has been invested both in the semantic characterization and its efficient implementation we cannot say the same of negation. Many research papers do propose semantics to understand and incorporate negation into logic programming, but only a small subset of these ideas have their corresponding implementation counterpart. In fact, the negation capabilities incorporated by current Prolog compilers are rather limited, namely: the (unsound) negation as failure rule, and the sound (but incomplete) delay technique of the language Gödel [8], or Nu-Prolog [11] (having the risk of floundering.) The constructive negation of ECLiPSe [1], which was announced in earlier versions has been removed from recent releases due to implementation errors.

[⋆] Dpto. LSIIS – Facultad de Informática. Campus de Montegancedo s/n, 28660, Madrid, SPAIN. email:{`susana,jmarino,jjmoreno`}`@fi.upm.es`, voice: +34-91-336-7455, fax: +34-91-336-6595. This research was partly supported by the Spanish MCYT project TIC2000-1632.

The authors have been involved in a project [9, 10] to incorporate negation in a real Prolog system (up to now Ciao [5], but the techniques are easily applicable to any other system). This allows us to keep the very advanced Prolog current implementations based on WAM technology as well as to reuse thousands of Prolog code lines. The basis of our work is to combine and refine existing techniques to make them useful for practical application. Furthermore, we try to use the simplest technique as possible in any particular case. To help on this distinction, we need to use some tests to characterize the situation for efficiency reasons. To avoid the execution of dynamic tests, we suggest to use the results of a global analysis of the source code. For instance, the primary technique is the built-in *negation as failure* that can be used if a groundness analysis tells that every negative literal will be ground at call time [10].

In order to handle non-ground literals, a number of alternatives to the negation-as-failure rule have been proposed under the generic name of *constructive negation*: Chan's *constructive negation* [6, 7, 13], *intensional negation* [2–4], fail substitutions, fail answers, etc. From a theoretical viewpoint Chan's approach is enough but it is quite difficult to implement and expensive in terms of execution resources. On the other hand, intensional negation uses a transformational approach, so most of the work is performed at compile time and then a variant of the standard SLD resolution is used to execute the resulting program, so a significant gain in efficiency is expected. There are, however, some problems when transforming certain classes of programs.

In this paper we concentrate on the study of the efficient implementation of intensional negation. As it is formulated in the original papers, it is not possible to derive an efficient transformation. On one hand, universally quantified goals generated are hard to be managed. On the other hand the operational behavior of the original program is modified computing infinitely many answers instead of compact results. Furthermore, many significant details were missing. We propose a way of implementing the universal quantification. It is based on the properties of the domain instead of the code of the program as in [4].

The rest of the paper is organized as follows. Section 2 introduces basic syntactic and semantic concepts needed to understand our method. Section 3 formally presents the transformation algorithm of the Constructive Intensional Negation technique. Section 4 discuss our universal quantification definition and implementation. Finally, we conclude and discuss some future work (Section 5). Due to space limitations we do not provide details of the proofs, that will appear in the full version of the paper.

2 Preliminaries

In this section the syntax of (constraint) logic programs and the intended notion of correctness are introduced. Programs will be constructed from a signature $\Sigma = \langle FS_\Sigma, PS_\Sigma \rangle$ of function and predicate symbols. Provided a numerable set of variables V the set $Term(FS_\Sigma, V)$ of terms is constructed in the usual way.

A *constraint* is a first-order formula whose atoms are taken from $Term(FS_\Sigma, V)$ and where the only predicate symbol is the binary equality operator $=_{/2}$. A formula $\neg(t_1 = t_2)$ will be abbreviated $t_1 \neq t_2$. The constants $\underline{\mathrm{t}}$ and $\underline{\mathrm{f}}$ will denote the neutral elements of conjunction and disjunction, respectively. A tuple $(x_1,\ldots,x_n)$ will be abbreviated by $\bar{x}$. The concatenation of tuples $\bar{x}$ and $\bar{y}$ is denoted $\bar{x}\cdot\bar{y}$.

A (constrained) Horn clause is a formula $h(\bar{x}) \leftarrow b_1(\bar{y}\cdot\bar{z}),\ldots,b_n(\bar{y}\cdot\bar{z}) [\!] c(\bar{x}\cdot\bar{y})$ where $\bar{x},\bar{y}$ and $\bar{z}$ are tuples from disjoint sets of variables[1]. The variables in $\bar{z}$ are called the *free* variables in the clause. The symbols "," and "$[\!]$" act here as aliases of logical conjunction. The atom to the left of the symbol "$\leftarrow$" is called the *head* or *left hand side* (lhs) of the clause. *Generalized* Horn clauses of the form $h(\bar{x}) \leftarrow B(\bar{y}\cdot\bar{z}) [\!] c(\bar{x}\cdot\bar{y})$ where the body B can have arbitrary disjunctions, denoted by ";", and conjunctions of atoms will be allowed as they can be easily translated into "traditional" ones.

A Prolog program (in Σ) is a set of clauses indexed by $p \in PS_\Sigma$:

$$\begin{gathered} p(\bar{x}) \leftarrow B_1(\bar{y}_1\cdot\bar{z}_1) [\!] c_1(\bar{x}\cdot\bar{y}_1) \\ \vdots \\ p(\bar{x}) \leftarrow B_m(\bar{y}_m\cdot\bar{z}_m) [\!] c_m(\bar{x}\cdot\bar{y}_m) \end{gathered}$$

The set of defining clauses for predicate symbol p in program P is denoted $def_P(p)$. Without loss of generality we have assumed that the left hand sides in $def_P(p)$ are syntactically identical. Actual Prolog programs, however, will be written using a more traditional syntax.

Assuming the normal form, let $def_P(p) = \{p(\bar{x}) \leftarrow B_i(\bar{y}_i\cdot\bar{z}_i) [\!] c_i(\bar{x}\cdot\bar{y}_i) | i \in 1\ldots m\}$. The *completed definition* of p, $cdef_P(p)$ is defined as the formula

$$\forall\bar{x}.\left[p(\bar{x}) \iff \bigvee_{i=1}^{m} \exists\bar{y}_i.\ c_i(\bar{x}\cdot\bar{y}_i) \wedge \exists\bar{z}_i.B_i(\bar{y}_i\cdot\bar{z}_i)\right]$$

The *Clark's completion* of the program is the conjunction of the completed definitions for all the predicate symbols in the program along with the formulas that establish the standard interpretation for the equality symbol, the so called *Clark's Equality Theory* or *CET*. The completion of program P will be denoted $Comp(P)$. Throughout the paper, the standard meaning of logic programs will be given by the three-valued interpretation of their completion – i.e. its minimum 3-valued model. These 3 values will be denoted $\underline{t}$ (or `success`), $\underline{f}$ (or `fail`) and $\underline{u}$ (or `unknown`).

Example 1. Let us start with a sample program to compute whether a number is even. It can be expressed as a set of normalized Horn clauses in the following way:

```
even(X) ← [] X=0.
even(X) ← even(N) [] X=s(s(N)).
```

Its completion is $CET \wedge \forall x.\left[even(x) \iff x = 0 \vee \exists n.x = s(s(n)) \wedge even(n)\right]$.

We are now able to specify intensional negation formally. Given a signature $\Sigma = \langle FS_\Sigma, PS_\Sigma\rangle$, let $PS'_\Sigma \supset PS_\Sigma$ be such that for every $p \in PS_\Sigma$ there exists a symbol $neg(p) \in PS'_\Sigma \setminus PS_\Sigma$. Let $\Sigma' = \langle FS_\Sigma, PS'_\Sigma\rangle$.

Definition 1 (Intensional Negation of a Program). *Given a program P_Σ, its* intensional negation *is a program $P'_{\Sigma'}$ such that for every p in PS_Σ the following holds:*

$$\forall\bar{x}.\left[Comp(P) \models_3 p(\bar{x}) \iff Comp(P') \models_3 \neg(neg(p)(\bar{x}))\right]$$

[1] The notation $p(\bar{x})$ expresses that $Vars(p(\bar{x})) \in \bar{x}$, not that it is identical to $p(x_1,\ldots,x_n)$

3 Toward a Better Transformation

Some of the problems with the previous methods [2–4] are due to the syntactical complexity of the negated program. A simplified translation can be obtained by taking into account some properties that are often satisfied by the source program. For instance, it is usually the case that the clauses for a given predicate are mutually exclusive. The following definition formalizes this idea:

Definition 2. *A pair of constraints c_1 and c_2 is said to be* incompatible *iff their conjunction $c_1 \wedge c_2$ is unsatisfiable. A set of constraints $\{c_i\}$ is* exhaustive *iff $\bigvee_i c_i = \underline{t}$. A predicate definition $def_P(p) = \{p(\bar{x}) \leftarrow B_i(\bar{y}_i \cdot \bar{z}_i)[\!]c_i(\bar{x} \cdot \bar{y}_i) | i \in 1 \ldots m\}$ is* nonoverlapping *iff $\forall i, j \in 1 \ldots m$ the constraints $\exists \bar{y}_i. c_i(\bar{x} \cdot \bar{y}_i)$ and $\exists \bar{y}_j. c_j(\bar{x} \cdot \bar{y}_j)$ are incompatible and it is exhaustive if the set of constraints $\{\exists \bar{y}_i. c_i(\bar{x} \cdot \bar{y}_i)\}$ is exhaustive.*

In the following, the symbols "$\twoheadrightarrow$" and "$\|$" will be used as syntactic sugar for "$\wedge$" and "$\vee$" respectively, when writing completed definitions from nonoverlapping and exhaustive sets of clauses, remarking their behaviour as a *case*-like construct. A nonoverlapping set of clauses can be made into a nonoverlapping and exhaustive one by adding an extra "default" case:

Lemma 1. *Let p be such that its definition $def_P(p) = \{p(\bar{x}) \leftarrow B_i(\bar{y}_i \cdot \bar{z}_i)[\!]c_i(\bar{x} \cdot \bar{y}_i) | i \in 1 \ldots m\}$ is nonoverlapping. Then its completed definition is logically equivalent to*

$$\begin{aligned} cdef_P(p) \equiv \forall \bar{x}.[p(\bar{x}) \iff & \exists \bar{y}_1.\ c_1(\bar{x} \cdot \bar{y}_1) \twoheadrightarrow \exists \bar{z}_1.B_1(\bar{y}_1 \cdot \bar{z}_1) \ \| \\ & \vdots \\ & \exists \bar{y}_m.\ c_m(\bar{x} \cdot \bar{y}_m) \twoheadrightarrow \exists \bar{z}_m.B_m(\bar{y}_m \cdot \bar{z}_m) \ \| \\ & \textstyle\bigwedge_{i=1}^{m} \neg \exists \bar{y}_i, \bar{z}_i.\ c_i(\bar{x} \cdot \bar{y}_i) \twoheadrightarrow \underline{f}] \end{aligned}$$

The interesting fact about this kind of definitions is captured by the following lemma:

Lemma 2. *Given first-order formulas $\{C_1, \ldots, C_n\}$ and $\{B_1, \ldots, B_n\}$ the following holds:*

$$\neg[\exists \bar{y}_1(C_1 \twoheadrightarrow B_1) \ \| \ldots \| \ \exists \bar{y}_n(C_n \twoheadrightarrow B_n)] \iff \exists \bar{y}_1(C_1 \twoheadrightarrow \neg B_1) \ \| \ldots \| \ \exists \bar{y}_n(C_n \twoheadrightarrow \neg B_n)$$

This means that the overall structure of the formula is preserved after negation, which seems rather promising for our purposes.

The idea of the transformation to be defined below is to obtain a program whose completion corresponds to the negation of the original program, in particular to a representation of the completion where negative literals have been eliminated. This can be formalized as the successive application of several transformation steps:

1. For every completed definition of a predicate p in PS_Σ, $\forall \bar{x}. [p(\bar{x}) \iff D]$, add the completed definition of its negated predicate, $\forall \bar{x}. [neg\_p(\bar{x}) \iff \neg D]$.
2. Move negations to the right of "$\twoheadrightarrow$" using lemma 2.
3. If negation is applied to a existential quantification, replace $\neg \exists \bar{z}.C$ by $\forall \bar{z}. \neg C$.
4. Replace $\neg C$ by its *negated normal form* $NNF(\neg C)$ [2].
5. Replace $\neg \underline{t}$ by $\underline{f}$, $\neg \underline{f}$ by $\underline{t}$, for every predicate symbol p, $\neg p(\bar{t})$ by $neg\_p(\bar{t})$.

[2] The *negated normal form* of C, is obtained by successive application of the De Morgan laws until negations only affect atomic subformulas of C

3.1 The Transformation

Definition 3. *The syntactic transformation negate_rhs is defined as follows:*

$$\begin{aligned}
negate\_rhs(P;Q) &= negate\_rhs(P), negate\_rhs(Q)\\
negate\_rhs(P,Q) &= negate\_rhs(P); negate\_rhs(Q)\\
negate\_rhs(\underline{t}) &= \underline{f}\\
negate\_rhs(\underline{f}) &= \underline{t}\\
negate\_rhs(p(\bar{t})) &= neg\_p(\bar{t})
\end{aligned}$$

Definition 4 (Constructive Intensional Negation). *For every predicate p in the original program, assuming $def_P(p) = \{p(\bar{x}) \leftarrow B_i(\bar{y}_i \cdot \bar{z}_i) [\![c_i(\bar{x} \cdot \bar{y}_i) | i \in 1 \ldots m\}$, the following clauses will be added to the negated program:*

– *If the set of constraints $\{\exists \bar{y}_i . c_i(\bar{x}.\bar{y}_i)\}$ is not exhaustive, a clause*

$$neg\_p(\bar{x}) \leftarrow [\![\bigwedge_{1}^{m} \neg \exists \bar{y}_i . c_i(\bar{x}.\bar{y}_i)$$

– *If $\bar{z}_j$ is empty, the clause*

$$neg\_p(\bar{x}) \leftarrow negate\_rhs(B_j(\bar{y}_j)) [\![\exists \bar{y}_j . c_j(\bar{x}.\bar{y}_j)$$

– *If $\bar{z}_j$ is not empty, the clauses*

$$neg\_p(\bar{x}) \leftarrow forall([\bar{z}_j], p\_j(\bar{y}_j \cdot \bar{z}_j)) [\![\exists \bar{y}_j . c_j(\bar{x}.\bar{y}_j)$$

$$p\_j(\bar{y}_j \cdot \bar{z}_j) \leftarrow negate\_rhs(B_j(\bar{y}_j \cdot \bar{z}_j))$$

We can see that negating a clause with free variables introduces "universally quantified" goals by means of a new predicate `forall/2` that will be discussed and proved correct in sections 4 and 4.1. In absence of free variables the transformation is trivially correct as the completion of the negated predicate corresponds to the negation-free normal form described above.

Theorem 1. *The result of applying the* Constructive Intensional Negation *transformation to a (nonoverlapping) program P is an intensional negation (as defined in def. 1).*

3.2 Overlapping Definitions

When the definition of some predicate has overlapping clauses the simplicity of the transformation above is lost. Rather than defining a different scheme for overlapping rules we will give a translation of general sets of clauses into nonoverlapping ones:

Lemma 3. *Let p be such that $def_P(p) = \{p(\bar{x}) \leftarrow B_i(\bar{y}_i \cdot \bar{z}_i) [\![c_i(\bar{x} \cdot \bar{y}_i) | i \in 1 \ldots m\}$ and there exist $j, k \in 1 \ldots m$ such that $\exists \bar{y}_j . c_j(\bar{x} \cdot \bar{y}_j)$ and $\exists \bar{y}_k . c_k(\bar{x} \cdot \bar{y}_k)$ are compatible. Then the j-th and k-th clauses can be replaced by the clause*

$$p(\bar{x}) \leftarrow B_j(\bar{y}_j \cdot \bar{z}_j); B_k(\bar{y}_k \cdot \bar{z}_k) [\![c_j(\bar{x} \cdot \bar{y}_j) \wedge c_k(\bar{x} \cdot \bar{y}_k)$$

and the following additional clauses (in case the new constraints were not equivalent to $\underline{\mathrm{f}}$*):*

$$p(\overline{x}) \leftarrow B_j(\overline{y}_j \cdot \overline{z}_j) \mathbin{[\![} c_j(\overline{x}\cdot\overline{y}_j) \wedge \neg c_k(\overline{x}\cdot\overline{y}_k)$$
$$p(\overline{x}) \leftarrow B_k(\overline{y}_k \cdot \overline{z}_k) \mathbin{[\![} \neg c_j(\overline{x}\cdot\overline{y}_j) \wedge c_k(\overline{x}\cdot\overline{y}_k)$$

without changing the standard meaning of the program. The process can be repeated if there are more than two overlapping clauses. It is clear that it is a finite process.

3.3 Examples

Let us show some motivating examples:

Example 2. The predicate `lesser/2` can be defined by the following nonoverlapping set of clauses

```
lesser(0,s(Y))
lesser(s(X),s(Y)) ← lesser(X,Y)
```

or, normalizing,

```
lesser(N,M) ← [] N=0, M=s(Y)
lesser(N,M) ← lesser(X,Y) [] N=s(X), M=s(Y)
```

By lemma 1, the completed definition of `lesser` is

$$\begin{aligned} cdef_P(lesser) \equiv \forall n,m.\, lesser(n,m) \iff & \exists y. n = 0 \wedge m = s(y) \twoheadrightarrow \underline{\mathrm{t}} \parallel \\ & \exists x,y. n = s(x) \wedge m = s(y) \twoheadrightarrow lesser(x,y) \parallel \\ & m = 0 \twoheadrightarrow \underline{\mathrm{f}} \end{aligned}$$

We have assumed that the constraint $\neg(\exists y.n = 0 \wedge m = s(y)) \wedge \neg(\exists x,y.n = s(x) \wedge m = s(y))$ has been simplified to $m = 0$ (if the only symbols are 0 and $s/1$).

And the generated Prolog program for `neg_lesser` is:

```
neg_lesser(N,M) ← forall([Z], (X,Y) =/= (0,s(Z))),
                  forall([V,W], (X,Y) =/= (s(V),s(W)) [] N=X, M=Y
neg_lesser(N,M) ← neg_lesser(X,Y) [] N=s(X), M=(Y)
```

where `forall/2` implements the universal quantification (see section 4) and `=/= /2` is the disequation constraint between terms.

Example 3. The second example, *family*, is also well-known, and includes free variables in the rhs of the definition of the predicate `ancestor/2`:

```
parent(john, mary)     ancestor(X, Y) ← parent(X, Y)
parent(john, peter)    ancestor(X, Y) ← parent(Z, Y) ∧
parent(mary, joe)                       ancestor(X, Z)
parent(peter, susan)
```

or, normalizing,

```
ancestor(N,M) ← parent(X,Y) ; (parent(Z,Y), ancestor(Y,X)) [] N=X, M=Y
```

The corresponding Prolog program for `neg_ancestor` is:

```
neg_ancestor(N,M) ← neg_parent(X,Y),
               forall([Z],neg_parent(Z,Y); neg_ancestor(Y,X)) [] N=X, M=Y
```

The *Compilative Constructive Negation* of [4] follows the ideas of constructive negation but introducing constraints for predicate lhs and therefore for its negation. The authors provide denotational semantics in terms of fixpoint of adequate immediate consequences operators as well as a description of the operational semantics, called SLD$^\forall$. Again, the resulting program should contain universally quantified goals. The problem of this resolution is that the satisfiability of these quantification is checked for the clauses of the goal and it is not so "compilative" as we would like. As the authors are mainly concerned with formal semantics, the resulting program may contain useless clauses and by no means is optimal neither in the size of the program nor in the number of solutions. Let us describe our own novel method to handle universally quantified goals in the next section.

4 Universal Quantification

We have achieved a compilative implementation of the universal quantification working over the Herbrand Universe instead over the definition of the program. Our implementation of universal quantification is based on two ideas:

1. A universal quantification of the goal Q over a variable X succeeds when Q succeeds without binding (or constraining) X.
2. A universal quantification of Q over X is true if Q is true for all possible values for the variable X.

This can be expressed formally in this way:

$$\forall X.\ Q(X) \equiv Q(sk) \vee [\forall \overline{X_1}.\ Q(c_1(\overline{X_1})) \wedge \cdots \wedge \forall \overline{X_n}.\ Q(c_n(\overline{X_n}))]$$

where $\{c_1 \dots c_n\} = FS$ and sk is a Skolem constant, that is, $sk \notin FS$. In practice, the algorithm proceeds by trying the Skolem case first and, if it fails, the variable expansion. The following definitions try to capture some delicate details of the procedure, necessary to ensure its completeness. In order to reduce the complexity of the presentation, in the following we will only consider quantifications of the form $\forall x.Q$ where Q has neither quantifiers nor free variables.

Definition 5 (Covering). *A* covering *of $Term(FS,V)$ is any finite sequence of terms $\langle t_1 \dots t_n \rangle$ such that:*

– *For every i, j with $i \neq j$, t_i and t_j do not superpose, i.e. there is no ground substitution σ with $t_i\sigma = t_j\sigma$.*

– *For every ground term s of the Herbrand Universe there exists i and a ground substitution σ with $s = t_i\sigma$.*

The simplest covering is a variable $C_1 = \{X\}$. E. g., if a program uses only natural numbers $C_2 = \{0, s(X)\}$, $C_3 = \{0, s(0), s(s(X))\}$ and $C_4 = \{0, s(0), s(s(0)), s(s(s(X)))\}$ are all coverings. This example also suggests how to generate coverings incrementally. We start from the simplest covering X. From one covering we generate the next one choosing one term and one variable of this term. The term is removed and then we add all the terms obtained replacing the variable by all the possible instances of that element.

Definition 6 (Next Covering). *Given a covering $C = \langle t_1, \dots, t_m \rangle$, the* next covering *to C over the variable X is defined as $next(C,X) = \langle t_1, \dots, t_{j-1}, t_{j+1}, \dots, t_m, t_{j1}, \dots, t_{jn} \rangle$ where each $t_{jk} = t_j\sigma_k$ is obtained from the symbols in FS by applying, for each $c_k \in FS$ the substitution $\sigma_k = [X \mapsto c_k(\overline{X_k})]$ where $\overline{X_k} \cap Vars(t_j) = \emptyset$. We can say that a covering C is less instantiated that $next(C,X)$ in the sense of [12].*

Definition 7 (Sequence of Coverings). $S = \langle C_1, \dots, C_n \rangle$ *is a* sequence of coverings *if $C_1 = \langle X \rangle$, for some variable X and for every $i \in \{1 \dots n-1\}$, $C_{i+1} = next(C_i, X_i)$, where X_i is the variable appearing in C_i at the leftmost position.*

In order to use coverings as successive approximations of a universal quantification it is necessary to replace their variables by Skolem constants:

Definition 8 (Evaluation of a Covering). *Given a covering C, we define $skolem(C)$ as the result of replacing every variable by a different Skolem constant. The* evaluation *of C for the quantified goal $\forall X.Q$ is the three-valued conjunction of the results of evaluating Q with all the elements in $skolem(C)$. That is*

$$eval(C, \forall X.Q) = Q[X \mapsto t_1] \wedge \cdots \wedge Q[X \mapsto t_m]$$

where $skolem(C) = \langle t_1 \dots t_m \rangle$.

After the evaluation of a covering C_i for a given quantified goal, several situations may arise:

1. $eval(C_i, \forall X.Q) = \underline{t}$. We can infer that the universal quantification of Q succeeds.
2. $eval(C_i, \forall X.Q) = \underline{u}$. We can infer that the universal quantification of Q loops.
3. $eval(C_i, \forall X.Q) = \underline{f}$. According to the three-valued semantics of conjunction this will happen whenever there exists some t_j in $skolem(C_i)$ such that $Q[X \mapsto t_j] \leadsto \underline{f}$. We can consider two subcases:
   – There is some t_j in $skolem(C_i)$ which does not contain Skolem constants such that $Q[X \mapsto t_j] \leadsto \underline{f}$. We can infer that the universal quantification of Q fails.
   – Otherwise, for every t_j in $skolem(C_i)$ such that $Q[X \mapsto t_j] \leadsto \underline{f}$, t_j contains Skolem constants. We cannot infer that the universal quantification of Q fails, and $eval(C_{i+1}, \forall X.Q)$ must be considered. We will say that C_i is *undetermined* for $\forall X.Q$.

During this process, nontermination may arise either during the evaluation of one covering or because the growth of the sequence of coverings does not end. The following lemmata show that this does not affect the completeness of the method:

Lemma 4. *Let $S = \langle C_1, C_2, \ldots \rangle$ be a sequence of coverings. If there is any C_i in S such that $eval(C_i, \forall X.Q) = \underline{u}$, then $eval(C_{i+1}, \forall X.Q) = \underline{u}$.*

Lemma 5. *Let $S = \langle C_1, C_2, \ldots \rangle$ be an infinite sequence of coverings such that for every $i > 0$ C_i is undetermined for $\forall X.Q$. Then it can be proved that, under the three-valued semantics of the program the quantification $\forall X.Q$ is undefined.*

That is, loops introduced during the computation of coverings are consistent with the semantics of universal quantification. Finally:

Theorem 2 (Correctness). *When the procedure above provides a result ($\underline{t}$ or $\underline{f}$) for $\forall X.Q$ this is the result that would be obtained from the application of Fitting's 3-valued operator to the computation of the universal quantification.*

4.1 Implementation Issues

Disequality Constraints An instrumental step in order to manage negation in a more advanced way is to be able to handle disequalities between terms such as $t_1 \neq t_2$. Prolog implementations typically include only the built-in predicate `\== /2` which can only work with disequalities if both terms are ground and simply succeeds in the presence of free variables. A "constructive" behavior must allow the "binding" of a variable with a disequality. On the other hand, the negation of an equation $X = t(\overline{Y})$ produces the universal quantification of the free variables in the equation, unless a more external quantification affects them. The negation of such an equation is $\forall\ \overline{Y}.\ X \neq t(\overline{Y})$. As we explained in [9], the inclusion of disequalities and constrained answers has a very low cost as a direct consequence of [6, 7, 13]. It incorporates negative normal form constraints instead of bindings and the decomposition step can produce disjunctions.

Implementation of Universal Quantification We implement the universal quantification by means of the predicate $forall([X1, \ldots, Xn],\ Q)$, where $X1, \ldots, Xn$ are the universal quantified variables and Q is the goal to quantify. We start with the initial covering $\{(X_1, \ldots, X_n)\}$ of depth 1.

There are two important details that optimize the execution. The first one is that in order to check if there are bindings in the covering variables, it is better to replace them by new constants that do not appear in the program. In other words, we are using "Skolem constants".

The second optimization is much more useful. Notice that the coverings grow up incrementally, so we only need to check the most recently included terms. The other ones have been checked before and there is no reason to do it again.

As an example, consider a program which uses only natural numbers: the sequence of coverings for the goal $\forall\ X, Y, Z.\ p(X, Y, Z)$ will be the following (where Sk_i, with i a number, represents the ith Skolem constant).

$C_1 = [(Sk_1, Sk_2, Sk_3)]$
$C_2 = [\underline{(0, Sk_1, Sk_2)}, \underline{(s(Sk_1), Sk_2, Sk_3)}]$
$C_3 = [\underline{(0, 0, Sk_1)}, \underline{(0, s(Sk_1), Sk_2)}, (s(Sk_1), Sk_2, Sk_3)]$

$C_4 = [\underline{(0,0,0)}, \underline{(0,0,s(Sk_1))}, (0,s(Sk_1),Sk_2), (s(Sk_1),Sk_2,Sk_3)]$
$C_5 = [(0,0,0), \underline{(0,0,s(0))}, \underline{(0,0,s(s(Sk_1)))}, (0,s(Sk_1),Sk_2),$
$\quad (s(Sk_1),Sk_2,Sk_3)]$
$C_6 = \ldots$

In each step, only two elements need to be checked, those that appear underlined. The rest are part of the previous covering and they do not need to be checked again.

Let us show some examples of the use of the *forall* predicate, indicating the covering found to get the solution. We are still working only with natural numbers:

```
| ?- forall([X], even(X)).
     no
```

with the covering of depth 3 $\{0, \underline{s(0)}, s(s(Sk_1))\}$.

```
| ?- forall([X], X =/= a).
     yes
```

with the covering of depth 1 $\{Sk_1\}$.

```
| ?- forall([X], less(0, X) -> less(X, Y)).
     Y = s(s(_A))
```

with the covering of depth 2 $\{0, s(Sk_1)\}$.

```
| ?- forall([X], (Y=X ; lesser(Y, X))).
     Y = 0
```

with the covering of depth 2 $\{0, s(Sk_1)\}$.

As we have seen in the definition of the quantifier, this implementation is correct and if we use a fair selection rule to chose the order in which the terms of the covering are visited then it is also complete. We can implement this because in Ciao Prolog is possible to ensures AND-fairness by goal shuffling [5].

If we implement this in a Prolog compiler using depth first SLD-resolution, the selection will make the process incomplete. When we are evaluating a covering $C = \{t_1, ..., t_m\}$ for a goal $Q(X)$ if we use the depth first strategy from the left to the right, we can find that the evaluation of a $Q(t_j)$ is unknown and is there exists one $k > j$ such that t_k is ground and $Q(t_k)$ fails, then we would obtain that $forall([X], Q(X))$ is unknown when indeed it is false.

Experimental Results Let us show some experimental results very encouraging. Table 1 includes some measurements of execution times to get the first solution of a list of positive goals, their negations as failure, their negation using our implementation of the constructive classical negation of Chan, and their negation using our implementation of the constructive intensional negation. There are represented the ratios of the constructive negation and the intensional negation w.r.t. the positive goal and the ratio of the constructive negation w.r.t. the intensional negation.

We present three set of examples. The first one collects some examples where it is slightly worst or a little better to use intensional negation instead of negation as failure.

goals - G	**G**	**naf(G)**	**cneg(G)**	ratio	**intneg(G)**	ratio	**cneg/intneg**
boole(1)	780	829	1319	1.69	780	1.00	1.69
positive(500)	1450	1810	2090	1.44	3430	2.36	0.60
positive(700)	1450	1810	2099	1.44	5199	3.58	0.40
positive(1000)	2070	2370	2099	1.01	8979	4.33	0.23
less(0,50000)	1189	1199	23209	19.51	6520	5.48	3.55
less(50000,0)	1179	1140	4819	4.08	10630	9.01	0.45
average				**4.86**		**4.29**	**1.15**
boole(X)	829	-	1340	1.61	830	1.00	1.61
ancestor(peter,X)	820	-	1350	1.64	821	1.00	1.64
average				**1.63**		**1.00**	**1.62**
less(50000,X)	1430	-	28159	19.69	6789	4.74	4.14
less(100000,X)	1930	-	54760	28.37	13190	6.83	4.15
less(X,50000)	1209	-	51480	42.58	12210	10.09	4.21
less(X,100000)	1580	-	102550	64.90	24099	15.25	4.25
add(X,Y,100000)	2219	-	25109	11.31	4209	1.81	5.96
add(100000,Y,Z)	2360	-	12110	5.10	2659	1.12	4.55
add(X,100000,Z)	2160	-	23359	10.81	2489	1.15	9.38
average				**26.10**		**5.86**	**5.23**

Table 1. Runtime comparison

The second set contains examples in which negation as failure cannot be used. Intensional negation is very efficient in the sense that the execution time is similar as the time needed to execute the positive goal.

The third set of examples is the most interesting because contains more complex goals where negation as failure cannot be used and the speed-up of intensional negation over constructive negation is over 5 times better.

5 Conclusion and Future Work

Intensional Negation is just a part of a more general project to implement negation where several other approaches are used: negation as failure possibly combined with dynamical goal reordering, intensional negation, and constructive negation. The decision of what approach can be used is fixed by the information of different program analyses: naf in case of groundness of statically detected goal reordering, finite constructive negation in case of finiteness of the number of solutions, etc. See [10] for details. Notice that the strategy also ensures the soundness of the method: if the analysis is correct, the precondition to apply a technique is ensured, so the results are sound.

Prolog can be more widely used for knowledge representation based applications if negated goals can be included in programs. We have presented a collection of techniques [9], more or less well-known in logic programming, that can be used together in order to produce a system that can handle negation efficiently. In our approach, intensional negation plays a significant role and a deeper study of it have been presented.

To our knowledge it is one of the first serious attempts to include such proposals as implementations into a Prolog compiler.

The main novelties of our constructive intensional negation approach are:

– The formulation in terms of disequality constraints, solving some of the problems of intensional negation [3] in a more efficient way than [4] because of the definition of the universal quantification without using the program code.
– The computation of universally quantified goals, that was sketched in [3], and we provide here an implementation and a discussion about its soundness and completeness.

As a future work, we plan to include this implementation into a compiler in order to produce a version of Ciao Prolog with negation. Our implementation of constructive intensional negation can be generalized to other simple Prolog Systems with the attributed variables extension that is the only requirement for implementing disequality constraints.

Another field of work is the optimization of the implementation of the *forall* using different search techniques and more specialized ways of generating the coverings of the Herbrand Universe of our negation subsystem.

References

1. The ECLiPSe website at Imperial College. http://www-icparc.doc.ic.ac.uk/eclipse/.
2. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional negation of logic programs. *Lecture notes on Computer Science*, 250:96–110, 1987.
3. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP*, 8(3):201–228, 1990.
4. P. Bruscoli, F. Levi, G. Levi, and M.C. Meo. Compilative Constructive Negation in Constraint Logic Programs. In Sophie Tyson, editor, *Proc. of the Nineteenth International Colloquium on Trees in Algebra and Programming, CAAP '94*, volume 787 of *LNCS*, pages 52–67, Berlin, 1994. Springer-Verlag.
5. F. Bueno. *The CIAO Multiparadigm Compiler: A User's Manual*, 1995.
6. D. Chan. Constructive negation based on the complete database. In *Proc. Int. Conference on LP'88*, pages 111–125. The MIT Press, 1988.
7. D. Chan. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89*, pages 477–493. The MIT Press, 1989.
8. P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.
9. S. Muñoz and J.J. Moreno. How to incorporate negation in a prolog compiler. In V. Santos Costa E. Pontelli, editor, *2nd International Workshop PADL'2000*, volume 1753 of *LNCS*, pages 124–140, Boston, MA (USA), 2000. Springer.
10. S. Muñoz, J.J. Moreno, and M. Hermenegildo. Efficient negation using abstract interpretation. In R.Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning*, La Habana (Cuba), 2001.
11. L. Naish. Negation and quantifiers in NU-Prolog. In *Proc. 3rd ICLP*, 1986.
12. A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as instantiation. *Information and Computation*, 120(2):263–278, 1995.
13. P. Stuckey. Negation and constraint logic programming. In *Information and Computation*, volume 118, pages 12–33, 1995.

A Demand Narrowing Calculus with Overlapping Definitional Trees

Rafael del Vado Vírseda

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Av. Complutense s/n, 28040 Madrid, Spain
rdelvado@sip.ucm.es

Abstract. We propose a *demand conditional narrowing calculus* in which a variant of *definitional trees* [2] is used to efficiently control the narrowing strategy. This calculus is sound and strongly complete w.r.t. *Constructor-based ReWriting Logic* (CRWL) semantics [7] for a wide class of constructor-based conditional term rewriting systems. The calculus maintains the optimality properties of the needed narrowing strategy [5]. Moreover, the treatment of *strict equality* as a primitive rather than a defined function symbol, leads to an improved behaviour w.r.t. needed narrowing.

1 Introduction

Many recent approaches to the integration of functional and logic programming take *constructor-based conditional rewrite systems* as programs and some *lazy narrowing strategy* or *calculus* as a goal solving mechanism and as operational semantics (see [8] for a detailed survey). On the other hand, *non-deterministic* functions are an essential feature of these integrated languages, since they allow problem solving using programs that are textually shorter, easier to understand and maintain, and more declarative than their deterministic counterparts (see [3], [7] or [13] for several motivating examples).

In this context, efficiency has been one of the major drawbacks associated to the functional logic programming paradigm, where the introduction of non-deterministic computations often generates huge search spaces with their associated overheads both in terms of time and space. Actually, the adoption of a *demand-driven* narrowing strategy can result in a large reduction of the search space [15]. The demand-driven strategy evaluates an argument of a function only if its value is needed to compute the result. An appropriate notion of *need* is a subtle point in the presence of a non-deterministic choice, e.g., a typical computation step, since the need of an argument might depend on the choice itself. For the case of rewriting, an adequate theory of neednedness has been proposed by Huet and Lévy [12] for orthogonal and unconditional rewrite systems and extended by Middeldorp [16] to the more general case of neednedness for the computation of root-stable forms. The so-called *needed narrowing* strategy [5] has been designed for *Inductively Sequential Systems* (ISS), a proper subclass of orthogonal rewriting systems which coincides with the class of strongly sequential constructor-based orthogonal TRSs [11] and defines only deterministic functions by means of unconditional rewrite rules.

Both needed narrowing and the demand driven strategy from [15] were originally defined with the help of *definitional trees*, a tool introduced by Antoy [2] for achieving a reduction strategy which avoids unneeded reductions. The needed narrowing strategy is sound and complete for the class ISS, and it enjoys interesting and useful optimality properties, but has the disadvantage that it refers only to a closed version of strict equality [5] where the goals are often solved by enumerating infinitely many ground solutions, instead of computing a single more general one.

Example 1. Consider the following ISS, and let $s^k(t)$ be result of applying the constructor *s*, k consecutives times, to the term *t*.

$$\begin{aligned} X + 0 &\to X \\ X + s(Y) &\to s(X + Y) \end{aligned}$$

For the goal X + Y == Z, needed narrowing as presented in [5] enumerates infinitely many ground solutions $\{Y \mapsto s(0), X \mapsto s^n(0), Z \mapsto s^{n+1}(0)\}$ (for all $n \geq 0$) instead of computing an open solution $\{Y \mapsto s(0), Z \mapsto s(X)\}$ which subsumes the previous ground ones. The reason of this behaviour is the formal treatment of == as a function defined by rewrite rules. In this example:

$$\begin{aligned} 0 == 0 &\to \text{true} \qquad\qquad \text{true}, Z \to Z \\ s(X) == s(Y) &\to X == Y \end{aligned}$$

□

A more recent paper [3] has proposed an extension of needed narrowing called *inductively sequential narrowing*. This strategy is sound and complete for *overlapping inductively sequential systems*, a proper extension of the class of ISSs which permits non-deterministic functions, but conditional rules and an 'open' strict equality are not adopted. The optimality properties of needed narrowing are retained in a weaker sense by inductively sequential narrowing. Even more recently, [10] has proposed a reformulation of needed narrowing as a goal transformation system, as well as an extension of its scope to a class of higher-order inductively sequential TRSs. Higher-order needed narrowing in the sense of [10] is sound and relatively complete, and it enjoys optimality properties similar to those known for the first-order case.

In spite of the good results from the viewpoint of neededness, [5, 2, 10] stick to a simplistic treatment of strict equality as a defined function symbol. Other recent works [14, 17, 7] have developed various *lazy narrowing calculi* based on *goal transformation* systems, where an *open semantics* of strict equality does not require the computation of ground solutions, as illustrated by Example 1. Moreover, [7] allows non-deterministic functions with so-called *call-time choice* semantics, borrowed from [13], and illustrated by the next example:

Example 2. Consider the TRS consisting of the rewrite rules for + given in Example 1 and the following rules:

$$\begin{aligned} coin &\to 0 \qquad\qquad double(X) \to X + X \\ coin &\to s(0) \end{aligned}$$

Intuitively, the constant function *coin* represents a non-deterministic choice. In order to evaluate the expression *double*(*coin*), *call-time choice* semantics chooses some possible value for the argument expression *coin*, and then applies the function *double* to that value. Therefore, when using *call-time choice*, *double*(*coin*) can be reduced to 0 and $s^2(0)$, but not to $s(0)$. □

As argued in [7, 13], the *call-time choice* view of non-determinism is the convenient one for many programming applications. Unfortunately, the narrowing methods proposed in [5, 2, 14, 17] are unsound w.r.t. *call-time choice* semantics. In [7], the *Constructor-based ReWriting*

Logic CRWL is proposed to formalize rewriting with *call-time* non-deterministic choices, and the *Constructor-based Lazy Narrowing Calculus CLNC* is provided as a sound and strongly complete goal solving method. Therefore, CLNC embodies the advantages of open strict equality and *call-time choice* non-determinism.

The aim of this paper is to refine CLNC in order to enrich its advantages with those of needed narrowing. We will combine goal transformation rules similar to those in [14, 17, 7] with the use of definitional trees to guide the selection of narrowing positions, in the vein of [15, 5, 3, 10]. As a result, we will propose a *Demand-driven Narrowing Calculus DNC* which can be proved sound and strongly complete w.r.t. CRWL's semantics (*call-time choice* non-determinism and open strict equality), contracts needed redexes, and maintains the good properties shown for needed narrowing in [5, 3]. All these properties qualify DNC as a convenient and efficiently implementable operational model for lazy FLP languages.

The organization of this paper is as follows: Section 2 contains some technical preliminaries regarding the constructor-based rewriting logic CRWL. Section 3 defines the subclass of rewriting systems used as programs in this work. In Section 4 we give a formal presentation of the calculus DNC. We discuss the soundness, completeness and optimality results in Section 5. Finally, some conclusions and plans for future work are drawn in Section 6.

2 Preliminaries

The reader is assumed to be familiar with the basic notions and notations of term rewriting, as presented e.g. in [6].

We assume a *signature* $\Sigma = DC_\Sigma \cup FS_\Sigma$ where $DC_\Sigma = \bigcup_{n\in\mathbb{N}} DC_\Sigma^n$ is a set of ranked *constructor* symbols and $FS_\Sigma = \bigcup_{n\in\mathbb{N}} FS_\Sigma^n$ is a set of ranked *function* symbols, all of them with associated *arity* and such that $DC_\Sigma \cap FS_\Sigma = \emptyset$. We also assume a countable set $\mathcal{V}$ of *variable* symbols. We write $Term_\Sigma$ for the set of (total) *terms* built over Σ and $\mathcal{V}$ in the usual way, and we distinguish the subset $CTerm_\Sigma$ of (total) constructor terms or (total) *c-terms*, which only make use of DC_Σ and $\mathcal{V}$. The subindex Σ will usually be omitted. Terms intend to represent possibly reducible expressions, while c-terms represent data values, not further reducible. The CRWL semantics also uses the constant symbol $\bot$, that plays the role of the *undefined* value. We define $\Sigma_\bot = \Sigma \cup \{\bot\}$; the sets $Term_\bot$ and $CTerm_\bot$ of (partial) terms and (partials) c-terms respectively, are defined in a natural way, by using the symbol $\bot$ just as a constant for syntatic purposes. Partial c-terms represent the result of partially evaluated expressions; thus, they can be seen as approximations to the value of expressions in the CRWL semantics. As usual notations we will write X,Y,Z for variables, *c,d* for constructor symbols, *f,g* for functions, *e* for terms and *s,t* for c-terms. Moreover, *Var(e)* will be used for the set of variables occurring in the term *e*. A term *e* is called *ground* term, if $Var(e) = \emptyset$. A term is called *linear* if it is does not contain multiple occurrences of any variable. A natural *approximation ordering* $\sqsubseteq$ over $Term_\bot$ can be defined as the least partial ordering over $Term_\bot$ satisfying the following properties: $\bot \sqsubseteq e$ for all $e \in Term_\bot$, and $h(e_1,...,e_n) \sqsubseteq h(e_1',...,e_n')$, if $e_i \sqsubseteq e_i'$ for all $i \in \{1,\dots,n\}$, $h \in DC^n \cup FS^n$.

To manipulate terms we give the following definitions. An *occurrence* or *position* is a sequence *p* of positive integers identifying a subterm in a term. For every term *e*, the set *O(e)* of *positions* in *e* is inductively defined as follows: the empty sequence denoted by ε, identifies *e* itself. For every term of the form $f(e_1,...,e_n)$, the sequence $i{\cdot}q$, where *i* is a positive integer not greater than *n* and *q* is a position, identifies the subterm of e_i at *q*. The subterm of *e* at *p* is denoted by $e|_p$ and the result of *replacing* $e|_p$ with *e'* in *e* is denoted by $e[e']_p$. If *p* and *q* are positions, we write $p \preccurlyeq q$ if *p* is above or is a *prefix* of *q*, and we write $p \parallel q$ if the positions are *disjoint*. The expression $p{\cdot}q$ denotes the position resulting from the concatenation of the positions p and q. The set $O(e)$ is partitioned into $\tilde{O}(e)$ and $O_{\mathcal{V}}(e)$ as follows: $\tilde{O}(e) = \{p \in O(e) \mid e|_p \notin \mathcal{V}\}$ and $O_{\mathcal{V}}(e) = \{p \in O(e) \mid e|_p \in \mathcal{V}\}$. If e is a linear term, *pos(X,e)* will be used for the position of the variable X occurring in e. The notation $Var_c(e)$ stands for the set of all variables occurring in e at some position whose ancestor positions are all occupied by constructors.

The sets of substitutions *CSubst* and $CSubst_\perp$ are defined as applications σ from $\mathcal{V}$ into CTerm and $\text{CTerm}_\perp$ respectively such its *domain* $\text{Dom}(\sigma) = \{X \in \mathcal{V} \mid \sigma(X) \neq X\}$ is finite. We will write θ,σ,μ for substitutions and $\{\}$ for the *identity* substitution. Substitutions are extended to morphisms on terms by $\sigma(f(e_1,...,e_n)) = f(\sigma(e_1),...,\sigma(e_n))$ for every term $f(e_1,...,e_n)$. The *composition* of two substitutions θ and σ is defined by $(\theta\circ\sigma)(X) = \theta(\sigma(X))$ for all $X \in \mathcal{V}$. A substitution σ is *idempotent* iff $\sigma\circ\sigma = \sigma$. We frequently write $\{X_1 \mapsto e_1,\ldots,X_n \mapsto e_n\}$ for the substitution that maps X_1 into e_1 … X_n into e_n. The *restriction* $\sigma\restriction_V$ of a substitution σ to a set $V \subseteq \mathcal{V}$, is defined by $\sigma\restriction_V(X) = \sigma(X)$ if $X \in V$ and $\sigma\restriction_V(X) = X$ if $X \notin V$. A substitution σ is *more general* than σ', denoted by $\sigma \lesssim \sigma'$, if there is a substitution τ with $\sigma' = \tau\circ\sigma$. If V is a set of variables, we write $\sigma = \sigma'$ [V] iff $\sigma\restriction_V = \sigma'\restriction_V$, and we write $\sigma \lesssim \sigma'$ [V] iff there is a substitution τ with $\sigma' = \tau\circ\sigma$ [V]. A term e' is an *instance* of e if there is a substitution σ with $e' = \sigma(e)$. In this case we write $e \preceq e'$. A term e' is a *variant* of e if $e \preceq e'$ and $e' \preceq e$.

Given a signature Σ, a *CRWL-program* $\mathfrak{R}$ is defined as a set of conditional rewrite rules of the form $f(t_1,...,t_n) \to r \Leftarrow a_1 == b_1, ..., a_m == b_m$ $(m \geq 0)$ with $f \in FS^n$, $t_1,...,t_n \in$ CTerm, $r,a_i,b_i \in$ Term, $i = 1,...,m$, and $f(t_1,...,t_n)$ is a linear term. The term $f(t_1,...,t_n)$ is called *left hand side* and r is the *right hand side*. From a CRWL-program we are interested in deriving sentences of two kinds: *approximation statements* $a \to b$, with $a \in \text{Term}_\perp$ and $b \in \text{CTerm}_\perp$, and *joinability statements* $a == b$, with $a, b \in \text{Term}_\perp$.

We formally define a *goal-oriented rewriting calculus* CRWL [7] by means of the following inference rules:

B *Bottom*: $e \to \perp$ for any $e \in \text{Term}_\perp$.

RR *Restricted Reflexivity*: $X \to X$ for any variable X.

DC *DeComposition*: $$\frac{e_1 \to t_1 \ \ldots\ e_n \to t_n}{c(e_1,...,e_n) \to c(t_1,...,t_n)}$$
for $c \in DC^n$, $t_i \in \text{CTerm}_\perp$.

OR *Outer Reduction*: $$\frac{e_1 \to t_1 \ \ldots\ e_n \to t_n \quad C \quad r \to t}{f(e_1,...,e_n) \to t}$$
if $t \neq \perp$ and $(f(t_1,...,t_n) \to r \Leftarrow C) \in [\mathfrak{R}]_\perp$ where $[\mathfrak{R}]_\perp = \{\theta(l \to r \Leftarrow C) \mid (l \to r \Leftarrow C) \in \mathfrak{R}, \theta \in \text{CSubst}_\perp\}$ is the set of all partial c-instances of rewrite rules in $\mathfrak{R}$.

J *Join* $$\frac{a \to t \qquad b \to t}{a == b}$$
for total $t \in$ CTerm.

In the sequel we will use the notations $\mathfrak{R} \vdash_{CRWL} \varphi$ to indicate that φ (an approximation statement or a joinability statement) can be deduced from the CRWL-program $\mathfrak{R}$ by finite application of the above rules. In the following lemma we collect some simple facts about provable statements involving c-terms, which will be used several times, possibly without mentioning them explicitly. The proof is straightforward by induction over the structure of c-terms or over the structure of CRWL-proofs (see [7] and [18] for details).

Lemma 1 (basic properties of the CRWL-deductions). *Let $\mathfrak{R}$ be a CRWL-program. For any $e \in Term_\perp$, $t,s \in CTerm_\perp$ and $\theta,\theta' \in CSubst_\perp$, we have:*

a) $\mathfrak{R} \vdash_{CRWL} t \to s \Leftrightarrow t \sqsupseteq s$. Furthermore, if $s \in CTerm$ is total, then $t \sqsupseteq s$ can be replaced by $t = s$.

b) $\mathfrak{R} \mid\!-_{CRWL}\, e \to t,\ t \sqsupseteq s \Rightarrow \mathfrak{R} \mid\!-_{CRWL}\, e \to s$.

c) $\mathfrak{R} \mid\!-_{CRWL}\, \theta(e) \to t,\ \theta \sqsubseteq \theta' \Rightarrow \mathfrak{R} \mid\!-_{CRWL}\, \theta'(e) \to t$, *with the same length and structure in both deductions.*

d) $\mathfrak{R} \mid\!-_{CRWL}\, e \to s \Rightarrow \mathfrak{R} \mid\!-_{CRWL}\, \theta(e) \to \theta(s)$, *if* θ *is a total substitution.*

e) $\mathfrak{R} \mid\!-_{CRWL}\, e \to s \Leftrightarrow \mathfrak{R} \mid\!-_{CRWL}\, e|_p \to t', \mathfrak{R} \mid\!-_{CRWL}\, e[t']_p \to s$ *for a certain* $t' \in CTerm_\perp$ *and for all* $p \in O(e)$.

3 Overlapping Definitional Tree and COISS

The demand narrowing calculus that we are going to present works for a class of CRWL-programs in which conditional rewrite rules with possibly overlapping left hand sides must be organized in a hierarchical structure called *definitional tree* [2]. More precisely, we choose to reformulate the notion of *overlapping definitional tree* and *overlapping inductively sequential system* introduced in [3] including now conditional rules.

Definition 1 (ODT). *Let* $\Sigma = DC \cup FS$ *be a signature and* $\mathfrak{R}$ *a CRWL-program over* Σ. *A* **pattern** *is any linear term of the form* $f(t_1,...,t_n)$, *where* $f \in FS^n$ *is a function symbol and* t_i *are c-terms.* $\mathfrak{I}$ *is an* **Overlapping Definitional Tree** *(***ODT** *for short) with pattern* π *iff its depth is finite and one of the following cases holds:*

- $\mathfrak{I} = \underline{rule}(l \to r_1 \Leftarrow C_1 \mid ... \mid r_m \Leftarrow C_m)$, *where* $l \to r_i \Leftarrow C_i$ *for all i in* $\{1,...,m\}$ *is a variant of a rule in* $\mathfrak{R}$ *such that* $l = \pi$.
- $\mathfrak{I} = \underline{case}(\pi, X, [\mathfrak{I}_1,...,\mathfrak{I}_k])$, *where X is a variable in* π, $c_1,...,c_k \in DC$ *for some* $k > 0$ *are pairwise different constructors of* $\mathfrak{R}$, *and for all i in* $\{1,...,k\}$, $\mathfrak{I}_i$ *is an ODT with pattern* $\sigma_i(\pi)$, *where* $\sigma_i = \{X \mapsto c_i(Y_1,...,Y_{m_i})\}$, m_i *is the arity of* c_i *and* $Y_1,...,Y_{m_i}$ *are new distinct variables.*

We represent an ODT $\mathfrak{I}$ with pattern π using the notation $\mathfrak{I}_\pi$. An **ODT of a function symbol** $f \in FS^n$ defined by $\mathfrak{R}$ is an ODT $\mathfrak{I}$ with pattern $f(X_1,...,X_n)$, where $X_1,...,X_n$ are new distinct variables.

Definition 2 (COISS). *A function f is called* **overlapping inductively sequential** *w.r.t. a CRWL-program* $\mathfrak{R}$ *iff there exists an ODT* $\mathfrak{I}_f$ *of f such that the collection of all the rewrite rules* $l \to r_i \Leftarrow C_i$ $(1 \le i \le m)$ *obtained from the different nodes* $\underline{rule}(l \to r_1 \Leftarrow C_1 \mid ... \mid r_m \Leftarrow C_m)$ *occurring in* $\mathfrak{I}_f$ *equals, up to variants, the collection of all the rewrite rules in* $\mathfrak{R}$ *whose left hand side has the root symbol f. A CRWL-program* $\mathfrak{R}$ *is called* **Conditional Overlapping Inductively Sequential System** *(***COISS**, *for short) iff each function defined by* $\mathfrak{R}$ *is overlapping inductively sequential.*

Example 3. We consider the next CRWL-program $\mathfrak{R}$. Is easy to check that $\mathfrak{R}$ is a COISS:

insertion (X, [ ]   , Zs) $\to$ true $\Leftarrow$ Zs == [X]
insertion (X, [Y|Ys], Zs) $\to$ true $\Leftarrow$ Zs == [X, Y|Ys]
insertion (X, [Y|Ys], Zs) $\to$ true $\Leftarrow$ *insertion* (X, Ys, Us) == true, Zs == [Y|Us]

The defined symbol *insertion* has the following ODT:

<u>case</u>(*insertion* (X, Xs, Zs), Xs, [
  <u>rule</u>(*insertion* (X, [ ]   , Zs) $\to$ true $\Leftarrow$ Zs == [X]),
  <u>rule</u>(*insertion* (X, [Y|Ys], Zs) $\to$ true $\Leftarrow$ Zs == [X, Y|Ys] |

true $\Leftarrow$ *insertion* (X, Ys, Us) == true, Zs == [Y|Us])]) □

Let $\mathfrak{R}$ be any COISS. For our discussion of a demand narrowing calculus with ODTs over COISSs we are going to use a presentation similar to that of the CLNC calculus (see [7]). So, goals for $\mathfrak{R}$ are essentially finite conjunctions of CRWL-statements, and solutions are c-substitutions such that the goal affected by the substitution becomes CRWL-provable. The precise definition of admissible goal includes a number of technical conditions which will be defined and explained below. The general idea is to ensure the computation of solutions which are correct w.r.t. CRWL's *call-time choice* semantics, while using ODTs in a similar way to [5, 3] to ensure that all the narrowing steps performed during the computation are needed ones.

4 The DNC Calculus

In this section we present the basis of our *Demand Narrowing Calculus* (shortly, *DNC*) for goal solving. We give first a precise definition for the class of admissible goals and solutions we are going to work with.

Definition 3 (Admissible goals). *An* **admissible goal** *for a given COISS $\mathfrak{R}$ must have the form $G \equiv \exists U.\ S \ \square\ P \ \square\ E$, where the symbols $\square$ and ',' must be interpreted as conjunction, and:*

- *$EVar(G) =_{def} U$ is the set of so-called* **existential variables** *of the goal G. These are intermediate variables, whose bindings in a solution may be partial c-terms.*
- *$S \equiv X_1 \approx s_1, \dots, X_n \approx s_n$ is a set of equations, called* **solved part***. Each s_i must be a total c-term, and each X_i must occur exactly once in the whole goal. This represents an idempotent c-substitution $\sigma_S = \{X_1 \mapsto s_1, \dots, X_n \mapsto s_n\}$.*
- *$P \equiv t_1 \to R_1, \dots, t_k \to R_k$ is a multiset of* **productions** *where each R_i is a variable and t_i is a term or a pair of the form $<\pi, \mathfrak{I}>$, where π is an instance of the pattern in the root of an ODT $\mathfrak{I}$. Those productions $l \to R$ whose left hand side l is simply a term are called* **suspensions***, while those whose left hand side is of the form $<\pi, \mathfrak{I}>$ are called* **demanded productions***. $PVar(P) =_{def} \{R_1,\dots,R_k\}$ is called the set of* **produced variables** *of the goal G. The* **production relation** *between G-variables is defined by $X \gg_P Y$ iff there is some $1 \le i \le k$ such that $X \in Var(t_i)$ and $Y = R_i$ (if t_i is a pair $<\pi, \mathfrak{I}>$, $X \in Var(t_i)$ must be interpreted as $X \in Var(\pi)$).*
- *$E \equiv l_1 == r_1, \dots, l_m == r_m$ is a multiset of* **strict equations***. $DVar(P \ \square\ E)$, called the set of* **demanded variables** *of the goal G, is the least subset of $Var(P \ \square\ E)$ which fulfills the following conditions:*
  - *$(l == r) \in E \Rightarrow Var_C(l) \cup Var_C(r) \subseteq DVar(P \ \square\ E)$.*
  - *$(<t, \underline{case}(\pi, X, [\mathfrak{I}_1,\dots,\mathfrak{I}_q])> \to R) \in P$ with $t|_{pos(X,\pi)} = Y \in Var(t)$ and $R \in DVar(P \ \square\ E)$ $\Rightarrow Y \in DVar(P \ \square\ E)$.*

Additionally, any admissible goal must fulfill the following *admissibility conditions*:

LIN Each produced variable is produced only once, i.e. variables $R_1,\dots,R_k$ must be different.

EX All the produced variables must be existential, i.e. $PVar(P) \subseteq EVar(G)$.

CYC The transitive closure of the production relation $\gg_P$ must be irreflexive, or equivalently, a strict partial order.

SOL The solved part S contains no produced variables.

DT For each production $<\pi,\Im> \to R$ in P, the variable R is demanded (i.e. $R \in DVar(P \square E)$), and the variables of $\Im$ have no occurrences in other place of the goal.

Admissibility conditions are natural in so far they are satisfied by all the goals arising from *initial goals* $G_0 \equiv \exists\, \emptyset.\ \emptyset \square \emptyset \square E$ by means of the goal transformations presented below. The following distinction between the two possible kinds of productions is useful:

1. Demanded productions $<\pi,\Im> \to R$ are used to compute a value for the demanded variable R. The value will be shared by all occurrences of R in the goal, in order to respect *call-time choice* semantics.
2. Suspensions $f(t_1,...,t_n) \to R$ eventually become demanded productions if R becomes demanded, or else disappear if R becomes absent from the rest of the goal. See the goal transformations **SA** and **EL** in Subsection 4.2. Suspensions $X \to R$ and $c(e_1,...,e_n) \to R$ are suitably treated by other goal transformations.

Definition 4 (Solutions). *Let $G \equiv \exists U.\ S \square P \square E$ be an admissible goal for a COISS $\Re$, and θ a partial c-substitution. We say that θ is a* **solution** *for G with respect to $\Re$ if:*

- *For each $X \notin PVar(P)$, $\theta(X)$ is a total c-term.*
- *For each solved equation $(X_i \approx s_i) \in S$, $\theta(X_i) = \theta(s_i)$.*
- *For each suspension $(t \to R) \in P$, $\Re \vdash_{CRWL} \theta(t) \to \theta(R)$.*
- *For each demanded production $(<f(t_1,...,t_n),\Im> \to R) \in P$, $\Re \vdash_{CRWL} \theta(f(t_1,...,t_n)) \to \theta(R)$.*
- *For each strict equation $(l == r) \in E$, $\Re \vdash_{CRWL} \theta(l) == \theta(r)$.*

A ***witness*** *$\mathcal{M}$ for the fact that θ is a solution of G is defined as a multiset containing the CRWL proofs mentioned above. We write Sol(G) for the set of all solutions for G.*

The next result will be useful to prove properties about DNC and shows that CRWL semantics does not accept an undefined value for demanded variables.

Lemma 2 (Demand lemma). *If θ is a solution of an admissible goal $G \equiv \exists U.\ S \square P \square E$ for a COISS $\Re$ and $X \in DVar(P \square E)$, then $\theta(X) \neq \bot$.*

Proof. Let $G \equiv \exists U.\ S \square P \square E$ be an admissible goal for a COISS $\Re$ and $X \in DVar(P \square E)$. We use induction on the order $\gg_P^+$ (the transitive closure of the production relation is well founded, due to the property **CYC** of admissible goals). We consider the two cases for X.

a) $X \in Var_C(l) \cup Var_C(r)$ with $(l == r) \in E$.

   We suppose that $X \in Var_C(l)$ (analogous if $X \in Var_C(r)$). There are a finite number $k \geq 0$ of constructor symbols above X in *l*. Due to θ is a solution of G, $\Re \vdash_{CRWL} \theta(l) == \theta(r)$ and exists $u \in CTerm$ with $\Re \vdash_{CRWL} \theta(l) \to u$ and $\Re \vdash_{CRWL} \theta(r) \to u$ using the CRWL-rule **J**. Since $\Re \vdash_{CRWL} \theta(l) \to u$, where $\theta \in CSubst_\bot$ and $u \neq \bot$, it is easy to see that *u* has the same constructor symbols and in the same orden that $\theta(l)$. So, the CRWL-deduction takes the rule **DC** k times on $\theta(l) \to u$ to obtain $\Re \vdash_{CRWL} \theta(X) \to t$ with $t \in CTerm$. It follows that $\theta(X) \neq \bot$ because $t \neq \bot$. Moreover, $\theta(X) \in CTerm_\bot$ and $t \in CTerm$ implies that $\theta(X)$ is a total c-term.

b) $X = t|_{pos(Y,\pi)}$ with $(<t,\underline{case}(\pi,Y,[\Im_1,...,\Im_k])> \to R) \in P$.

   In this case, $t = f(t_1,...,t_n)$ with $f \in FS^n$, and since θ is a solution, $\Re \vdash_{CRWL} f(\theta(t_1),...,\theta(t_n)) \to \theta(R)$. By Definition 3, $X \gg_P R$ and hence $X \gg_P^+ R$. Since $R \in DVar(P \square E)$ due to the property **DT** of admissible goals, the induction hypothesis yields $\theta(R) \neq \bot$. Moreover, $pos(Y,\pi) = i{\cdot}p$ with $1 \leq i \leq n$ and $p \in O(t_i)$ because $\pi \preceq t$. Therefore, the CRWL-deduction $\Re \vdash_{CRWL} f(\theta(t_1),...,\theta(t_n)) \to \theta(R)$ must be of the form **OR**:

$$\frac{\theta(t_1) \to t_1' \ \ldots\ \theta(t_i) \to t_i' \ \ldots\ \theta(t_n) \to t_n' \quad C \quad r \to \theta(R)}{f(\theta(t_1),\ldots,\theta(t_n)) \to \theta(R)}$$

where $(f(t_1',\ldots,t_i',\ldots,t_n') \to r \Leftarrow C) \in [\mathfrak{R}]_\perp$, and $t_1',\ldots,t_i',\ldots,t_n' \in CTerm_\perp$. Due to the form of the ODT in the production, t_i' has a constructor symbol c_j $(1 \le j \le k)$ in the position p. Moreover, there must be only constructor symbols above c_j in t_i'. So, $t_i' \neq \perp$. Moreover, since $\mathfrak{R} \vdash_{CRWL} \theta(t_i) \to t_i'$ where $t_i' \in CTerm_\perp$ and $\theta \in CSubst_\perp$, there must be the same constructor symbols and in the same order above $\theta(X)$ in the position p of $\theta(t_i)$ (recall $\pi \le t$ with $X = t_i|_p$). It follows that $\mathfrak{R} \vdash_{CRWL} \theta(t_i) \to t_i'$ applies the CRWL-rule **DC** over $\theta(t_i) \to t_i'$ to yield $\mathfrak{R} \vdash_{CRWL} \theta(X) \to c_j(\ldots)$. We conclude that $\theta(X) \neq \perp$. □

The aim when using DNC is to transform an initial goal into a *solved goal* of the form $\exists U.\, S \square \square$ with $S \equiv X_1 \approx s_1,\ldots,X_n \approx s_n$ representing a solution $\sigma_S = \{X_1 \mapsto s_1,\ldots,X_n \mapsto s_n\}$. The notation ■, used in failure rules, represents an inconsistent goal without any solution. The calculus DNC consists of a set of transformation rules for goals. Each transformation takes the form $G \rightsquigarrow G'$, specifying one of the possible ways of performing one step of goal solving. Derivations are sequences of $\rightsquigarrow$-steps. In addition, to the purpose of applying the rules, we see conditions $a == b$ as symmetric.

4.1 Rules for Strict Equations

DPI Demanded Production Introduction

$\exists U\,.\, S \square P \square f(s_1,\ldots,s_n) == t, E \rightsquigarrow \exists R, U\,.\, S \square {<}f(s_1,\ldots,s_n)\,,\, \mathfrak{I}_{f(X_1,\ldots,X_n)}{>} \to R, P \square R == t, E$

if $f \in FS$, and both R and all variables in $\mathfrak{I}_{f(X_1,\ldots,X_n)}$ are new variables.

DC DeComposition

$\exists U\,.\, S \square P \square c(s_1,\ldots,s_n) == c(t_1,\ldots,t_n), E \rightsquigarrow \exists U\,.\, S \square P \square s_1 == t_1, \ldots, s_n == t_n, E$ if $c \in DC$.

ID IDentity

$\exists U\,.\, S \square P \square X == X, E \rightsquigarrow \exists U\,.\, S \square P \square E$ if $X \notin PVar(P)$.

BD BinDing

$\exists U\,.\, S \square P \square X == t, E \rightsquigarrow \exists U\,.\, X \approx t\,,\, \sigma(S \square P \square E)$

if $t \in CTerm$, $Var(t) \cap PVar(P) = \emptyset$, $X \notin PVar(P)$, $X \notin Var(t)$, and $\sigma = \{X \mapsto t\}$.

IM IMitation

$\exists U\,.\, S \square P \square X == c(s_1,\ldots,s_n), E \rightsquigarrow$

$\exists X_1,\ldots,X_n, U\,.\, X \approx c(X_1,\ldots,X_n)\,,\, \sigma(S \square P \square X_1 == s_1,\ldots,X_n == s_n, E)$

if $c \in DC$, $c(s_1,\ldots,s_n) \notin CTerm$ or $Var(c(s_1,\ldots,s_n)) \cap PVar(P) \neq \emptyset$, $X \notin PVar(P)$, $X \notin Var_c(c(s_1,\ldots,s_n))$, and $\sigma = \{X \mapsto c(X_1,\ldots,X_n)\}$ with $X_1,\ldots,X_n$ new variables.

4.2 Rules for Productions

IB Input Binding

$\exists R, U . S \square t \to R, P \square E \leadsto \exists U . S \square \sigma(P \square E)$ if $t \in$ CTerm, and $\sigma = \{R \mapsto t\}$.

IIM Input IMitation

$\exists R, U . S \square c(t_1,...,t_n) \to R, P \square E \leadsto \exists R_1,...,R_n, U . S \square \sigma(t_1 \to R_1, ... , t_n \to R_n, P \square E)$

if $c(t_1,...,t_n) \notin$ CTerm, $R \in$ DVar(P $\square$ E), and $\sigma = \{R \mapsto c(R_1,...,R_n)\}$ with $R_1,...,R_n$ new variables.

SA Suspension Awakening

$\exists R, U . S \square f(t_1,...,t_n) \to R, P \square E \leadsto \exists R, U . S \square \langle f(t_1,...,t_n) , \Im_{f(X1,...,Xn)} \rangle \to R, P \square E$

if $f \in$ FS, $R \in$ DVar(P $\square$ E), and all variables in $\Im_{f(X1,...,Xn)}$ are new variables.

EL ELimination

$\exists R, U . S \square t \to R, P \square E \leadsto \exists U . S \square P \square E$ if $R \notin$ Var(P $\square$ E).

RRA Rewrite Rule Application *(don't know choice: 1≤i≤k)*

$\exists R, U. S \square \langle t , \underline{rule}(l \to r_1 \Leftarrow C_1 \mid ... \mid r_k \Leftarrow C_k) \rangle \to R, P \square E \leadsto$

$\exists 8, R, U. S \square \sigma_f(R_1) \to R_1,..., \sigma_f(R_m) \to R_m, \sigma_c(r_i) \to R, P \square \sigma_c(C_i), E$

where

- $\sigma = \sigma_c \cup \sigma_f$ with Dom(σ) = Var(l) and $\sigma(l) = t$
- $\sigma_c =_{def} \sigma \upharpoonright \mathrm{Dom}_c(\sigma)$, where $\mathrm{Dom}_c(\sigma) = \{X \in \mathrm{Dom}(\sigma) \mid \sigma(X) \in \mathrm{CTerm}\}$
- $\sigma_f =_{def} \sigma \upharpoonright \mathrm{Dom}_f(\sigma)$, where $\mathrm{Dom}_f(\sigma) = \{X \in \mathrm{Dom}(\sigma) \mid \sigma(X) \notin \mathrm{CTerm}\} = \{R_1,...,R_m\}$
- $8 = \mathrm{Var}(l \to r_i \Leftarrow C_i) \setminus \mathrm{Dom}_c(\sigma)$

CS Case Selection

$\exists R, U . S \square \langle t , \underline{case}(\pi,X,[\Im_1,...,\Im_k]) \rangle \to R, P \square E \leadsto \exists R, U . S \square \langle t , \Im_i \rangle \to R, P \square E$

if $t|_{pos(X,\pi)} = c_i(...)$, with $1 \le i \le k$ given by t, where c_i is the constructor symbol associated to $\Im_i$.

DI Demanded Instantiation *(don't know choice: 1≤i≤k)*

$\exists R, U . S \square \langle t , \underline{case}(\pi,X,[\Im_1,...,\Im_k]) \rangle \to R, P \square E \leadsto$

$\exists Y_1,...,Y_{mi},R, U . Y \approx c_i(Y_1,...,Y_{mi}), \sigma(S \square \langle t , \Im_i \rangle \to R, P \square E)$

if $t|_{pos(X,\pi)} = Y$, $Y \notin$ PVar(P), $\sigma = \{Y \mapsto c_i(Y_1,...,Y_{mi})\}$ with c_i ($1 \le i \le k$) the constructor symbol associated to $\Im_i$ and $Y_1,...,Y_{mi}$ are new variables.

DN Demanded Narrowing

$\exists R, U . S \square \langle t , \underline{case}(\pi,X,[\Im_1,...,\Im_k]) \rangle \to R, P \square E \leadsto$

$\exists R', R, U . S \square \langle t|_{pos(X,\pi)} , \Im_{g(Y1,...,Ym)} \rangle \to R',$

$\langle t[R']_{pos(X,\pi)} , \underline{case}(\pi,X,[\Im_1,...,\Im_k]) \rangle \to R, P \square E$

if $t|_{pos(X,\pi)} = g(...)$ with $g \in FS^m$, and both R' and all variables in $\Im_{g(Y1,...,Ym)}$ are new variables.

4.3 Rules for Failure Detection

CF ConFlict

$\exists U\,.\; S \,\Box\, P \,\Box\, c(s_1,\ldots,s_n) == d(t_1,\ldots,t_m),\, E \rightsquigarrow \blacksquare$ if $c,d \in DC$ and $c \neq d$.

CY CYcle

$\exists U\,.\; S \,\Box\, P \,\Box\, X == t,\, E \rightsquigarrow \blacksquare$ if $X \neq t$ and $X \in \mathrm{Var}_c(t)$.

UC Unconvered Case

$\exists R, U\,.\; S \,\Box <t\,,\, \underline{case}(\pi, X, [\Im_1,\ldots,\Im_k])> \rightarrow R,\, P \,\Box\, E \rightsquigarrow \blacksquare$

if $t|_{pos(X,\pi)} = c(\ldots)$ and $c \notin \{c_1,\ldots,c_k\}$, where c_i is the constructor symbol associated to $\Im_i$ $(1 \leq i \leq k)$.

Example 4. We compute the solutions from Example 2. This ilustrates the use of productions for ensuring *call-time choice* and the use of ODTs for ensuring needed narrowing steps.

$$\begin{array}{ll}
\Box\,\Box\, \underline{doble(coin) == N} \rightsquigarrow & \textbf{DPI} \\
\exists R\,.\,\Box\, \underline{<doble(coin),\Im_{doble(X)}> \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{RRA} \\
\exists X,R\,.\,\Box\, coin \rightarrow X,\, \underline{X+X \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{SA} \\
\exists X,R\,.\,\Box\, \underline{coin \rightarrow X},\, <X+X,\Im_{A+B}> \rightarrow R \,\Box\, R == N \rightsquigarrow & \textbf{SA} \\
\exists X,R\,.\,\Box\, \underline{<coin,\Im_{coin}> \rightarrow X},\, <X+X,\Im_{A+B}> \rightarrow R \,\Box\, R == N \rightsquigarrow & \textbf{RRA} \\
\exists X,R\,.\,\Box\, \underline{0 \rightarrow X},\, <X+X,\Im_{A+B}> \rightarrow R \,\Box\, R == N \rightsquigarrow_{\{X \mapsto 0\}} & \textbf{IB} \\
\exists R\,.\,\Box\, \underline{<0+0,\Im_{A+B}> \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{CS} \\
\exists R\,.\,\Box\, \underline{<0+0,\Im_{0+B}> \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{RRA} \\
\exists R\,.\,\Box\, \underline{0 \rightarrow R} \,\Box\, R == N \rightsquigarrow_{\{R \rightarrow 0\}} & \textbf{IB} \\
\Box\,\Box\, \underline{0 == N} \rightsquigarrow_{\{N \mapsto 0\}} & \textbf{BD} \\
N \approx 0 \,\Box\,\Box \qquad \Rightarrow \;\; \underline{\textit{computed solution}}\text{: } \sigma_1 = \{N \mapsto 0\} &
\end{array}$$

$$\begin{array}{ll}
\Box\,\Box\, \underline{doble(coin) == N} \rightsquigarrow & \textbf{DPI} \\
\exists R\,.\,\Box\, \underline{<doble(coin),\Im_{doble(X)}> \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{RRA} \\
\exists X,R\,.\,\Box\, coin \rightarrow X,\, \underline{X+X \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{SA} \\
\exists X,R\,.\,\Box\, \underline{coin \rightarrow X},\, <X+X,\Im_{A+B}> \rightarrow R \,\Box\, R == N \rightsquigarrow & \textbf{SA} \\
\exists X,R\,.\,\Box\, \underline{<coin,\Im_{coin}> \rightarrow X},\, <X+X,\Im_{A+B}> \rightarrow R \,\Box\, R == N \rightsquigarrow & \textbf{RRA} \\
\exists R\,.\,\Box\, \underline{<s(0)+s(0),\Im_{A+B}> \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{CS} \\
\exists R\,.\,\Box\, \underline{<s(0)+s(0),\Im_{s(C)+B}> \rightarrow R} \,\Box\, R == N \rightsquigarrow & \textbf{RRA} \\
\exists X,R\,.\,\Box\, \underline{s(0) \rightarrow X},\, <X+X,\Im_{A+B}> \rightarrow R \,\Box\, R == N \rightsquigarrow_{\{X \mapsto s(0)\}} & \textbf{IB} \\
\exists R\,.\,\Box\, \underline{s(0+s(0)) \rightarrow R} \,\Box\, R == N \rightsquigarrow_{\{R \mapsto s(R')\}} & \textbf{IIM} \\
\exists R'\,.\,\Box\, \underline{0+s(0) \rightarrow R'} \,\Box\, s(R') == N \rightsquigarrow & \textbf{SA} \\
\exists R'\,.\,\Box\, \underline{<0+s(0),\Im_{A'+B'}> \rightarrow R'} \,\Box\, s(R') == N \rightsquigarrow & \textbf{CS} \\
\exists R'\,.\,\Box\, \underline{<0+s(0),\Im_{0+B'}> \rightarrow R'} \,\Box\, s(R') == N \rightsquigarrow & \textbf{RRA} \\
\exists R'\,.\,\Box\, \underline{s(0) \rightarrow R'} \,\Box\, s(R') == N \rightsquigarrow_{\{R' \mapsto s(0)\}} & \textbf{IB} \\
\Box\,\Box\, \underline{s(s(0)) == N} \rightsquigarrow_{\{N \mapsto s(s(0))\}} & \textbf{BD} \\
N \approx s(s(0)) \,\Box\,\Box \qquad \Rightarrow \;\; \underline{\textit{computed solution}}\text{: } \sigma_2 = \{N \mapsto s(s(0))\} &
\end{array}$$

We note that both computations are in essence needed narrowing derivations module non-deterministic choices between overlapping rules, as in inductively sequential narrowing [3]. □

5 Properties of DNC

To prove soundness and completeness of DNC w.r.t. CRWL semantics we use techniques similar to those used for the CLNC calculus in [7]. The first result proves correctness of a single DNC step. It says that transformation steps preserve admissibility of goals, fail only in case of unsatisfiable goals and do not introduce new solutions.

Lemma 3 (Correctness Lemma).

a) *If $G \rightsquigarrow G'$ and G is admissible, then G' is admissible.*

b) *If $G \rightsquigarrow \blacksquare$ then $Sol(G) = \varnothing$.*

c) *If $G \rightsquigarrow G'$ and $\theta' \in Sol(G')$ then there exists $\theta \in Sol(G)$ with $\theta = \theta'\ [\mathcal{V} - (EVar(G) \cup EVar(G'))]$.*

The following soundness result follows easily from Lemma 3. It ensures that computed answers for a goal G are indeed solutions of G.

Theorem 1 (Soundness of DNC). *If G_0 is an initial goal and $G_0 \rightsquigarrow G_1 \rightsquigarrow \ldots \rightsquigarrow G_n$, where $G_n \equiv \exists U.\ S \,\square\, \square$, then $\sigma_S \in Sol(G_0)$.*

Proof. If we repeatedly backwards apply *c)* of Lemma 3, we obtain $\theta \in Sol(G_0)$ such that $\theta = \sigma_S\ [\mathcal{V} - \bigcup_{0\le i\le n} EVar(G_i)]$. By noting that $EVar(G_0) = \varnothing$ and $Var(G_0) \cap \bigcup_{0\le i\le n} EVar(G_i) = \varnothing$, we conclude $\theta = \sigma_S\ [Var(G_0)]$. But then, since σ_S is a total c-substitution, $\sigma_S \in Sol(G_0)$. □

Completeness of DNC is proved with the help of a well-founded ordering $\rhd$ for admissible goals and a result that guarantees that DNC transformations can be chosen to make progress towards the computation of a given solution. A similar technique is used in [7] to prove completeness of CLNC, but the well-founded ordering there is a simpler one.

Definition 5 (Well-founded ordering for admissible goals). *Let $\mathfrak{R}$ be a COISS, $G \equiv \exists U.\ S \,\square\, P \,\square\, E$ an admissible goal for $\mathfrak{R}$ and $\mathcal{M}$ a witness for $\theta \in Sol(G)$. We define:*

a) *A multiset of pairs of natural numbers $\mathcal{W}(G,\theta,\mathcal{M}) =_{def} \{||\Pi||_{G,\theta} \mid \Pi \in \mathcal{M}\}$, called the* **weight of the witness***, where for each CRWL-proof $\Pi \in \mathcal{M}$, the pair of numbers $||\Pi||_{G,\theta} \in \mathbb{N} \times \mathbb{N}$ is:*

- *if $\Pi \equiv \mathfrak{R} \vdash_{CRWL} \theta(t) \to \theta(R)$ for $(\langle t, \mathfrak{I}\rangle \to R) \in P$, then $||\Pi||_{G,\theta} =_{def} (|\Pi|_{OR}, |\Pi|)$.*
- *if $\Pi \equiv \mathfrak{R} \vdash_{CRWL} \theta(t) \to \theta(R)$ for $(t \to R) \in P$, then $||\Pi||_{G,\theta} =_{def} (|\Pi|_{OR}, 1 + |\Pi|)$.*
- *if $\Pi \equiv \mathfrak{R} \vdash_{CRWL} \theta(t) == \theta(s)$ for $(t == s) \in E$, then $||\Pi||_{G,\theta} =_{def} (|\Pi|_{OR}, |\Pi|)$.*

In all the cases, $|\Pi|_{OR}$ is the number of deduction steps in Π which use the CRWL deduction rule ***OR*** *and $|\Pi|$ is the total number of deduction steps in Π.*

b) *A multiset of natural numbers $\mathcal{D}(G) =_{def} \{\ |\mathfrak{I}| \mid (\langle t, \mathfrak{I}\rangle \to R) \in P\ \}$, called the* **weight of the definitional trees** *of the goal G, where each $|\mathfrak{I}| \in \mathbb{N}$ is the total number of nodes in the tree $\mathfrak{I}$.*

Over sets $(G,\theta,\mathcal{M})$ such that G is an admissible goal for $\mathfrak{R}$ and $\mathcal{M}$ is a witness of $\theta \in Sol(G)$, we define a well-founded ordering $(G,\theta,\mathcal{M}) \rhd (G',\theta',\mathcal{M}') \Leftrightarrow_{def} (\mathcal{W}(G,\theta,\mathcal{M}),\mathcal{D}(G)) >_{lex} (\mathcal{W}(G',\theta',\mathcal{M}'),\mathcal{D}(G'))$, where $>_{lex}$ is the lexicographic product of $>_{mul1}$ and $>_{mul2}$, $>_{mul1}$ is the multiset order for multisets over $\mathbb{N}\times\mathbb{N}$ induced by the lexicographic product of $>_{\mathbb{N}}$ and $>_{\mathbb{N}}$, and $>_{mul2}$ is the multiset order for multisets over $\mathbb{N}$ induced by $>_{\mathbb{N}}$. See [6] for definitions of these notions.

Lemma 4 (Progress Lemma). *Assume an admissible goal G for some COISS, which is not in solved form, and a given solution* $\theta \in Sol(G)$ *with witness* $\mathcal{M}$. *Then:*

a) *There is some DNC transformation applicable to G.*

b) *Whatever applicable DNC transformation is chosen, there is some transformation step* $G \rightsquigarrow G'$ *and some solution* $\theta' \in Sol(G')$ *with a witness* $\mathcal{M}'$, *such that* $\theta = \theta'\ [\mathcal{V} - (EVar(G) \cup EVar(G'))]$ *and* $(G,\theta,\mathcal{M}) \rhd (G',\theta',\mathcal{M}')$.

By reiterated application of Lemma 4, the following completeness result is easy to prove. The proof reveals that DNC is *strongly complete*, i.e. completeness does not depend on the choice of the transformation rule (among all the applicable rules) in the current goal.

Theorem 2 (Completeness of DNC). *Let* $\Re$ *be a COISS, G an initial goal and* $\theta \in Sol(G)$. *Then there exists a solved form* $\exists U.\ S \ \square\square$ *such that* $G \rightsquigarrow^{*} \exists U.\ S \ \square\square$ *and* $\sigma_S \lesssim \theta\ [Var(G)]$.

Proof. Thanks to Lemma 4 it is possible to construct a derivation $G \equiv G_0 \rightsquigarrow G_1 \rightsquigarrow G_2 \rightsquigarrow \ldots$ for which there exist $\theta_0 = \theta, \theta_1, \theta_2, \ldots$ and $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \ldots$ such that $\theta_i = \theta_{i-1}\ [\mathcal{V} - (EVar(G_{i-1}) \cup EVar(G_i))]$, $\mathcal{M}_i$ is a witness of $\theta_i \in Sol(G_i)$ and $(G_i,\theta_i,\mathcal{M}_i) \rhd (G_{i-1},\theta_{i-1},\mathcal{M}_{i-1})$. Since $\rhd$ is well founded, such a derivation must be finite, ending with a solved form $G_n \equiv \exists\ U.\ S\ \square\ \square$. Since $EVar(G_0) = \emptyset$ and $Var(G_0) \cap EVar(G_i)$ for all $i = 1,\ldots,n$ it is easy to see that $\theta_n = \theta\ [Var(G)]$. Now, if $X \in Var(G)$ and there is an equation $X \approx t$ in S, we can use the facts $\theta_n = \theta\ [Var(G)]$ and $\theta_n \in Sol(S)$ to obtain $\theta(X) = \theta_n(X) = \theta_n(t) = \theta_n(\sigma_S(X))$. It follows that $\theta = \theta_n \circ \sigma_S\ [Var(G)]$, and thus $\sigma_S \lesssim \theta\ [Var(G)]$. □

Let us now briefly analyze the behaviour of DNC computations from the viewpoint of needed narrowing. Obviously, the goal transformation rules for productions which deal with ODTs already present in the goal (namely, **RRA**, **CS**, **DI** and **DN**) lead to performing needed narrowing steps. On the other hand, the goal transformation **DPI** and **SA** are the only way to introduce demanded productions $<\pi,\Im> \to R$ into the goal. In both cases, R is a demanded variable in the new goal. From the definition of demanded variables within Definition 3, it is easily seen that any demanded variable always occurs at some position of the goal which is needed in the sense of needed narrowing (viewing == as a function symbol, so that the conceptual framework from [5, 3] can be applied). Therefore, DNC derivations actually encode needed narrowing derivations in the sense of [5, 3], with an improved treatment of == as open strict equality and *call-time* non-deterministic choice.

Regarding the range of applicability of DNC, the restriction to the class of COISSs is not a serious one. Using techniques known from [4, 15] we have proved in [18] that any CRWL-program $\Re$ can be effectively transformed into a COISS $\Re'$ using new auxiliary function symbols, such that for all $e \in Term_\perp$ and $t \in CTerm_\perp$ in the original signature of $\Re$ one has $\Re \vdash_{CRWL} e \to t \Leftrightarrow \Re' \vdash_{CRWL} e \to t$. The combination of these transformation techniques with the DNC calculus offers a sound and complete narrowing procedure for the whole class of the left-linear constructor-based conditional rewrite systems.

6 Conclusions

We have presented a demand narrowing calculus DNC for COISS, a wide class of constructor-based conditional TRSs. We have proved that DNC conserves the good properties of needed narrowing [5, 3] while being sound and strongly complete w.r.t. CRWL semantics [7], which ensures a *call-time choice* treatment of non-determinism and an open interpretation of strict equality.

Because of these results, we believe that DNC contributes to bridge the gap between formally defined lazy narrowing calculi and existing languages such as Curry [9] and *TOY* [1]. As future work, we plan to extend DNC and COISS in order to include more expressive features, such as higher-order rewrite rules and constraints.

Acknowledgement

The author is thankful to Mario Rodríguez-Artalejo and Francisco J. López-Fraguas for their collaboration and comments and for the help in preparing the final version of this paper.

References

[1] M. Abengózar-Carneros et al. *TOY*: a multiparadigm declarative language, version 2.0. *Technical report*, Dep. SIP, UCM Madrid, January 2001.

[2] S. Antoy. Definitional trees. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92)*, volume 632 of Springer LNCS, pages 143-157, 1992.

[3] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of ALP'97*, pages 16-30. Springer LNCS 1298, 1997.

[4] S. Antoy. Constructor-based conditional narrowing. In *Proc. PPDP'01*, ACM Press, pp. 199-206, 2001.

[5] S. Antoy, R. Echahed, M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776-822, 2000.

[6] F. Baader, T. Nipkow. Term Rewriting and All That. *Cambridge University Press*, 1998.

[7] J. C. González-Moreno, F.J.López-Fraguas, M.T. Hortalá-González, M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40:47-87, 1999.

[8] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583-628, 1994.

[9] M. Hanus (ed.). Curry: An Integrated FunctionalLogic Language. Version 0.7.1, June 2000. Available at http://www.informatik.uni-kiel.de/curry/report.html.

[10] M. Hanus, C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33-75, 1999.

[11] M. Hanus, S. Lucas, A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters (Elsevier)*, vol. 67, nº 1, pp. 1-8, 1998.

[12] G. Huet, J. J. Lévy. Computations in orthogonal term rewriting systems I, II. In J. L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honour of J. Alan Robinson*, pages 395-414 and 415-443. The MIT Press, 1991.

[13] H. Hussmann. Nondeterministic Algebraic Specification and non confluent term rewriting. *Journal of Logic Programming*, 12:237-255, 1992.

[14] T. Ida, K. Nakahara. Leftmost outside-in narrowing calculi. *Journal of Functional Programming*, 7(2):129-161, 1997.

[15] R. Loogen, F. J. López-Fraguas, M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, vol. 714 of Springer LNCS pp. 184-200, 1993.

[16] A. Middeldorp. Call by Need Computations to Root-Stable Form. In *Proc. POPL'1997*, ACM Press, 1997.

[17] A. Middeldorp, S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25 (6), pp. 733-757, 1998.

[18] R. del Vado Vírseda. Estrategias de Estrechamiento Perezoso. A *Master's Thesis* Directed by Dr. M. Rodríguez-Artalejo. Dpto. Sistemas Informáticos y Programación. Facultad de CC. Matemáticas, U.C.M. September 2002.

Narrowing-based Simulation of Term Rewriting Systems with Extra Variables and its Termination Proof

Naoki Nishida[1], Masahiko Sakai[2], and Toshiki Sakabe[2]

[1] Graduate School of Engineering, Nagoya University
[2] Graduate School of Information Science, Nagoya University,
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan
`nishida@sakabe.nuie.nagoya-u.ac.jp`
`{sakai,sakabe}@is.nagoya-u.ac.jp`

Abstract. Term rewriting systems (TRSs) are extended by allowing to contain extra variables in their rewrite rules. We call the extended systems EV-TRSs. They are ill-natured since every one-step reduction by their rules with extra variables is infinitely branching and they are not terminating. To solve these problems, this paper extends narrowing on TRSs into that on EV-TRSs and show that it simulates the reduction sequences of EV-TRSs as the narrowing sequences starting from ground terms. We prove the soundness of ground narrowing-sequences for the reduction sequences. We prove the completeness for the case of right-linear systems, and also for the case that no redex in terms substituted for extra variables is reduced in the reduction sequences. Moreover, we give a method to prove the termination of the simulation, extending the termination proof of TRSs using dependency pairs, to that of narrowing on EV-TRSs starting from ground terms.

1 Introduction

An extra variable is a variable appearing only in the right-hand side of a rewrite rule. Term rewriting systems (TRSs) are extended by allowing to contain extra variables in their rewrite rules. We call the extended systems *EV-TRSs*, especially *proper* EV-TRSs if they contain at least one extra variable. Proper EV-TRSs are ill-natured since every one-step reduction by their rules with extra variables is infinitely branching even up to renaming and none of them are terminating.

On the other hand, as a transformational approach to inverse computation of term rewriting, we have recently proposed an algorithm to generate a program computing the inverses of the functions defined by a given constructor TRS [10]. Unfortunately, the algorithm produces EV-TRSs in general. This fact gives rise to necessity of a simulation method of EV-TRSs.

This paper shows how to simulate the reduction sequences of EV-TRSs, and discusses the termination of the simulation. We first extend narrowing [6] on TRSs to that on EV-TRSs, restricting the substitutions for extra variables. In case of TRSs, the narrowing derivations starting from ground terms is just equivalent to the reduction sequences. This fact leads us to simulate the reduction sequences by the ground narrowing-sequences which are narrowing derivations starting from ground terms. Such a simulation solves the infinitely-branching problem, and terminates for some proper EV-TRSs. We prove the soundness of the ground narrowing-sequences for the reduction sequences of EV-TRSs. Then, we prove the completeness for the case of right-linear systems, and also for case that no redex in the terms substituted for extra variables is reduced in the reduction sequences. One of the typical instances of the latter case is a sequence constructed by substituting normal forms for extra variables. As a technique to prove termination of the proposed simulation, we extend the termination proof technique of TRSs using dependency pairs, proposed by T. Arts and J. Giesl [1], to that of narrowing on EV-TRSs (starting from ground terms).

This paper is organized as follows. In Section 3, we explain the idea for simulating EV-TRSs, define narrowing on EV-TRSs and prove the soundness and completeness. In Section 4, we discuss the termination of the simulations, i.e., narrowing starting from ground terms. Section 5 compares our results with the related works. We give the proofs of the theorems in the appendix.

2 Preparation

This paper follows the general notation of term rewriting [2, 7]. In this section, we briefly describe the notations used in this paper.

Let $\mathcal{F}$ be a signature and $\mathcal{X}$ be a countably infinite set of variables. The set of *terms* over $\mathcal{F}$ and $\mathcal{X}$ is denoted by $T(\mathcal{F}, \mathcal{X})$. The set $T(\mathcal{F}, \emptyset)$ of *ground terms* is simply written as $T(\mathcal{F})$. For a function symbol f, $\mathtt{arity}(f)$ denotes the number of arguments of f. The identity of terms s and t is denoted by $s \equiv t$. The set of variables in terms t_1,..., t_n is represented as $\mathcal{V}ar(t_1, ..., t_n)$. The top symbol of a term t is denoted by $\mathtt{top}(t)$.

We use $\mathcal{O}(t)$ to denote the set of all positions of term t, and $\mathcal{O}_{\mathcal{F}}(t)$ and $\mathcal{O}_{\mathcal{X}}(t)$ to denote the set of function symbol positions and variable positions of t, respectively. For $p, q \in \mathcal{O}(t)$, we write $p \leq q$ if there exists p' satisfying $pp' = q$. The subterm at a position $p \in \mathcal{O}(t)$ is represented by $t|_p$. We use contexts with exactly one-hole $\Box$. When we explicitly write positions of $\Box$ in a context C, we write $C[\ ]_p$ where $p \in \mathcal{O}(C)$ and $C|_p \equiv \Box$. The notation $u \trianglelefteq t$ means that u is a subterm of t.

A *substitution* is a mapping σ from $\mathcal{X}$ to $T(\mathcal{F}, \mathcal{X})$ such that $\sigma(x) \not\equiv x$ for finitely many $x \in \mathcal{X}$. We use σ, δ and θ to denote substitutions. Substitutions are naturally extended to mappings from $T(\mathcal{F}, \mathcal{X})$ to $T(\mathcal{F}, \mathcal{X})$ and $\sigma(t)$ is often written as $t\sigma$. We call $t\sigma$ an *instance* of t. The *composition* of σ and θ, denoted by $\sigma\theta$, is defined as $x\sigma\theta = \theta(\sigma(x))$. The *domain* and *range* of σ are defined as $\mathcal{D}om(\sigma) = \{ x \in \mathcal{X} \mid x\sigma \not\equiv x \}$ and $\mathcal{R}an(\sigma) = \{ x\sigma \mid x \in \mathcal{D}om(\sigma) \}$, respectively. The set of variables occurring in a term of $\mathcal{R}an(\sigma)$ is denoted by $\mathcal{VR}an(\sigma)$, i.e., $\mathcal{VR}an(\sigma) = \bigcup_{t \in \mathcal{R}an(\sigma)} \mathcal{V}ar(t)$. We write $\{x_1 \mapsto t_1,\ ...,\ x_n \mapsto t_n\}$

as σ if $\mathcal{D}om(\sigma) = \{x_1, ..., x_n\}$ and $x_i\sigma \equiv t_i$ for each i, and write $\emptyset$ instead of σ if $\mathcal{D}om(\sigma) = \emptyset$. We write $\sigma = \theta$ if $\mathcal{D}om(\sigma) = \mathcal{D}om(\theta)$ and $\sigma(x) \equiv \theta(x)$ for all $x \in \mathcal{D}om(\sigma)$. The *restriction* of σ to $X \subseteq \mathcal{X}$ is denoted by $\sigma|_X$, i.e., $\sigma|_X = \{x \mapsto t \mid x \in \mathcal{D}om(\sigma) \cap X, x\sigma \equiv t\}$. We write $\sigma \lesssim \sigma'$ if there exists θ satisfying $\sigma\theta = \sigma'$.

A *rewrite rule* is a pair (l, r), written as $l \to r$, where l ($\notin \mathcal{X}$) and r are terms. It may have a unique label ρ and be written as $\rho : l \to r$. Variables appearing only in the right-hand side of the rule ρ is called *extra variables* and the set of them are denoted by $\mathcal{EV}ar(\rho)$. An *EV-TRS* is a finite set of rewrite rules. Especially, it is called a *term rewriting system* if every rewrite rule $l \to r$ satisfies $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. Let R be an EV-TRS. The *reduction relation* $\to_R$ is a binary relation on terms defined by $\to_R = \{(C[l\sigma], C[r\sigma]) \mid C \text{ is a context}, l \to r \in R\}$. When we explicitly specify the position p and the rule ρ in $s \to_R t$, we write $s \to_R^{[p,\rho]} t$ or $s \to_R^p t$. As usual $\xrightarrow{*}_R$ and $\xrightarrow{n}_R$ are a reflexive and transitive closure of $\to_R$, and the n-step reduction of $\to_R$, respectively. We call $s_0 \to_R s_1 \to_R \cdots$ a *reduction sequence* of R. R is said to be *right-linear* if the right-hand side of every rule is linear. A term t is *terminating* (*SN*, for short) with respect to R if there is no infinite reduction sequence of R starting from t. R is *terminating* (*SN*, for short) if every term is terminating with respect to R.

Terms s and t are *variants* if s and t are instances of each other. Letting $\to$ be a binary relation on terms, $\to$ is *finitely branching* if a set $\{\, t \mid s \to t \,\}$ is finite up to renaming for any term s. Otherwise, it is *infinitely branching.* Let R be an EV-TRS. If R is a TRS then $\to_R$ is finitely branching. Otherwise, however, it is infinitely branching in general.

Example 1. The following R_1 is an EV-TRS;

$$R_1 = \{\ f(x, 0) \to s(x),\ \ g(x) \to h(x, y),\ \ h(0, x) \to f(x, x),\ \ a \to b\ \}.$$

A term $g(0)$ can be reduced by the second rule above to any of terms, $h(0, 0)$, $h(0, g(0))$, $h(0, f(0, 0))$ and so on. Thus, $\to_{R_1}$ is infinitely branching. □

3 Simulation by Ground Narrowing-sequences

In this section, we first explain the intuitive idea for simulating EV-TRSs. Then, we extend narrowing [6] on TRSs to that on EV-TRSs, and show how to simulate the reduction sequences of EV-TRSs using the ground narrowing-sequences. We prove the soundness of the ground narrowing-sequences for the reduction sequences of EV-TRSs, Then, we prove the completeness for the case of right-linear systems, and also for the EV-safe reduction sequences defined later.

Consider the sequences starting from $g(0)$ by R_1 in Example 1. Substituting a fresh variable for each extra variable makes it possible to focus on only a term such as $h(0, z)$ to which $g(0)$ is reduced by the second rule. In addition, narrowing enables z to act as an arbitrary term. Then, narrowing with such a restriction constructs the sequence in Fig.1 (a) which represents the infinitely many possibly infinite sequences in Fig.1 (b). That's why we extend narrowing

$$\begin{tikzcd}
\text{(a)} & g(0) \arrow[r, rightsquigarrow, "R_1"'] & h(0,z) \arrow[r, rightsquigarrow, "R_1"'] & f(z,z) \arrow[r, rightsquigarrow, "\{z\mapsto 0\}\ R_1"'] & s(0) \\
\text{(b)} & & h(0,a) \arrow[r, "R_1"'] & \cdots & \\
& g(0) \arrow[ur, "R_1"] \arrow[r, "R_1"'] \arrow[dr, "R_1"] & h(0,0) \arrow[r, "R_1"'] & f(0,0) \arrow[r, "R_1"'] & s(0) \\
& & h(0,g(a)) \arrow[r, "R_1"'] & \cdots & \\
& \vdots & \vdots & \ddots &
\end{tikzcd}$$

Fig. 1. (a) Simulation we propose, and (b) the sequences we simulate.

to simulate EV-TRSs.

On the other hand, since variables in the reduction sequences (not in narrowing sequences) act as constants, it is enough to treat terms in the sequences as ground terms by introducing new constants. For instance, the ground term $f(c_x, 0)$ represents the term $f(x, 0)$ with variable x. Thus, considering the reduction sequences starting from only ground terms covers those starting from any terms.

Next, we define narrowing on EV-TRSs, following the idea above. A *unifier* of terms s and t is a pair (σ, σ') of substitutions such that $s\sigma \equiv t\sigma'$[3]. The *most general unifier* of s and t, denoted by $\mathtt{mgu}(s,t)$, is a unifier (σ, σ') of s and t such that $\sigma \lesssim \theta$ and $\sigma' \lesssim \theta'$ for all unifier (θ, θ') of s and t.

Definition 1. *Let R be an EV-TRS. A term s is said to be* narrowable *into a term t with a substitution δ, a position $p \in \mathcal{O}(s)$ and a rewrite rule $\rho \in R$, written as $s \underset{\delta}{\rightsquigarrow}{}^{[p,\rho]}_{R}\, t$, if there exist a context C, a term u and a substitution σ such that*

(a) $p \in \mathcal{O}_{\mathcal{F}}(s)$, $s \equiv C[u]_p$, $t \equiv C\delta[r\sigma]_p$, $(\delta, \sigma) = \mathtt{mgu}(u, l)$ and $\mathcal{VR}an(\delta) \cap (\mathcal{V}ar(s) \setminus \mathcal{D}om(\delta)) = \emptyset$,
(b) $x\sigma \in (\mathcal{X} \setminus \mathcal{V}ar(s, u\delta))$ for all $x \in \mathcal{EV}ar(\rho)$, and
(c) $x \not\equiv y$ implies $x\sigma \not\equiv y\sigma$ for any $x, y \in \mathcal{EV}ar(\rho)$,

where $\rho : l \to r$. We assume $\mathcal{D}om(\delta) \subseteq \mathcal{V}ar(u)$. We call $\rightsquigarrow_R$ narrowing *by R. Note that p and ρ may be omitted like as $s \underset{\delta}{\rightsquigarrow}_R t$ or $s \underset{\delta}{\rightsquigarrow}{}^{p}_{R}\, t$.*

The above extension is done by adding the conditions (b) and (c) on extra variables to the ordinary definition of narrowing. We write $s \underset{\delta}{\overset{n}{\rightsquigarrow}}_R t$ or $s \underset{\delta}{\overset{*}{\rightsquigarrow}}_R t$ if there exists a narrowing derivation $s \equiv t_0 \underset{\delta_0}{\rightsquigarrow}_R t_1 \underset{\delta_1}{\rightsquigarrow}_R \cdots \underset{\delta_{n-1}}{\rightsquigarrow}_R t_n \equiv t$, called *narrowing sequence*, such that $\delta = \delta_0\delta_1 \cdots \delta_{n-1}$. If $n = 0$ then $\delta = \emptyset$. Note that δ may be omitted like as $s \overset{n}{\rightsquigarrow}_R t$ or $s \overset{*}{\rightsquigarrow}_R t$ if $\delta = \emptyset$. Especially, narrowing sequences starting from ground terms are said to be *ground*. From the definition

[3] Usually, unifiers are defined as single substitutions under the condition with no common variable in both terms. But we define unifiers as pairs of substitutions to eliminate renaming variables of rewrite rules in narrowing and to simplify treatments of variables in the proofs of theorems.

$$
\begin{array}{lll}
R_2 = \{ & add^{\#}(y) \to tp_2(0,y), & add^{\#}(s(z)) \to u_1(add^{\#}(z)), \\
& u_1(tp_2(x,y)) \to tp_2(s(x),y), & add^{\#}(add(x,y)) \to tp_2(x,y), \\
& mul^{\#}(0) \to tp_2(0,y), & mul^{\#}(0) \to tp_2(x,0), \\
& mul^{\#}(s(z)) \to u_2(add^{\#}(z)), & u_2(tp_2(w,y)) \to u_3(mul^{\#}(w),y), \\
& u_3(tp_2(x,s(y)),y) \to tp_2(s(x),s(y)), & mul^{\#}(mul(x,y)) \to tp_2(x,y) \quad \}.
\end{array}
$$

Fig. 2. The EV-TRS computing the inverses of addition and multiplication.

of narrowing, it is clear that $\underset{\delta}{\rightsquigarrow}_R$ are finitely branching for any EV-TRS R. It is also clear that $\to_R \; = \; \rightsquigarrow_R$ on ground terms for any TRS R.

Here, we show an example of the simulation by narrowing on EV-TRSs.

Example 2. The system computing inverse images $add^{\#}$ and $mul^{\#}$ of addition and multiplication, respectively, of two natural numbers is resulted in the EV-TRS seen in Fig.2 [10]. Considering the narrowing sequences starting from $mul^{\#}(s^4(0))$, there exist only 16 finite-paths up to renaming. This means that all solutions of $mul^{\#}(s^4(0))$ are found in finite time and space. One of such paths is as follows;

$$
\begin{array}{l}
mul^{\#}(s^4(0)) \overset{*}{\rightsquigarrow}_{R_2} u_3(u_3(mul^{\#}(0), s(0)), s(0)) \rightsquigarrow_{R_2} u_3(u_3(tp_2(0,y), s(0)), s(0)) \\
\qquad \underset{\{y \mapsto s^2(0)\}}{\rightsquigarrow}_{R_2} u_3(tp_2(s(0), s^2(0)), s(0)) \rightsquigarrow_{R_2} tp_2(s^2(0), s^2(0)). \qquad \square
\end{array}
$$

The following theorem shows the soundness whose proof is similar to that on TRSs [6].

Theorem 1. *Let R be an EV-TRS. For all $s \in T(\mathcal{F})$, $t \in T(\mathcal{F}, \mathcal{X})$ and a substitution δ, $s \underset{\delta}{\overset{*}{\rightsquigarrow}}_R t$ implies $s\delta \overset{*}{\to}_R t$.*

Here, we introduce the notion of EV-safety of the reduction sequences. We say that a reduction sequence is *EV-safe* if no redex in the terms substituted for extra variables is reduced in that sequence. In the beginning of the appendix, a precise definition of this notion is found. Followings are results on the completeness.

Theorem 2. *Let R be an EV-TRS. For all $s, t \in T(\mathcal{F})$, the EV-safe sequence $s \overset{*}{\to}_R t$ implies a term t' and a substitution θ such that $s \underset{\delta}{\overset{*}{\rightsquigarrow}}_R t'$ and $t \equiv t'\theta$.*

Theorem 3. *Let R be a right-linear EV-TRS. For all $s, t \in T(\mathcal{F})$, $s \overset{*}{\to}_R t$ implies a linear term t' and a substitution θ such that $s \underset{\delta}{\overset{*}{\rightsquigarrow}}_R t'$ and $t \equiv t'\theta$.*

Since variables in target terms of reduction can be considered as constants and the simulation of EV-TRSs is done by ground narrowing-sequences, we focused on the ground reduction sequences in the above theorems.

The following example shows that the completeness does not hold in general, and also shows that the conditions in Theorem 2 and 3 are essential and necessary.

Example 3. Consider the sequence starting from $g(0)$ by R_1 in Example 1 again. We have the non-EV-safe reduction-sequence $g(0) \to_{R_3} h(0,a) \to_{R_3} f(a,a) \to_{R_3} f(a,b)$. However, this sequence cannot be simulated by narrowing. In fact, $g(0)$ is not narrowable to any term matchable with $f(a,b)$. $\square$

4 Termination of EV-TRSs' Simulation

In order to give the termination of the simulation by ground narrowing-sequences, we extend the termination proof technique of TRSs using dependency pairs, proposed by T. Arts and J. Giesl [1], to that of narrowing starting from ground terms.

Let R be an EV-TRS and t be a term. We say that t is *N-SN* with respect to R if there exists no infinite narrowing sequence starting from it. R is said to be *N-SN* if all terms are N-SN with respect to R. R is *N-GSN* if all ground terms are N-SN with respect to R. Since the proposed simulation is done by narrowing starting from ground terms, it is enough to consider N-GSN. Conversely, since most of EV-TRSs (even TRSs) are not N-SN, results on N-SN have very restrictive power. The following proposition holds obviously because the ground narrowing and reduction sequences on TRSs are equivalent.

Proposition 1. *A TRS is terminating if and only if it is N-GSN.*

The following proposition associated with N-SN and N-GSN also holds obviously.

Proposition 2. *If an EV-TRS is N-SN then it is also N-GSN.*

The converse of the above does not hold. For example, considering a TRS $R_4 = \{\ d(0) \to 0,\ \ d(s(x)) \to s(s(d(x)))\ \}$, this is N-GSN but not N-SN.

Let R be an EV-TRSs over a signature $\mathcal{F}$. The set of *defined symbols* of R is defined as $\mathcal{D}_R = \{\ \mathtt{top}(l) \mid l \to r \in R\ \}$, and the set of *constructors* of R as $\mathcal{C}_R = \mathcal{F} \setminus \mathcal{D}_R$. To define the dependency pairs, we prepare a fresh function symbol F not in a signature $\mathcal{F}$ for each defined symbol f. We call F the *capital symbol* of f. This paper uses small letters for function symbols in $\mathcal{F}$ and uses the string obtained by replacing the first letter of a defined symbol with the corresponding capital letter. For example, we use *Abc* as the capital symbol of a defined symbol *abc*. The set $\overline{\mathcal{D}}_R$ of capital symbols determined by the set $\mathcal{D}_R$ is defined as $\overline{\mathcal{D}}_R = \{F \mid f \in \mathcal{D}_R\}$. Moreover, we define $\overline{\mathcal{F}} = \mathcal{F} \cup \overline{\mathcal{D}}_R$. The definition of dependency pairs of EV-TRSs is the same as that of TRSs.

Definition 2. *Let R be an EV-TRS. The pair $\langle F(s_1, ..., s_n), G(t_1, ..., t_m)\rangle$ is called a* dependency pair *of R if there are a rewrite rule $f(s_1, ..., s_n) \to r \in R$ and a subterm $g(t_1, ..., t_m) \trianglelefteq r$ with $g \in \mathcal{D}_R$. The set of all dependency pairs of R is denoted by $\mathcal{DP}_R$.*

Let $\langle s, t\rangle \in \mathcal{DP}_R$. We also call variables only in t (i.e.,in $\mathcal{V}ar(t) \setminus \mathcal{V}ar(s)$) extra variables, and write $\mathcal{EV}ar(\langle s, t\rangle)$ as the set of all extra variables of $\langle s, t\rangle$.

4.1 Termination Proof of Reduction of TRSs

We first explain briefly the main part of the termination proof technique of TRSs' reduction using dependency pairs [1]. The notion of chain is as follows.

Definition 3 ([1]). *Let R be a TRS and $\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, ... \in \mathcal{DP}_R$. The sequence $\langle s_1, t_1\rangle\langle s_2, t_2\rangle \cdots$ of dependency pairs is called an $R\&\mathcal{DP}_R$-*reduction-chain *(R-*chain*, for short) if there exist a term $s'_i \in T(\overline{\mathcal{F}}, \mathcal{X})$ and a substitution σ_i such that $t_i\sigma_i \xrightarrow{*}_R s'_{i+1}$ and $s_i\sigma_i \equiv s'_i$ for each $i > 0$.*

Then, there is the following theorem that relates between non-existence of infinite chains and termination of TRSs.

Theorem 4 ([1]). *A TRS R is SN if and only if there is no infinite R-chain.*

To guarantee the non-existence of infinite chains, the following theorem is useful.

Theorem 5 ([1]). *Let R be a TRS over a signature $\mathcal{F}$. There is no infinite R-chain if and only if there is a quasi-reduction order*[4] $\succsim$ *on* $T(\overline{\mathcal{F}}, \mathcal{X})$ *such that*

- $l \succsim r$ *for every rule* $l \to r \in R$, *and*
- $s \succ t$ *for every dependency pair* $\langle s, t \rangle \in \mathcal{DP}_R$.

To find such an order, argument filtering functions are used.

Definition 4 ([8]). *An* argument filtering (AF) *is a function π such that for any $f \in \mathcal{F}$, $\pi(f)$ is either an integer i or a list $[i_1, ..., i_m]$ of integers where $0 \leq m, i \leq \mathtt{arity}(f)$ and $1 \leq i_1 < \cdots < i_m \leq \mathtt{arity}(f)$. Note that we assume $\pi(f) = [1, ..., n]$, where $n = \mathtt{arity}(f)$, if $\pi(f)$ is not defined. and also for each capital symbol. We can naturally extend π over terms as follows;*

- $\pi(x) = x$ *where* $x \in \mathcal{X}$,
- $\pi(f(t_1, ..., t_n)) = \pi(t_i)$ *where* $\pi(f) = i$,
- $\pi(f(t_1, ..., t_n)) = f(\pi(t_{i_1}), ..., \pi(t_{i_m}))$ *where* $\pi(f) = [i_1, ..., i_m]$.

Moreover, π is extended over an EV-TRS R and the set of its dependency pairs as $\pi(R) = \{\pi(l) \to \pi(r) \mid l \to r \in R\}$ and $\pi(\mathcal{DP}_R) = \{\langle \pi(s), \pi(t) \rangle \mid \langle s, t \rangle \in \mathcal{DP}_R\}$.

This paper assumes that $\pi(f)$ is not an integer but in form $[i_1, ..., i_m]$ for any defined symbol f, and also for any capital symbol. We say that such an AF function is *simple*. Order using AF function π is defined as follows;

- $s \succsim_\pi t$ if and only if $\pi(s) \succ \pi(t)$ or $\pi(s) \equiv \pi(t)$, and
- $s \succ_\pi t$ if and only if there are a contest C such that $\pi(s) \succ C[\pi(t)]$, or $C \not\equiv \Box$ and $\pi(s) \equiv C[\pi(t)]$.

Dependency graphs [1] combined with the above techniques are more powerful.

Example 4. Consider the following TRS;

$$\begin{aligned} R_5 = \{ & \mathit{minus}(x, 0) \to x, \quad \mathit{minus}(s(x), s(y)) \to \mathit{minus}(x, y), \\ & \mathit{quot}(0, s(y)) \to 0, \quad \mathit{quot}(s(x), s(y)) \to s(\mathit{quot}(\mathit{minus}(x, y), s(y))) \}. \end{aligned}$$

The dependency pairs of R_5 are as follows;

$$\begin{aligned} \mathcal{DP}_{R_5} = \{ & \langle M(s(x), s(y)), M(x, y) \rangle, \ \langle Q(s(x), s(y)), M(x, y) \rangle, \\ & \langle Q(s(x), s(y)), Q(\mathit{minus}(x, y), s(y)) \rangle \qquad \}, \end{aligned}$$

[4] A *quasi-order* $\succsim$ is a reflexive and transitive relation, and $\succsim$ is called *well-founded* if its strict part $\succ$ is well-founded. A *quasi-reduction order* $\succsim$ is a well-founded quasi-order such that $\succsim$ is compatible with contexts and both $\succsim$ and $\succ$ are closed under substitutions.

$$\begin{array}{cccccccc}
 & \langle\, s_1,\ t_1 \,\rangle & & \langle\, s_2,\ t_2 \,\rangle & & \langle\, s_3,\ t_3 \,\rangle & & \cdots \\
 & \begin{array}{c}(\delta_1,\sigma_1) = \vdots\\ \mathtt{mgu}(s'_1, s_1)\end{array} \quad \vdots & & \begin{array}{c}(\delta_2,\sigma_2) = \vdots\\ \mathtt{mgu}(s'_2, s_2)\end{array} \quad \vdots & & \begin{array}{c}(\delta_3,\sigma_3) = \vdots\\ \mathtt{mgu}(s'_3, s_3)\end{array} \quad \vdots & & \\
\left(T(\overline{\mathcal{F}}) \ni \exists s_0 \overset{*}{\underset{\delta'_0}{\rightsquigarrow}}_R\right) & s'_1 \quad t_1\sigma_1 & \overset{*}{\underset{\delta'_1}{\rightsquigarrow}}_R & s'_2 \quad t_2\sigma_2 & \overset{*}{\underset{\delta'_2}{\rightsquigarrow}}_R & s'_3 \quad t_3\sigma_3 & \overset{*}{\underset{\delta'_3}{\rightsquigarrow}}_R & \cdots
\end{array}$$

Fig. 3. A (ground) R-narrowing-chain.

where M and Q are abbreviations of $Minus$ and $Quot$, respectively. Let π_5 be an AF function with $\pi_5(minus) = [1]$. Then, the following inequalities are satisfied by the recursive path order (rpo) with a precedence $quot > s > m$ and $Q > M$;

$$\begin{array}{rlrl}
minus(x) & \gtrsim_{\pi_5} x, & minus(x(x)) & \gtrsim_{\pi_5} minus(x), \\
quot(0, s(y)) & \gtrsim_{\pi_5} 0, & quot(s(x), s(y)) & \gtrsim_{\pi_5} s(quot(minus(x), s(y))), \\
M(s(x), s(y)) & \gtrsim_{\pi_5} M(x, y), & Q(s(x), s(y)) & \gtrsim_{\pi_5} Q(minus(x), s(y)), \\
Q(s(x), s(y)) & \gtrsim_{\pi_5} M(x, y). & &
\end{array}$$

Therefore, R_5 is terminating by Theorem 4, 5. □

4.2 Termination Proof of Narrowing on EV-TRSs

Here, we extend the termination proof in Subsection 4.1 to that of narrowing.

We first extend the chain constructed by reduction to that by narrowing.

Definition 5. *Let R be an EV-TRS and $\langle s_1, t_1\rangle, \langle s_2, t_2\rangle, ... \in \mathcal{DP}_R$. The sequence $\langle s_1, t_1\rangle\langle s_2, t_2\rangle \cdots$ of dependency pairs is called an $R\&\mathcal{DP}_R$-*narrowing-chain *(R-*narrowing-chain*, for short) if there exist a term $s'_i \in T(\overline{\mathcal{F}}, \mathcal{X})$ and the most general unifier $(\delta_i, \sigma_i) = \mathtt{mgu}(s'_i, s_i)$ such that $t_i\sigma_i \overset{*}{\underset{\delta'_i}{\rightsquigarrow}}_R s'_{i+1}$ for each $i > 0$ where $x\sigma_i$ is a fresh variable for all $x \in \mathcal{EV}ar(\langle s_i, t_i\rangle)$ (see Fig.3). Especially, it is said to be* ground*, written as $s_0\langle s_1, t_1\rangle\langle s_2, t_2\rangle \cdots$, if there exists some ground term $s_0 \in T(\overline{\mathcal{F}})$ such that $s_0 \overset{*}{\underset{\delta'_0}{\rightsquigarrow}}_R s'_1$.*

Then, we obtain the following theorem that corresponds to Theorem 4.

Theorem 6. *Let R be an EV-TRS.*

(a) R is N-SN if and only if there is no infinite R-narrowing-chain.
(b) R is N-GSN if and only if there is no infinite ground R-narrowing-chain.

In the case of the termination proof of TRSs, we can do it by finding a reduction order to ensure no infinite chain. However, it is difficult to find such a order on proper EV-TRSs. Hence, we use AF functions again to eliminate extra variables.

Let R be an EV-TRS and π be an AF function. We say that π *eliminates all extra variables* of R and $\mathcal{DP}_R$ if $\mathcal{V}ar(\pi(l)) \supseteq \mathcal{V}ar(\pi(r))$ for all rules $l \to r \in R$ and $\mathcal{V}ar(\pi(s)) \supseteq \mathcal{V}ar(\pi(t))$ for all dependency pairs $\langle s, t\rangle \in \mathcal{DP}_R$. We extend

$$\begin{tikzcd}[column sep=small]
 & & \langle \pi(s_1) \arrow[d, dotted, no head, "{(\theta_1,\sigma_1)=\mathtt{mgu}(s'_1,\pi(s_1))}"'] & \pi(t_1)\rangle \arrow[d, dotted, no head] & \langle \pi(s_2) \arrow[d, dotted, no head, "{(\theta_2,\sigma_2)=\mathtt{mgu}(s'_2,\pi(s_2))}"'] & \pi(t_2)\rangle \arrow[d, dotted, no head] & \cdots \\
\Bigl(T(\overline{\mathcal{F}}) \ni \exists s_0 \arrow[rr, squiggly, "*", "\delta'_0\ R"'] & & \Bigr)\, s'_1 & \pi(t_1)\sigma_1 \arrow[r, squiggly, "*", "\delta'_1\ \pi(R)"'] & s'_2 & \pi(t_2)\sigma_2 \arrow[r, squiggly, "*", "\delta'_2\ \pi(R)"'] & \cdots
\end{tikzcd}$$

Fig. 4. A (ground) $\pi(R\&\mathcal{DP}_R)$-narrowing-chain.

the notion of (ground) $R\&\mathcal{DP}_R$-narrowing-chains into (ground) $\pi(R\&\mathcal{DP}_R)$-*narrowing-chains*, which is obtained by replacing R and $\mathcal{DP}_R$ with $\pi(R)$ and $\pi(\mathcal{DP}_R)$, respectively, in Definition 5 (see Fig.4).

No infinite $\pi(R\&\mathcal{DP}_R)$-narrowing-chain gives us the following theorem.

Theorem 7. *Let R be an EV-TRS and π be a simple AF function that eliminates all extra variables of R and $\mathcal{DP}_R$. If there exists no infinite $\pi(R\&\ \mathcal{DP}_R)$-narrowing-chain then R is N-GSN. Moreover, if $\pi(t)$ is a ground term for all $\langle s,t\rangle \in \mathcal{DP}_R$ then R is N-SN.*

If there is no extra variable in $\pi(R)$ and $\pi(\mathcal{DP}_R)$, we can check whether an infinite $\pi(R\&\mathcal{DP}_R)$-narrowing-chain exists, by using the termination proof techniques in Subsection 4.1 similarly to the case of TRSs. Note that this theorem is also usable to check whether ordinary narrowing of TRSs is terminating, although the result is very restrictive and narrowing sequences seldom terminate. For example, even a simple TRS $\{\ f(s(x)) \to f(x)\ \}$ which terminates is not N-SN, since term $f(y)$ with variable y leads to an infinite narrowing sequence.

Example 5. Consider R_1 in Example 1 again. The set of its dependency pairs is $\mathcal{DP}_{R_1} = \{\ \langle G(x), H(x,y)\rangle,\ \ \langle H(0,x), F(x,x)\rangle\ \}$.

The sequence $\langle G(x), H(x,y)\rangle\langle H(0,x), F(x,x)\rangle$ is an R_1-narrowing-chain and $G(0)\langle G(x), H(x,y)\rangle\langle H(0,x), F(x,x)\rangle$ is the ground one.

Let π_1 be a simple AF function with $\pi_1(h) = \pi_1(H) = \pi_1(s) = \pi_1(g) = \pi_1(G) = \pi_1(f) = \pi_1(F) = [\,]$. Then, we have $\pi_1(R_1) = \{\ f \to s,\ \ g \to h,\ \ h \to f,\ \ a \to b\ \}$ and $\pi_1(\mathcal{DP}_{R_1}) = \{\ \langle G, H\rangle,\ \ \langle H, F\rangle\ \}$. It is clear that no infinite $\pi_1(R_1\&\mathcal{DP}_{R_1})$-narrowing-chain exists. Moreover, all right-hand sides of dependency pairs are ground. Therefore, R_1 is N-SN. □

Example 6. Consider the EV-TRS $R_6 = \{\ a \to d(c(y))\ \} \cup R_4$. Let π_6 be an AF function with $\pi_6(c) = [\,]$. Here, we have $\pi_6(R_6) = \{a \to d(c), d(0) \to 0, d(s(x)) \to s^2(d(x))\}$ and $\pi_6(\mathcal{DP}_{R_6}) = \{\langle A, D(c)\rangle, \langle D(s(x)), D(x)\rangle\}$. The inequalities $a \gtrsim_{\pi_6} d(c)$, $d(0) \gtrsim_{\pi_6} 0$, $d(s(x)) \gtrsim_{\pi_6} d(x)$, $A \succ_{\pi_6} D(c)$, and $D(s(x)) \succ_{\pi_6} D(x)$ are satisfied by the rpo with $a > d$ and $A > D$. Therefore, R_6 is N-GSN. It is clear that R_6 is not N-SN. □

The following corollary is a little weaker but easier to use than Theorem 7.

Corollary 1. *Let R be an EV-TRS and π be a simple AF function that eliminates all extra variables of R and $\mathcal{DP}_R$ and satisfying $\pi(\mathcal{DP}_R) = \mathcal{DP}_{\pi(R)}$. If*

$\pi(R)$ is terminating then R is N-GSN. Moreover, if every subterm u of $\pi(r)$ with $\mathtt{top}(u) \in \mathcal{D}_R$ is ground for all $l \to r \in R$ then R is N-SN.

In Example 5, we have $\pi_1(\mathcal{DP}_{R_1}) = \mathcal{DP}_{\pi_1(R_1)}$, and the right-hand side of all rules in $\pi_1(R_1)$ is ground. Hence, Corollary 1 is usable to prove that R_1 is N-GSN and N-SN.

5 Related Works

In studies on normalizing reduction strategies [3, 4, 12, 11], several kinds of EV-TRSs as approximations of TRSs are used. Arbitral reduction systems [3] are formalized as EV-TRSs whose right-hand sides are extra variables. They introduced an Ω-reduction system to simulate the reduction sequence, which is a special case of narrowing extended in this paper. Although they have termination, the theorems in Section 4 does not work to show their termination. The reason is that argument filtering method in this paper cannot eliminate all extra variables of collapsing rules. To overcome this problem is one of future works.

There are some studies on narrowing of conditional TRSs (CTRSs) with extra variables [5, 9]. The targets of their results are 3-CTRSs, in which every extra variable must appear in condition parts. On the other hand, EV-TRSs are not 3-CTRSs but 4-CTRSs, CTRSs with no restrictions. In addition, the CTRS, from which our motivating EV-TRS R_2 in Fig.2 is obtained by transformation, is not 3-CTRS but 4-CTRS.

Acknowledgments

This work is partially supported by Grants from Ministry of Education, Science and Culture of Japan #15500007, the Intelligent Media Integration Project of Nagoya University as one of the 21st Century COE of JAPAN, and the Nitto Foundation.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, vol.236, pp.133–178, 2000.
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
3. G. Huet and J.-J. Lèvy. Call-by-need computations in ambiguous linear term rewriting systems. Technical report, INRIA, 1979.
4. G. Huet and J.-J. Lèvy. Computations in orthogonal rewriting systems, I and II. In J.-L.Lassez and G.Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pp.395–443. MIT Press, Cambridge, 1991.
5. M. Hanus. On extra variables in (equational) logic programming. In *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*, pp.665–679, Cambridge, 1995. MIT Press.

6. J.M. Hullot. Canonical forms and unification. In *Proceedings of the 5th International Conference on Automated Deduction, LNCS 87*, pp.318–334, 1980.
7. J.W. Klop. Term rewriting systems. In S.Abramsky, et al, editors, *Handbook of Logic in Computer Science*, vol.2, pp.2–116. Oxford University Press, New York, 1992.
8. K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proceedings of Principles and Practice of Declarative Programming (PPDP'99), LNCS 1702*, pp.47–61, 1999.
9. A. Middeldorp, T. Suzuki, and M. Hamada. Complete selection functions for a lazy conditional narrowing calculus. *Journal of Functional and Logic Programming 2002(3)*, 2002.
10. N. Nishida, M. Sakai, and T. Sakabe. Generation of a TRS implementing the inverses of the functions with specified arguments fixed. Technical report, the Institute of Electronics, Information and Communication Engineers (IEICE), COMP 2001-67, pp.33–40, 2001 (in Japanese).
11. M. Oyamaguchi. NV-sequentiality: a decidable condition for call-by-need computations in term rewriting systems. *SIAM Journal on Computing*, vol.22, no.1, pp.114–135, 1993.
12. T. Nagaya, M. Sakai, and Y. Toyama. NVNF-sequentiality of left-linear term rewriting systems. Technical report, RIMS, 1995.

A EV-safe Reduction Sequences on EV-TRSs

Here, we give a precise definition of the EV-safe reduction sequences on EV-TRSs. Let t be a term and x be a variable. The set of positions that x occurs in t is denoted by $\mathcal{O}_x(t)$; $\mathcal{O}_x(t) = \{ p \mid p \in \mathcal{O}_\mathcal{X}(t), t|_p \equiv x \}$. Let $P, Q \subseteq \mathcal{O}(t)$. We write $P \leq Q$ if for all $q \in Q$ there exists some $p \in P$ such that $p \leq q$. The set $P \backslash p$ is defined as $P \backslash p = \{ q \mid pq \in P \}$. The minimum set of P is defined as $min(P) = \{ p \mid p \in P, \neg(\exists q \in P, q < p) \}$. For example, $min(\{11, 1, 2\}) = \{1, 2\}$. We define the minimum set of union of P and Q as $P \sqcup Q = min(P \cup Q)$, and the minimum set of intersection of P and Q as $P \sqcap Q = \{ p \mid p \in min(P), (\ \exists q \in min(Q), q \leq p\ ) \} \cup \{ q \mid q \in min(Q), (\ \exists p \in min(P), p \leq q\ ) \}$. For example, $\{11, 22\} \sqcup \{112, 2, 31\} = \{11, 2, 31\}$ and $\{11, 22\} \sqcap \{112, 2, 31\} = \{112, 22\}$.

We give the notion of the transition of positions at one-step reduction, adding the positions of extra variables.

Definition 6. *Let a rewrite rule $\rho : l \to r$, let P be a set of positions and p be a position. We write $P \Rightarrow^{[p,\rho]} Q$ if there is no position q in P such that $q \leq p$, and Q satisfies the following;*

$$Q = \{ q \mid q \in P, p \not\leq q \} \sqcup (\textstyle\bigsqcup_{x \in \mathcal{EV}ar(\rho)} \{ pq \mid r|_q \equiv x \}) \sqcup (\textstyle\bigsqcup_{x \in \mathcal{V}ar(l)} \{ pqw \mid r|_q \equiv x, w \in (\textstyle\bigsqcap_{q' \in \mathcal{O}_x(l)} P \backslash pq') \}).$$

This notion of transition is similar to that of descendants that follows redex positions [4]. We use a set of positions, such as P and Q, to represent positions, under which reductions are prohibited. The notation of $P \Rightarrow^{[p,\rho]} Q$ shows the transition in the one-step reduction at the position p by the rule ρ. Now, we define EV-safety as follows.

Definition 7. *Let R be an EV-TRS, $\rho_i : l_i \to r_i \in R$. We say that the reduction sequence $s_0 \to_R^{[p_0,\rho_0]} s_1 \to_R^{[p_1,\rho_1]} \cdots$ is* EV-safe*, written as $P_0 : s_0 \to_R^{[p_0,\rho_0]} P_1 : s_1 \to_R^{[p_1,\rho_1]} \cdots$, if there are set $P_0, P_1, \ldots$ of positions such that (a) $P_0 = \emptyset$, and (b) for each $i \geq 0$, $P_{i+1} \subseteq \mathcal{O}(s_{i+1})$ and there exists $Q_{i+1} \subseteq \mathcal{O}(s_{i+1})$ with $P_i \Rightarrow^{[p_i,\rho_i]} Q_{i+1}$ and $P_{i+1} \leq Q_{i+1}$.*

Example 7. Consider R_1 in Example 1. The sequence $g(0) \to_{R_1} h(0,0) \to_{R_1} f(0,0) \to_{R_1} s(0)$ is EV-safe because of $\emptyset : g(0) \to_{R_1} \{2\} : h(0,0) \to_{R_1} \emptyset : f(0,0) \to_{R_1} \emptyset : s(0)$. On the other hand, the sequence $g(0) \to_{R_1} h(0,a) \xrightarrow{*}_{R_1} h(0,b)$ does not since the subterm a of $h(0,a)$ is reduced. □

B Proofs

Proof of Theorem 1 Let s and t be terms and δ be a substitution. We prove by induction on n that $s \overset{n}{\underset{\delta}{\rightsquigarrow}}_R t$ implies $s\delta \xrightarrow{*}_R t$.

Since the case of $n = 0$ is trivial, we assume that $s \overset{n-1}{\underset{\delta}{\rightsquigarrow}}_R u \underset{\delta'}{\rightsquigarrow}{}_R^{[p,\rho]} t$. Letting $\rho : l \to r \in R$, there exist a context C, a term u' and a substitution σ such that $u \equiv C[u']_p \underset{\delta'}{\rightsquigarrow}_R C\delta'[r\sigma]_p \equiv t$ and $(\delta', \sigma) = \mathtt{mgu}(u', l)$. From the definition of most general unifiers, we have $u'\delta' \equiv l\sigma$. We have $s\delta \xrightarrow{*}_R u$ by induction hypothesis. Then, it follows from the stability of reduction that $s\delta\delta' \xrightarrow{*}_R u\delta'$. Therefore, we have $s\delta\delta' \xrightarrow{*}_R u\delta' \equiv C[u']_p\delta' \equiv C\delta'[u'\delta']_p \equiv C\delta'[l\sigma]_p \to_R C\delta'[r\sigma]_p \equiv t$. □

We prepare the following lemmas. The following can be easily proved.

Lemma 1. *Let (σ, σ') be the most general unifier of terms s and t. For any unifier (θ, θ') of s and t, there exists a substitution δ such that $s\sigma\delta \equiv s\theta$ and $t\sigma'\delta \equiv t\theta'$.*

Lemma 2. *Let R be an EV-TRS. Let $P : s\theta \to_R P' : t$ and $P \leq \mathcal{O}_{\mathcal{X}}(s)$. Then, there is t' and θ' such that $s \underset{\delta}{\rightsquigarrow}_R t'$, $t \equiv t'\theta'$ and $P' \leq \mathcal{O}_{\mathcal{X}}(t')$.*

Proof. We assume that $s\theta \to_R^{[p,\rho]} t$ where $\rho : l \to r \in R$. Then, $s\theta \equiv C[l\sigma]_p$ and $t \equiv C[r\sigma]_p$, where we assume $\mathcal{D}om(\theta) \cap \mathcal{D}om(\sigma) = \emptyset$ and $\mathcal{D}om(\theta) = \mathcal{V}ar(s)$ without loss of generality.

From EV-safety, we have $p \in \mathcal{O}(s) \setminus \mathcal{O}_{\mathcal{X}}(s)$. Since there exist a context $C'[\,]_p$ and a term v such that $s \equiv C'[v]_p$, we have $s\theta \equiv C'\theta[v\theta]_p \equiv C[l\sigma]_p$. It follows from $v\theta \equiv l\sigma$ that (θ, σ) is a unifier of v and l. Let (δ, σ') be the most general unifier such that $x\sigma' \in \mathcal{X} \setminus (\mathcal{V}ar(s) \cup \mathcal{VR}an(\delta))$ for every $x \notin \mathcal{V}ar(l)$. Then, we have $s \equiv C'[v]_p \underset{\delta}{\rightsquigarrow}_R C'\delta[r\sigma']_p$. On the other hand, it follows from $\mathcal{V}ar(C'[\,]) \subseteq \mathcal{V}ar(s) = \mathcal{D}om(\theta)$ and Lemma 1 that $r\sigma'\theta' \equiv r\sigma$ and $C'\delta\theta'[\,]_p \equiv C'\theta[\,]_p \equiv C[\,]_p$. Hence, $t \equiv C[r\sigma]_p \equiv C'\delta\theta'[r\sigma'\theta']_p \equiv (C'\delta[r\sigma']_p)\theta'$, which conclude the first part of the proof by taking $t' \equiv C'\delta[r\sigma]_p$.

Now, we show that $P' \leq \mathcal{O}_{\mathcal{X}}(t')$. Let $q \in \mathcal{O}_{\mathcal{X}}(t')$. Consider the case that $p \not\leq q$. Since $q \not\leq p$ from $t' \equiv C'\delta[r\sigma']_p$, we have $q \in \mathcal{O}_{\mathcal{X}}(C'\delta[\,]_p)$. There exists $q' \leq q$ such that $q' \in \mathcal{O}_{\mathcal{X}}(C'[\,]_p) \subseteq \mathcal{O}_{\mathcal{X}}(s)$. It follows from $p \in \mathcal{O}_{\mathcal{X}}(s)$ that

$p' \leq q'$ for some $p' \in P$. Thus, $p' \in P'$ follows from $P \Rightarrow^{[p,\rho]} P'$. We have shown $p' \leq q$ and $p' \in P'$. Consider the case that $p \leq q$. If q was introduced by an extra variable, that is $q = pq'$ and $r|_{q'} \equiv x \in \mathcal{EV}ar(\rho)$ for some q', we have $pq' \in P'$. Otherwise, q was moved via the reduction, that is $q = pq'w$ and $r|_{q'} \equiv y$ for some $y \in \mathcal{V}ar(l)$ and $w \in \mathcal{O}_{\mathcal{X}}(y\sigma')$. Then, we can show $p' \leq pq'w$ for some $p' \in P'$, from the fact that there exists $p'' \in P$ satisfying $p'' \leq pq'w$ for all q' such that $l|_{q'} \equiv y$. $\square$

Proof of Theorem 2 From Lemma 2, we can easily prove the following claim by induction on n; if $P : s'\theta \xrightarrow{n}_R P' : t$ and $P \leq \mathcal{O}_{\mathcal{X}}(s)$, then $s' \overset{*}{\rightsquigarrow}_R t'$ and $t'\theta' \equiv t$ for some t' and θ'. $\square$

Proof of Theorem 3 We prove by induction on n that $s \xrightarrow{n}_R t$ implies a linear term t' and a substitution θ such that $s \underset{\delta}{\overset{*}{\rightsquigarrow}}_R t'$ and $t \equiv t'\theta$. The case of $n = 0$ is trivial. Suppose $s \xrightarrow{n-1}_R u \rightarrow_R t$. By induction hypothesis, there exist a linear term u' and a substitution θ' such that $s \overset{*}{\rightsquigarrow}_R u'$ and $u'\theta' \equiv u$. Suppose $u \equiv C[l\sigma]_p \rightarrow_R^{[p,\rho]} C[r\sigma] \equiv t$ where $\rho : l \rightarrow r \in R$. Consider the case $p \in \mathcal{O}(u') \backslash \mathcal{O}_{\mathcal{X}}(u')$. Then, we have $u' \equiv C'[v]_p$ for some $C'[\,]$ and v, and also have $u'\theta' \equiv C'\theta'[v\theta']_p \equiv C[l\sigma]_p$. In similar to the proof of Lemma 2, we can show that $u' \equiv C'[v]_p \underset{\delta}{\rightsquigarrow}_R C'\delta[r\sigma']_p$ and $t \equiv (C'\delta[r\sigma']_p)\theta'$, where $(\delta, \sigma') = \mathtt{mgu}(v, l)$. Here, we can show the linearity of $t' \equiv C'\delta[r\sigma']_p$ from the linearity of u' and r. Consider the case $p \notin \mathcal{O}(u') \setminus \mathcal{O}_{\mathcal{X}}(u')$. Then, we have $u'|_q \equiv y$ and $p = qq'$ for some $y \in \mathcal{V}ar(u')$, q and q'. From the linearity of u', we have $u' \equiv C'[y]_q$ and $y\theta' \equiv C''[l\sigma]_{q'}$ for some $C'[\,]_q$ and $C''[\,]_{q'}$. Let $\theta = \theta'|_{\mathcal{D}om(\theta')\setminus\{y\}} \cup \{y \mapsto C''[r\sigma]_{q'}\}$. Then, θ is a substitution, and we have $s \overset{*}{\rightsquigarrow}_R u'$ and $u'\theta \equiv C'\theta[y\theta]_q \equiv C[r\sigma]_p \equiv t$. $\square$

Let R be an EV-TRS and t be a term. We say that t is *almost terminating* with respect to $\rightsquigarrow_R$ if there exists an infinite narrowing sequence starting from t and every proper subterm of t is N-SN with respect to R. It is clear that t has an almost-terminating subterm with respect to $\rightsquigarrow_R$ if there is an infinite narrowing sequence starting from t. Let $s \rightsquigarrow_R^q t$. Then, we write $s \rightsquigarrow_R^{p<} t$ if $p < q$, and write $s \rightsquigarrow_R^{p\leq} t$ if $p \leq q$. We abbreviate the sequence $a_{i,1}, ..., a_{i,n_i}$ as $\boldsymbol{a}_i$.

Proof of Theorem 6 We prove here only the claim (b) since the proof of (a) is similar to (b).

We first show the *only if*-part by constructing an infinite ground R-narrowing-chain from an infinite ground sequence. We assume that R is not N-GSN. Then, there exists an infinite ground narrowing-sequence. Let s_0 be an almost-terminating ground-term with respect to $\rightsquigarrow_R$, and $s_0 \equiv f_1(\boldsymbol{u}_0)$. Then, we have $f_1(\boldsymbol{u}_0) \overset{*}{\rightsquigarrow}_R^{\varepsilon<} f_1(\boldsymbol{v}_1) \equiv s_1' \underset{\delta_1}{\rightsquigarrow}_R^{[\varepsilon,\rho_1]} r_1\sigma_1 \rightsquigarrow_R \cdots$ where $\rho_1 : f_1(\boldsymbol{w}_1)(\equiv l_1) \rightarrow r_1 \in R$ and $(\delta_1, \sigma_1) = \mathtt{mgu}(s_1', l_1)$. Since $\boldsymbol{v}_1$ are N-SN, $x\sigma_1$ is N-SN for any $x \in \mathcal{D}om(\sigma_1)$. Hence, there is a subterm $t_1 \equiv f_2(\boldsymbol{u}_1)$ of r_1 such that $t_1\sigma_1$ is almost terminating with respect to $\rightsquigarrow_R$. Since $t_1\sigma_1$ is almost terminating with respect to $\rightsquigarrow_R$, as similar as the case of s_0, we have $t_1\sigma_1 \equiv f_2(\boldsymbol{u}_1\sigma_1) \overset{*}{\rightsquigarrow}_R^{\varepsilon<} f_2(\boldsymbol{v}_2) \equiv s_2' \underset{\delta_2}{\rightsquigarrow}_R^{[\varepsilon,\rho_2]} r_2\sigma_2 \rightsquigarrow_R \cdots$ where $\rho_2 : f_2(\boldsymbol{w}_2)(\equiv l_2) \rightarrow r_2 \in R$ and $(\delta_2, \sigma_2) = \mathtt{mgu}(s_2', l_2)$. Since $\boldsymbol{v}_2$ are N-SN, $x\sigma_2$ is also N-SN for any $x \in \mathcal{D}om(\sigma_2)$. Hence, there is a subterm $t_2 \equiv f_3(\boldsymbol{u}_2)$ of r_2 such that $t_2\sigma_2$ is almost terminating with respect to $\rightsquigarrow_R$. Here, $\langle F_1(\boldsymbol{w}_1), F_2(\boldsymbol{u}_2)\rangle, \langle F_2(\boldsymbol{w}_2), F_3(\boldsymbol{u}_3)\rangle \in \mathcal{DP}_R$ follow from ρ_1 and ρ_2. Since $u_{0,i} \overset{*}{\rightsquigarrow}_R v_{1,i}$, $(\delta_1, \sigma_1) = \mathtt{mgu}(s'_1, l_1)$, $u_{1,i} \overset{*}{\rightsquigarrow}_R v_{2,i}$ and $(\delta_2, \sigma_2) = \mathtt{mgu}(s'_2, l_2)$, we have a ground chain $F_1(\boldsymbol{u}_0)\langle F_1(\boldsymbol{w}_1), F_2(\boldsymbol{u}_2)\rangle\langle F_2(\boldsymbol{w}_2), F_3(\boldsymbol{u}_3)\rangle$.

By repeating similarly to the above, we obtain an infinite ground chain $F_1(\boldsymbol{u}_0)\langle F_1(\boldsymbol{w}_1), F_2(\boldsymbol{u}_2)\rangle\langle F_2(\boldsymbol{w}_2), F_3(\boldsymbol{u}_3)\rangle \cdots$.

We prove *if*-part by constructing an infinite ground narrowing-sequence from an infinite ground R-narrowing-chain $F_1(\boldsymbol{u}_0)\langle F_1(\boldsymbol{w}_1), F_2(\boldsymbol{u}_2)\rangle\langle F_2(\boldsymbol{w}_2), F_3(\boldsymbol{u}_3)\rangle \cdots$. From the definition of R-narrowing-chain, there are a term $F_i(\boldsymbol{v}_i)$ and the most general unifier $(\delta_i, \sigma_i) = \mathtt{mgu}(F_i(\boldsymbol{v}_i), F_i(\boldsymbol{w}_i))$ such that $F_i(\boldsymbol{u}_i)\sigma_{i-1} \overset{*}{\rightsquigarrow}_R F_i(\boldsymbol{v}_i)$, where $F_1(\boldsymbol{u}_0)\sigma_0 \equiv F_1(\boldsymbol{u}_0)$. From the construction of dependency pairs, we have $\rho_i : f_i(\boldsymbol{w}_i) \to C_i[f_{i+1}(\boldsymbol{u}_{i+1})] \in R$. Hence, we can easily construct an infinite ground narrowing-sequence $f_1(\boldsymbol{u}_0) \overset{*}{\rightsquigarrow}_R f_1(\boldsymbol{v}_1) \underset{\delta_1}{\rightsquigarrow}{}^{[\varepsilon,\rho_1]}_R C_1\delta_1[f_2(\boldsymbol{u}_1)]_{p_1}$ $\overset{*}{\rightsquigarrow}{}^{p_1<}_R C_1\delta_1[f_2(\boldsymbol{v}_2)] \underset{\delta_2}{\rightsquigarrow}{}^{[p_1,\rho_2]}_R C_1\delta_1[C_2\delta_2[f_3(\boldsymbol{u}_2)\sigma_2]_{p_2}]_{p_1} \rightsquigarrow_R \cdots$. □

Let π be a simple AF function and θ be a substitution. We define the substitution θ_π as $\theta_\pi = \{\, x \mapsto \pi(t) \mid x \in \mathcal{D}om(\theta), x\theta \equiv t \,\}$. Let t be a term. It is clear that $\pi(t\theta) \equiv \pi(t)\theta_\pi$ holds.

Lemma 3. *Let R be an EV-TRS, π be a simple AF function that eliminates all extra variables of R and $\mathcal{DP}_R$. For any $s, t \in T(\mathcal{F}, \mathcal{X})$, if $s \underset{\delta}{\overset{*}{\rightsquigarrow}}_R t$ and $\pi(s)$ is ground then $\pi(s) \overset{*}{\rightsquigarrow}_{\pi(R)} \pi(t)$ and $\pi(s) \xrightarrow{*}_{\pi(R)} \pi(t)$.*

Proof. We first prove by induction on n that $s \overset{n}{\rightsquigarrow}_R t$ and $\pi(s) \in T(\mathcal{F})$ imply $\pi(s) \overset{*}{\rightsquigarrow}_{\pi(R)} \pi(t)$. In case of $n = 0$, it is trivial. We assume that $s \underset{\delta}{\rightsquigarrow}{}^{[p,\rho]}_R u \underset{\delta'}{\overset{n-1}{\rightsquigarrow}}_R t$ and $\pi(s)$ is ground, where $\rho : l \to r \in R$. Then, there are $C[\;]_p$ and s' such that $s \equiv C[s']_p$, $(\delta, \sigma) = \mathtt{mgu}(s', l)$ and $u \equiv C\delta[r\sigma]_p$. Consider the case that $\Box$ in $C[\;]$ is eliminated by π. Let $\pi(C[\;]_p) \equiv u'$. Then, we have $\pi(s) \equiv \pi(C[s']_p) \equiv u'$ and $\pi(u) \equiv \pi(C\delta[r\sigma]_p) \equiv \pi(C\delta[\;]_p) \equiv \pi(C[\;]_p)\delta_\pi \equiv u'\delta_\pi$. Since $\pi(s)$ is ground, u' is also ground. Then, we have $\pi(u) \equiv u'\delta_\pi \equiv u'$. By induction hypothesis, we have $\pi(u) \overset{*}{\rightsquigarrow}_{\pi(R)} \pi(t)$. Therefore, $\pi(s) \equiv \pi(u) \overset{*}{\rightsquigarrow}_{\pi(R)} \pi(t)$. Consider the otherwise. Let $\pi(C[\;]_p) \equiv C'[\;]_q$. Then, we have $\pi(s) \equiv \pi(C[s']_p) \equiv (\pi(C[\;]_p)[\pi(s')]_q \equiv C'[\pi(s')]_q$ and $\pi(u) \equiv \pi(C\delta[r\sigma]_p) \equiv (\pi(C\delta[\;]_p))[\pi(r\sigma)]_q \equiv C'\delta_\pi[\pi(r\sigma)]_q$. Since $\pi(s)$ is ground, $C'[\;]_q$ is also ground. Then, it follows from $C'\delta_\pi \equiv C'$ that $\pi(s) \equiv C'[\pi(r\sigma)]_q$. We also have $\pi(l) \to \pi(r) \in \pi(R)$. It follows from the assumption that $\mathcal{V}ar(\pi(l)) \supseteq \mathcal{V}ar(\pi(r))$. Then, $\pi(r\sigma)$ is ground. Since $\pi(u) \equiv C'[\pi(r\sigma)]$ is also ground, we have $\pi(u) \overset{*}{\rightsquigarrow}_{\pi(R)} \pi(t)$ by induction hypothesis. On the other hand, $\pi(s')\delta_\pi \equiv \pi(l)\sigma_\pi$ follows from $s'\delta \equiv l\sigma$, $\pi(s'\delta) \equiv \pi(s')\delta_\pi$ and $\pi(l\sigma) \equiv \pi(l)\sigma_\pi$. Since $\pi(s')$ is ground, we have $\pi(s')\delta_\pi \equiv \pi(s') \equiv \pi(l)\sigma_\pi$. Therefore, we have the sequence $\pi(s) \equiv C'[\pi(s')]_q \equiv C'[\pi(l)\sigma_\pi]_q \rightsquigarrow_{\pi(R)} C'[\pi(r)\sigma_\pi]_q \equiv C'[\pi(r\sigma)]_q \equiv \pi(u) \overset{*}{\rightsquigarrow}_{\pi(R)} \pi(t)$. Since $\pi(R)$ is a TRS and $\pi(s)$ is ground, it is clear that $\pi(s) \overset{*}{\rightsquigarrow}_{\pi(R)} \pi(t)$ if and only if $\pi(s) \xrightarrow{*}_{\pi(R)} \pi(t)$. □

Proof of Theorem 7 By Theorem 6, there exists an infinite R-narrowing-chain when R is not N-GSN. The first part can be easily proved by constructing an infinite ground $\pi(R\&\mathcal{DP}_R)$-narrowing-chain from an infinite ground R-narrowing-chain, using Lemma 3. The second part can be proved similarly. □

A Safe Relational Calculus for Functional Logic Deductive Databases [⋆]

Jesús M. Almendros-Jiménez and Antonio Becerra-Terón

Dpto. de Lenguajes y Computación. Universidad de Almería.
email: {jalmen,abecerra}@ual.es

Abstract. In this paper, we present an extended relational calculus for expressing queries in functional-logic deductive databases. This calculus is based on first-order logic and handles relation predicates, equalities and inequalities over partially defined terms, and approximation equations. For the calculus formulas, we have studied syntactic conditions in order to ensure the domain independence property. Finally, we have studied its equivalence w.r.t. the original query language which is based on equality and inequality constraints.

1 Introduction

Database technology is involved in most software applications. For this reason *functional logic languages* [7] should include database features in order to increase its application field and cover with 'real world' applications. In order to integrate functional logic programming and databases, we propose: (1) to adapt functional logic programs to databases, by considering a suitable *data model* and a *data definition language*; (2) to *consider an extended relational calculus* as query language, which handles the proposed data model; and finally, (3) to provide *semantic foundations* to the new query language.

With respect to (1), the underlying data model of functional logic programming is *complex* from a database point of view [1]. Firstly, types can be defined by using *recursively defined datatypes*, as *lists* and *trees*. Therefore, the attribute values can be *multi-valued*; that is, more than one value (for instance, a set of values enclosed in a list) for a given attribute corresponds to each set of key attributes. In addition, we have adopted *non-deterministic semantics* from functional-logic programming, investigated in the framework *CRWL* [6]. Under non-deterministic semantics, values can be *grouped into sets*, representing the set of values of the output of a non-deterministic function. Therefore, the data model is complex in a double sense, allowing the handling of complex values built from recursively defined datatypes, and complex values grouped into sets.

Moreover, functional logic programming is able to handle *partial and possibly infinite data*. Therefore, in our setting, an attribute can be partially defined or, even, include possibly infinite information. The first case can be interpreted

[⋆] This work has been partially supported by the Spanish project of the Ministry of Science and Technology "INDALOG" TIC2002-03968.

as follows: the database includes *unknown information* o *partially defined information*; and the second case indicates that the database can store *infinite information.* In addition, database instances can be infinite (*infinite attribute values* or an *infinite set of tuples*). The infinite information can be handled by means of *partial approximations.* Moreover, we have adopted the handling of *negation* from functional logic programming, studied in the framework *CRWLF* [9]. As a consequence, the data model proposed here also handles *non-existent information*, and *partially non-existent information.*

Finally, we propose a *data definition language* which, basically, consists on *database schema definitions*, *database instance definitions* and *(lazy) function definitions.* A database schema definition includes *relation names*, and a set of *attributes* for each relation. For a given database schema, the *database instances* define *key values* and *non-key attribute values*, by means of *(constructor-based) conditional rewriting rules*, where conditions handle equality and inequality constraints. In addition, we can define a set of functions. These functions will be used by queries in order to handle recursively defined datatypes, also named *interpreted functions* in a database setting. As a consequence, "pure" functional-logic programs can be considered as a particular case of our programs.

With respect to (2), typically the query language of functional logic languages is based on the solving of conjunctions of (in)equality constraints, which are defined w.r.t. some (in)equality relations over terms [6, 9]. Our relational calculus will handle *conjunctions* of *atomic formulas*, which are *relation predicates*, *(in)equality relations* over terms, and *approximation equations* in order to handle interpreted functions. Logic formulas are either existentially or universally quantified, depending on whether they include negation or not.

However, it is known in database theory that a suitable query language must ensure the property of *domain independence* [2]. A query is domain independent, whenever the query satisfies, properly, two conditions: *(a) the query output over a finite relation is also a finite relation; and (b) the output relation only depends on the input relations.* In general, it is undecidable, and therefore syntactic conditions have to be developed in such a way that, only the so-called *safe queries* (satisfying these conditions) ensure the property of domain independence. For instance, [1] and [10] propose syntactic conditions, which allow the building of safe formulas in a relational calculus with complex values and linear constraints, respectively. In this line, we have developed syntactic conditions over our query language, which allow the building of the so-called *safe formulas.*

Extended relational calculi have been studied as alternative query languages for *deductive databases* [1], and *constraint databases* [8, 10]. Our extended relational calculus is in the line of [1], in which deductive databases handle complex values in the form of *set* and *tuple* constructors. In our case, we generalize the mentioned calculus for handling *complex values built from (arbitrary) recursively defined datatypes.* In addition, our calculus is similar to the calculi for constraint databases in the sense of allowing the handling of *infinite databases.* However, in the framework of constraint databases, infinite databases model *infinite objects* by means of *(linear) equations* and *inequations*, and *intervals*, which are handled in a symbolic way. Here, infinite databases are handled by means of *lazyness* and partial approximations. In addition, we handle constraints which consist on equality and inequality relations over complex values.

Finally, and w.r.t. (3), we will show that our relational calculus is *equivalent* to a query language based on *(in)equality constraints*, similar to existent functional logic languages. In addition, we have developed theoretical foundations for the database instances, by defining a *partial order* representing the *approximation ordering on database instances*, and a suitable *fixed point operator* which computes the *least database instance* (w.r.t. the approximation order) induced from a set of conditional rewriting rules.

Finally, remark that this work goes towards the design of a functional logic deductive language for which an operational semantics [3, 5], and a relational algebra [4] have been studied.

The organization of this paper is as follows. Section 2 describes the data model; section 3 presents the relational calculus; section 4 states the equivalence result between the relational calculus and the original query language; and section 5 defines the least database induced from a set or conditional rules.

2 The Data Model

In our framework, we consider two main kinds of partial information: undefined information (ni), represented by $\perp$, which means *information unknown, although it may exist*, and nonexistent information (ne), represented by F, which means *the information does not exist.*

Now, let's suppose a complex value, storing information about job salary and salary bonus, by means of a data constructor (like a *record*) `s&b(Salary, Bonus)`. Then, we can additionally consider the following kinds of partial information:

s&b(3000, 100)	totally defined information, *expressing that a person's salary is* 3000 euros, *and his(her) salary bonus is* 100 euros
s&b($\perp$, 100)	partially undefined information (pni), *expressing that a person's salary bonus is known, that is* 100 euros, *but not his(her) salary*
s&b(3000, F)	partially nonexistent information (pne), *expressing that a person's salary is* 3000 euros, *but (s)he has no salary bonus*

Over these kinds of information, the (in)equality relations can be defined as follows:

- (1) = (*syntactic equality*), expressing that *two values are syntactically equal*; for instance, the relation s&b(3000, $\perp$) = s&b(3000, $\perp$) is satisfied.
- (2) $\downarrow$ (*strong equality*), expressing that *two values are equal and totally defined*; for instance, the relation s&b(3000, 25) $\downarrow$ s&b(3000, 25) holds, and the relations s&b(3000, $\perp$) $\downarrow$ s&b(3000, 25) and s&b(3000, F) $\downarrow$ s&b(3000, 25) do not hold.
- (3) $\uparrow$ (*strong inequality*), where *two values are (strongly) different, if they are different in their defined information*; for instance, the relation s&b(3000, $\perp$) $\uparrow$ s&b(2000, 25) is satisfied, whereas the relation s&b(3000, F) $\uparrow$ s&b(3000, 25) does not hold.

In addition, we will consider their logical negations, that is, $\neq$, $\not\downarrow$ and $\not\uparrow$, which represent a *syntactic inequality*, *(weak) inequality* and *(weak) equality* relation, respectively. Next, we will formally define the above equality and inequality relations.

Assuming *constructor symbols* $c, d, \ldots$ $DC = \cup_n DC^n$ each one with an associated arity, and the symbols $\bot$, F as special cases with arity 0 (not included in DC), and a set $\mathcal{V}$ of variables $X, Y, \ldots$, we can build the set of *c-terms with* $\bot$ *and* F, denoted by $CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V})$. C-terms are complex values including variables which implicitly are universally quantified. We can use *substitutions* $Subst_{DC,\bot,\mathsf{F}} = \{\theta \mid \theta : \mathcal{V} \rightarrow CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V})\}$, in the usual way. The above (in)equality relations can be formally defined as follows.

Definition 1 (Relations over Complex Values [9]). *Given c-terms t, t': (1) $t = t' \Leftrightarrow_{def}$ t and t' are syntactically equal; (2) $t \downarrow t' \Leftrightarrow_{def}$ $t = t'$ and $t \in CTerm_{DC}(\mathcal{V})$; (3) $t \uparrow t' \Leftrightarrow_{def}$ they have a DC-clash, where t and t' have a DC-clash whether they have different constructor symbols of DC at the same position.*
In addition, their logical negations can be defined as follows: (1') $t \neq t' \Leftrightarrow_{def}$ t and t' have a $DC \cup \{\mathsf{F}\}$-clash; (2') $t \not\downarrow t' \Leftrightarrow_{def}$ t or t' contains F as subterm, or they have a DC-clash; (3') $\not\uparrow$ is defined as the least symmetric relation over $CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V})$ satisfying: $X \not\uparrow X$ for all $X \in \mathcal{V}$, $\mathsf{F} \not\uparrow t$ for all t, and if $t_1 \not\uparrow t'_1, \ldots, t_n \not\uparrow t'_n$, then $c(t_1, \ldots, t_n) \not\uparrow c(t'_1, \ldots, t'_n)$ for $c \in DC^n$.

Given that complex values can be partially defined, a *partial ordering* $\leq$ can be considered. This ordering is defined as the least one satisfying: $\bot \leq t$, $X \leq X$, and $c(t_1, \ldots, t_n) \leq c(t'_1, \ldots, t'_n)$ if $t_i \leq t'_i$ for all $i \in \{1, \ldots, n\}$ and $c \in DC^n$. The intended meaning of $t \leq t'$ is that t is *less defined or has less information* than t'. In particular, $\bot$ is the *bottom element*, given that $\bot$ represents undefined information (ni), that is, information more refinable can exist. In addition, F is *maximal* under $\leq$ (F satisfies the relations $\bot \leq \mathsf{F}$ and $\mathsf{F} \leq \mathsf{F}$), representing nonexistent information (ne), that is, no further refinable information can be obtained, given that it does not exist. Now, we can build the set of (possibly infinite) cones of c-terms $\mathcal{C}(CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V}))$, and the set of (possibly infinite) ideals of c-terms $\mathcal{I}(CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V}))$. Cones and ideals can also be partially ordered under the *set-inclusion* ordering (i.e. $\subseteq$) in such a way that, the set of ideals is a *complete partial order* (*cpo*). Over cones and ideals, we can define the following *equality* and *inequality* relations.

Definition 2 (Relations over Sets of Complex Values). *Given $\mathcal{C}$ and $\mathcal{C}' \in \mathcal{C}(CTerm_{DC,\bot,\mathsf{F}}(\mathcal{V}))$: (1) $\mathcal{C} \bowtie \mathcal{C}'$ holds, whenever at least one value in $\mathcal{C}$ and $\mathcal{C}'$ is* strongly equal *and (2) $\mathcal{C} <> \mathcal{C}'$ holds, whenever at least one value in $\mathcal{C}$ and $\mathcal{C}'$ is* strongly different*; and their logical negations (1') $\mathcal{C} \not\bowtie \mathcal{C}'$ holds, whenever all values in $\mathcal{C}$ and $\mathcal{C}'$ are* weakly different *and (2') $\mathcal{C} \not<> \mathcal{C}'$ holds, whenever all values in $\mathcal{C}$ and $\mathcal{C}'$ are* weakly equal.

Definition 3 (Database Schemas). *Assuming a Milner's style polymorphic type system, a database schema S is a finite set of relation schemas $R_1, \ldots, R_p$*

in the form: $R(\underline{A_1} : T_1, \ldots, \underline{A_k} : T_k, A_{k+1} : T_{k+1}, \ldots, A_n : T_n)$*, wherein the relation names are a pairwise disjoint set, and the relation schemas* $R_1, \ldots, R_p$ *include a pairwise disjoint set of typed attributes*[1] $(A_1 : T_1, \ldots, A_n : T_n)$.

In the relation schema R, $A_1, \ldots, A_k$ are *key attributes* and $A_{k+1}, \ldots, A_n$ are *non-key attributes*, denoted by the sets $Key(R)$ and $NonKey(R)$, respectively. Key values are supposed to identify each tuple of the relation. Finally, we denote by $nAtt(R) = n$ and $nKey(R) = k$.

Definition 4 (Databases). *A database* D *is a triple* (S, DC, IF)*, where* S *is a database schema,* DC *is a set of constructor symbols, and* IF *represents a set of interpreted function symbols, each one with an associated arity.*

We denote the set of *defined schema symbols* (i.e. relation and non-key attribute symbols) by $DSS(D)$, and the set of *defined symbols* by $DS(D)$ (i.e. $DSS(D)$ together with IF). As an example of database, we can consider the following one:

$$\begin{array}{ll}
S & \left\{\begin{array}{l}
\mathtt{person\_job}(\underline{\mathtt{name}} : \mathtt{people}, \mathtt{age} : \mathtt{nat}, \mathtt{address} : \mathtt{dir}, \mathtt{job\_id} : \mathtt{job}, \mathtt{boss} : \mathtt{people}) \\
\mathtt{job\_information}(\underline{\mathtt{job\_name}} : \mathtt{job}, \mathtt{salary} : \mathtt{nat}, \mathtt{bonus} : \mathtt{nat}) \\
\mathtt{person\_boss\_job}(\underline{\mathtt{name}} : \mathtt{people}, \mathtt{boss\_age} : \mathtt{cbossage}, \mathtt{job\_bonus} : \mathtt{cjobbonus}) \\
\mathtt{peter\_workers}(\underline{\mathtt{name}} : \mathtt{people}, \mathtt{work} : \mathtt{job})
\end{array}\right. \\
DC & \left\{\begin{array}{l}
\mathtt{john} : \mathtt{people},\ \mathtt{mary} : \mathtt{people},\ \mathtt{peter} : \mathtt{people} \\
\mathtt{lecturer} : \mathtt{job},\ \mathtt{associate} : \mathtt{job},\ \mathtt{professor} : \mathtt{job} \\
\mathtt{add} : \mathtt{string} \times \mathtt{nat} \rightarrow \mathtt{dir} \\
\mathtt{b\&a} : \mathtt{people} \times \mathtt{nat} \rightarrow \mathtt{cbossage} \\
\mathtt{j\&b} : \mathtt{job} \times \mathtt{nat} \rightarrow \mathtt{cjobbonus}
\end{array}\right. \\
IF & \left\{\ \mathtt{retention\_for\_tax} : \mathtt{nat} \rightarrow \mathtt{nat}\right.
\end{array}$$

where S includes the schemas `person_job` (storing information about people and their jobs) and `job_information` (storing generic information about jobs), and the *"views"* `person_boss_job`, and `peter_workers`, which will take key values from the set of key values defined for `person_job`. The first view includes, for each person, the pairs in the form of records constituted by: (a) his/her boss and boss' age, by using the complex c-term `b&a(people,nat)`; and (b) his/her job and job salary bonus, by using the complex c-term `j&b(job,nat)`. The second view includes `peter`'s workers. The set DC includes constructor symbols for the types `people`, `job`, `dir`, `cbossage` and `cjobbonus`, and IF defines the interpreted function symbol `retention_for_tax`, which computes the salary free of taxes. In addition, we can consider database schemas involving *(possibly) infinite databases* such as shown in the following:

$$\begin{array}{ll}
S & \left\{\begin{array}{l}
\mathtt{2Dpoint}(\underline{\mathtt{coord}} : \mathtt{cpoint},\ \mathtt{color} : \mathtt{nat}) \\
\mathtt{2Dline}(\underline{\mathtt{origin}} : \mathtt{cpoint},\ \underline{\mathtt{dir}} : \mathtt{orientation}, \mathtt{next} : \mathtt{cpoint}, \mathtt{points} : \mathtt{cpoint}, \\
\mathtt{list\_of\_points} : \mathtt{list(cpoint)})
\end{array}\right. \\
DC & \left\{\begin{array}{l}
\mathtt{north} : \mathtt{orientation}, \mathtt{south} : \mathtt{orientation}, \mathtt{east} : \mathtt{orientation}, \mathtt{west} : \mathtt{orientation}, \ldots \\
[\ ] : \mathtt{list\ A},\quad [\ |\ ] : \mathtt{A} \times \mathtt{list\ A} \rightarrow \mathtt{list\ A} \\
\mathtt{p} : \mathtt{nat} \times \mathtt{nat} \rightarrow \mathtt{cpoint}
\end{array}\right. \\
IF & \left\{\ \mathtt{select} : (\mathtt{list\ A}) \rightarrow \mathtt{A}\right.
\end{array}$$

[1] We can suppose attributes qualified with the relation name when the names coincide.

wherein the schemas `2Dpoint` and `2Dline` are defined for representing bidimensional points and lines, respectively. `2Dpoint` includes the point coordinates (`coord`) and `color`. Lines represented by `2Dline` are defined by using a starting point (`origin`) and direction (`dir`). Furthermore, `next` indicates the next point to be drawn in the line, `points` stores the *(infinite) set* of points of this line, and `list_of_points` the *(infinite) list* of points of the line. Here, we can see the double use of complex values: (1) a set (which can be implicitly assumed), and (2) a list.

Definition 5 (Schema Instances). *A schema instance $\mathcal{S}$ of a database schema S is a set of relation instances $\mathcal{R}_1, \ldots \mathcal{R}_p$, where each relation instance $\mathcal{R}_j$, $1 \le j \le p$, is a (possibly infinite) set of tuples of the form $(V_1, \ldots, V_n)$ for the relation $R_j \in S$, with $n = nAtt(R)$ and $V_i \in \mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V}))$. In particular, each V_j $(j \le nKey(R))$ satisfies $V_j \in CTerm_{DC,\mathsf{F}}(\mathcal{V})$.*

The last condition forces the key values to be one-valued and without $\perp$. Attribute values can be non-ground, wherein the variables are implicitly universally quantified.

Definition 6 (Database Instances). *A database instance $\mathcal{D}$ of a database $D = (S, DC, IF)$ is a triple $(\mathcal{S}, \mathcal{DC}, \mathcal{IF})$, where $\mathcal{S}$ is a schema instance, $\mathcal{DC} = CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, and $\mathcal{IF}$ is a set of function interpretations $f^{\mathcal{D}}, g^{\mathcal{D}}, \ldots$ satisfying $f^{\mathcal{D}} : CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})^n \to \mathcal{C}(CTerm_{DC,\perp,\mathsf{F}}\ (\mathcal{V}))$ is monotone, that is, $f^{D}(t_1, \ldots, t_n) \subseteq f^{D}(t'_1, \ldots, t'_n)$ if $t_i \le t'_i$, $1 \le i \le n$, for each $f \in IF^n$.*

Databases can be infinite, although, as a particular case, we can consider finite databases. A schema instance $\mathcal{S}$ is *ground* if tuples only contain ground c-terms. A schema instance $\mathcal{S}$ is *finite* if it contains a finite set of tuples and finite attributes. A database instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ is *finite*, whenever $\mathcal{S}$ is ground and finite, and $\mathcal{IF}$ is finite w.r.t. $\mathcal{S}$.

Next, we will show an example of schema instance for the schemas `person_job`, `job_information`, and the views `person_boss_job` and `peter_workers`:

$$
\begin{array}{ll}
\texttt{person\_job} & \left\{ \begin{array}{l}
(\texttt{john}, \{\perp\}, \{\texttt{add}('\texttt{6th Avenue}', 5)\}, \{\texttt{lecturer}\}, \{\texttt{mary}, \texttt{peter}\}) \\
(\texttt{mary}, \{\perp\}, \{\texttt{add}('\texttt{7th Avenue}', 2)\}, \{\texttt{associate}\}, \{\texttt{peter}\}) \\
(\texttt{peter}, \{\perp\}, \{\texttt{add}('\texttt{5th Avenue}', 5)\}, \{\texttt{professor}\}, \{\mathsf{F}\})
\end{array} \right. \\
\texttt{job\_information} & \left\{ \begin{array}{l}
(\texttt{lecturer}, \{1200\}, \{\mathsf{F}\}) \\
(\texttt{associate}, \{2000\}, \{\mathsf{F}\}) \\
(\texttt{professor}, \{3200\}, \{1500\})
\end{array} \right. \\
\texttt{person\_boss\_job} & \left\{ \begin{array}{l}
(\texttt{john}, \{\texttt{b\&a}(\texttt{mary}, \perp), \texttt{b\&a}(\texttt{peter}, \perp)\}, \{\texttt{j\&b}(\texttt{lecturer}, \mathsf{F})\}) \\
(\texttt{mary}, \{\texttt{b\&a}(\texttt{peter}, \perp)\}, \{\texttt{j\&b}(\texttt{associate}, \mathsf{F})\}) \\
(\texttt{peter}, \{\texttt{b\&a}(\mathsf{F}, \perp)\}, \{\texttt{j\&b}(\texttt{professor}, 1500)\})
\end{array} \right. \\
\texttt{peter\_workers} & \left\{ \begin{array}{l}
(\texttt{john}, \{\texttt{lecturer}\}) \\
(\texttt{mary}, \{\texttt{associate}\})
\end{array} \right.
\end{array}
$$

With respect to the modeling of (possibly) infinite databases, we can consider the following approximation to the instance of the relation schema `2Dline` including (*possibly infinite*) values in the defined attributes:

$$
\begin{array}{ll}
\texttt{2Dpoint} & \left\{ (\texttt{p}(0,0), \{1\}), (\texttt{p}(0,1), \{2\}), (\texttt{p}(1,0), \{\mathsf{F}\}), \ \ldots \right. \\
\texttt{2Dline} & \left\{ \begin{array}{l}
(\texttt{p}(0,0), \texttt{north}, \{\texttt{p}(0,1)\}, \{\texttt{p}(0,1), \texttt{p}(0,2), \perp\}, \{[\texttt{p}(0,0), \texttt{p}(0,1), \texttt{p}(0,2) | \perp]\}), \ \ldots \\
(\texttt{p}(1,1), \texttt{east}, \{\texttt{p}(2,1)\}, \{\texttt{p}(2,1), \texttt{p}(3,1), \perp\}, \{[\texttt{p}(1,1), \texttt{p}(2,1), \texttt{p}(3,1) | \perp]\}), \ \ldots
\end{array} \right.
\end{array}
$$

Instances can also be *partially ordered* as follows.

Definition 7 (Approximation Ordering on Databases). *Given a database* $D = (S, DC, IF)$ *and two instances* $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ *and* $\mathcal{D}' = (\mathcal{S}', \mathcal{DC}, \mathcal{IF}')$*, then* $\mathcal{D} \sqsubseteq \mathcal{D}'$*, if (1)* $V_i \subseteq V_i'$ *for each* $k+1 \leq i \leq n$*,* $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}$ *and* $(V_1, \ldots, V_k, V_{k+1}', \ldots, V_n') \in \mathcal{R}'$*, where* $\mathcal{R} \in \mathcal{S}$ *and* $\mathcal{R}' \in \mathcal{S}'$*, are relation instances of* $R \in S$ *and* $k = nKey(R)$*; and (2)* $f^{\mathcal{D}}(t_1, \ldots, t_n) \subseteq f^{\mathcal{D}'}(t_1, \ldots, t_n)$ *for each* $t_1, \ldots, t_n \in \mathcal{DC}$*,* $f^{\mathcal{D}} \in \mathcal{IF}$ *and* $f^{\mathcal{D}'} \in \mathcal{IF}'$*.*

In particular, the *bottom database* has an empty set of tuples and each interpreted function is undefined. Instances (key and non-key values, and interpreted functions) are defined by means of *constructor-based conditional rewriting rules.*

Definition 8 (Conditional Rewriting Rules). *A constructor-based conditional rewrite rule RW for a symbol* $H \in DS(D)$ *has the form* $H\ t_1 \ldots t_n := r \Leftarrow C$ *representing that* r is the value of $H\ t_1 \ldots t_n$, whenever the condition C holds.
In this kind of rule $(t_1, \ldots, t_n)$ *is a linear tuple (each variable in it occurs only once) with* $t_i \in CTerm_{DC}(\mathcal{V})$*,* $r \in Term_D(\mathcal{V})$*;* C *is a set of constraints of the form* $e \bowtie e', e <> e', e \not\bowtie e', e \not<> e'$*, where* $e, e' \in Term_D(\mathcal{V})$ *and* extra variables *are not allowed, i.e.* $var(r) \cup var(C) \subseteq var(\bar{t})$*.*

$Term_D(\mathcal{V})$ represents the set of *terms* or expressions over a database D, and they are built from DC, $DS(D)$ and variables of $\mathcal{V}$. Each term e represents a cone (or an ideal), in such a way that, the constraints allow to compare the cones of e and e', accordingly to the semantics of the defined operators (i.e. $\bowtie$, $<>$, $\not\bowtie$, $\not<>$). For instance, the above mentioned instances can be defined by the following rules:

```
                  ⎧ person_job john := ok.                       person_job mary := ok.
                  ⎪ person_job peter := ok.
                  ⎪ address john := add('6th Avenue', 5).         address mary := add('7th Avenue', 2).
person_job        ⎨ address peter := add('5th Avenue', 5).
                  ⎪ job_id john := lecturer.                      job_id mary := associate.
                  ⎪ job_id peter := professor.
                  ⎪ boss john := mary.                            boss john := peter.
                  ⎩ boss mary := peter.
                  ⎧ job_information lecturer := ok.         job_information associate := ok.
                  ⎪ job_information professor := ok.
job_information   ⎨ salary lecturer := retention_for_tax 1500.
                  ⎪ salary associate := retention_for_tax 2500.
                  ⎪ salary professor := retention_for_tax 4000.
                  ⎩ bonus professor := 1500.
                  ⎧ person_boss_job Name := ok ⇐ person_job Name ⋈ ok.
person_boss_job   ⎨ boss_age Name := b&a(boss Name, address (boss Name)).
                  ⎩ job_bonus Name := j&b(job_id (Name), bonus (job_id (Name))).
peter_workers     ⎧ peter_workers Name := ok ⇐ person_job Name ⋈ ok, boss Name ⋈ peter.
                  ⎩ work Name := job_id Name.
retention_for_tax { retention_for_tax Fullsalary :=  Fullsalary − (0.2 * Fullsalary).
```

The rules $R\ t_1\ \ldots\ t_k := r \Leftarrow C$, where r is a term of type `typeok`, allow the setting of $t_1, \ldots, t_k$ as key values of the relation R. `typeok` consists of a unique special value `ok` (`ok` is a shorthand of *object key*). The rules $A\ t_1\ \ldots\ t_k := r \Leftarrow C$, where $A \in NonKey(R)$, set r as the value of A for the tuple of R with key values $t_1, \ldots, t_k$. In these kinds of rules, $t_1, \ldots, t_k$ (and r) can be non-ground

Table 1. Examples of (Functional-Logic) Queries

Query	Description	Answer
	Handling of Multi-valued Attributes	
`boss X` $\bowtie$ `peter.`	*who has* `peter` *as boss?*	$\begin{cases}\texttt{Y/john}\\ \texttt{Y/mary}\end{cases}$
`address (boss X)` $\bowtie$ `Y,` `job_id X` $\not\bowtie$ `lecturer.`	*To obtain non-*`lecturer` *people and their bosses' address*	$\{$ `X/mary Y/add('5th Avenue', 5)`
	Handling of Partial Information	
`job_bonus X` $\not\Diamond$ `j&b(associate, Y).`	*To obtain people whose all jobs are equal to* `associate`*, and their salary bonuses, although they do not exist*	$\{$ `X/mary,   Y/F`
	Handling of Infinite Databases	
`select (list_of_points p(0, 0) Z)` $\bowtie$ `p(0, 2).`	*To obtain the orientation of the line from* `p(0, 0)` *to* `p(0, 2)`	$\{$ `Z/north`

values, and thus the key and non-key values are so too. Rules for the non-key attributes $A\ t_1\ \ldots\ t_k\ :=\ r\ \Leftarrow\ C$ are implicitly constrained to the form $A\ t_1\ \ldots\ t_k := r \Leftarrow R\ t_1\ \ldots\ t_k \bowtie \texttt{ok},\ C$, in order to guarantee that $t_1, \ldots, t_k$ are key values defined in a tuple of R.

As can be seen in the rules, undefined information (ni) is interpreted, whenever *there are no rules* for a given attribute. In addition, whenever the attribute is defined by rules, it is assumed that the attribute does not exist for the keys for which either the attribute is not defined or the rule conditions do not hold (i.e. nonexistent information (ne)). It fits with the failure of reduction of conditional rewriting rules [9]. Once $\bot$ and F are introduced as special cases of attribute values, the view person_boss_job will include partially undefined (pni) and partially nonexistent (pne) information. In addition, from the form of the rules for the key values of person_boss_job and peter_workers, we can consider them as views defined from person_job.

Now, we can consider (functional-logic) *queries*, which are similar to the condition of a conditional rewriting rule. For instance, table 1 shows some examples, with their corresponding meanings and expected answers.

3 Extended Relational Calculus

Next, we present the *extension of the relational calculus*, by showing its syntax, safety conditions, and, finally, its semantics.

Definition 9 (Atomic Formulas). *Given a database $D = (S, DC, IF)$, the* atomic formulas *are expressions of the form:*

1. $R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$*, where R is a schema of S, the variables x_i's are pairwise distinct, $k = nKey(R)$, and $n = nAtt(R)$*
2. $x = t$*, where $x \in \mathcal{V}$ and $t \in CTerm_{DC}(\mathcal{V})$*
3. $t \Downarrow t'$ *or* $t \Uparrow t'$*, where $t, t' \in CTerm_{DC}(\mathcal{V})$*
4. $e \lhd x$*, where $e \in Term_{DC,IF}(\mathcal{V})^2$, and $x \in \mathcal{V}$*

[2] Terms used in the calculus equations do not include schema symbols.

(1) represents *relation predicates*, (2) the *syntactic equality*, (3) the *(strong) equality and inequality equations*, which have the same meaning as the corresponding relations (see section 2, definition 1). Finally, (4) is an *approximation equation*, representing approximation values obtained from interpreted functions.

Definition 10 (Calculus Formulas). *A calculus formula φ against an instance $\mathcal{D}$ has the form $\{x_1, \ldots, x_n \mid \phi\}$, such that ϕ is a conjunction of the form $\phi_1 \wedge \ldots \wedge \phi_n$ where each ϕ_i has the form ψ or $\neg\psi$, and each ψ is an existentially quantified conjunction of atomic formulas. Variables x_i's are the* free variables *of ϕ, denoted by $free(\phi)$. Finally, variables x_i's occurring in all $R(\bar{x})$ are distinct and the same happens to variables x's occurring in equations $e \lhd x$.*

Formulas can be built from $\forall, \rightarrow, \vee, \leftrightarrow$ whenever they are logically equivalent to the defined calculus formulas. For instance, the (functional-logic) query $\mathcal{Q}_s \equiv$ `retention_for_tax X` $\bowtie$ `salary (job_id peter)` w.r.t the database schemas `person_job` and `job_information`, requests `peter`'s full salary, and obtains `X/4000` as answer. This query can be written as follows:

$$\varphi_s \equiv \{\mathtt{x} \mid (\exists \mathtt{y}_1.\exists \mathtt{y}_2.\exists \mathtt{y}_3.\exists \mathtt{y}_4.\exists \mathtt{y}_5.\ \mathtt{person\_job}(\mathtt{y}_1,\mathtt{y}_2,\mathtt{y}_3,\mathtt{y}_4,\mathtt{y}_5) \wedge \mathtt{y}_1 = \mathtt{peter} \wedge \exists \mathtt{z}_1.\exists \mathtt{z}_2.\ \exists \mathtt{z}_3.\ \mathtt{job\_information}(\mathtt{z}_1,\mathtt{z}_2,\mathtt{z}_3) \wedge \mathtt{z}_1 = \mathtt{y}_4 \wedge \exists \mathtt{u}.\ \mathtt{retention\_for\_tax}\ \mathtt{x} \lhd \mathtt{u} \wedge \mathtt{z}_2 \Downarrow \mathtt{u})\}$$

In this case, φ_s expresses to obtain the full salary, that is, `retention_for_tax` $\mathtt{x} \lhd \mathtt{u}$ and $\exists \mathtt{z}_1.\exists \mathtt{z}_2.\exists \mathtt{z}_3.\mathtt{job\_information}(\mathtt{z}_1,\mathtt{z}_2,\mathtt{z}_3) \wedge \mathtt{z}_2 \Downarrow \mathtt{u}$, for `peter`, that is, $\exists \mathtt{y}_1.\ldots\exists \mathtt{y}_5.\ \mathtt{person\_job}(\mathtt{y}_1,\ldots,\mathtt{y}_5) \wedge \mathtt{y}_1 = \mathtt{peter} \wedge \mathtt{z}_1 = \mathtt{y}_4$

In database theory, it is known that any query language must ensure the property of *domain independence* [2]. This has led to define syntactic conditions, called *safety conditions*, over the queries in such a way that the so-called *safe queries* guarantee this property. For example, in [2], the variables occurring in formulas must be *range restricted.* In our case, we generalize the notion of *range restricted* to c-terms. In addition, we require *safety conditions over atomic formulas*, and *conditions over bounded variables.*

Now, given a calculus formula φ against a database D, we define the sets: $formula\_key(\varphi) = \{x_i \mid there\ \ exists\ R(x_1, \ldots, x_i, \ldots, x_n)\ occurring\ in\ \varphi\ and\ 1 \leq i \leq nKey(R)\}$, $formula\_nonkey(\varphi) = \{x_j \mid there\ exists\ \ R(x_1, \ldots, x_j, \ldots, x_n)\ \ occurring\ in\ \ \varphi\ and\ nKey(R)+1 \leq j \leq n\}$, and $approx(\varphi) = \{x \mid there\ exis\ ts\ \ e\ \lhd\ x\ occurring\ in\ \varphi\}$.

Definition 11 (Safe Atomic Formulas). *An atomic formula is safe in φ in the following cases:*

- *$R(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n)$ is safe, if the variables $x_1, \ldots, x_n$ are bound in φ, and for each x_i, $i \leq nKey(R)$, there exists one equation $x_i = t_i$ in φ*
- *$x = t$ is safe, if the variables occurring in t are distinct from the variables of $formula\_key(\varphi)$, and $x \in formula\_key(\varphi)$*
- *$t \Downarrow t'$ and $t \Uparrow t'$ are safe, if the variables occurring in t and t' are distinct from the variables of $formula\_key(\varphi)$*

Table 2. Examples of Calculus Formulas

Query	Calculus Formula
$\mathtt{boss\ X} \bowtie \mathtt{peter.}$	$\{\mathtt{x} \mid (\exists \mathtt{y_1}.\exists \mathtt{y_2}.\exists \mathtt{y_3}.\exists \mathtt{y_4}.\exists \mathtt{y_5}.\ \mathtt{person\_job}(\mathtt{y_1},\mathtt{y_2},\mathtt{y_3},\mathtt{y_4},\mathtt{y_5}) \wedge \mathtt{y_1} = \mathtt{x} \wedge \mathtt{y_5} \Downarrow \mathtt{peter})\}$
$\mathtt{address\ (boss\ X)} \bowtie \mathtt{Y},$ $\mathtt{job\_id\ X} \not\bowtie \mathtt{lecturer.}$	$\{\mathtt{x},\mathtt{y} \mid (\exists \mathtt{y_1}.\exists \mathtt{y_2}.\exists \mathtt{y_3}.\exists \mathtt{y_4}.\exists \mathtt{y_5}.\ \mathtt{person\_job}(\mathtt{y_1},\mathtt{y_2},\mathtt{y_3},\mathtt{y_4},\mathtt{y_5}) \wedge \mathtt{y_1} = \mathtt{x} \ \wedge \exists \mathtt{z_1}.\exists \mathtt{z_2}.\exists \mathtt{z_3}.\exists \mathtt{z_4}.\exists \mathtt{z_5}.\mathtt{person\_job}(\mathtt{z_1},\mathtt{z_2},\mathtt{z_3},\mathtt{z_4},\mathtt{z_5}) \ \wedge\ \mathtt{z_1} = \mathtt{y_5} \wedge \mathtt{z_3} \Downarrow \mathtt{y}) \wedge (\forall \mathtt{v_4}.((\exists \mathtt{v_1}.\exists \mathtt{v_2}.\exists \mathtt{v_3}.\exists \mathtt{v_5}.\ \mathtt{person\_job}(\mathtt{v_1},\mathtt{v_2},\mathtt{v_3},\mathtt{v_4},\mathtt{v_5}) \wedge \mathtt{v_1} = \mathtt{x}) \rightarrow \neg \mathtt{v_4} \Downarrow \mathtt{lec-turer}))\}$
$\mathtt{job\_bonus\ X} \not\lhd\!\rhd$ $\mathtt{j\&b(associate,Y).}$	$\{\mathtt{x},\mathtt{y} \mid (\forall \mathtt{y_3}.(\exists \mathtt{y_1}.\exists \mathtt{y_2}.\ \mathtt{person\_boss\_job}(\mathtt{y_1},\mathtt{y_2},\mathtt{y_3}) \wedge \mathtt{y_1} = \mathtt{x}) \rightarrow \neg \mathtt{y_3} \Uparrow \mathtt{j\&b}(\mathtt{associate},\mathtt{y}))\}$
$\mathtt{select\ (list\_of\_points}$ $\mathtt{p(0,0)\ Z)} \bowtie \mathtt{p(0,2).}$	$\{\mathtt{z} \mid (\exists \mathtt{y_1}.\exists \mathtt{y_2}.\exists \mathtt{y_3}.\exists \mathtt{y_4}.\exists \mathtt{y_5}.\ \mathtt{2Dline}(\mathtt{y_1},\mathtt{y_2},\mathtt{y_3},\mathtt{y_4},\mathtt{y_5}) \wedge \mathtt{y_1} = \mathtt{p(0,0)} \ \wedge \mathtt{y_2} = \mathtt{z} \ \wedge\ \exists \mathtt{u}.\mathtt{select}\ \mathtt{y_5} \lhd \mathtt{u} \ \wedge\ \mathtt{u} \Downarrow \mathtt{p(0,2)})\}$

– *$e \lhd x$ is safe, if the variables occurring in e are distinct from the variables of $formula\_key(\varphi)$, and x is bound in φ*

Definition 12 (Ranged Restricted C-Terms of Calculus Formulas). *A c-term is range restricted in a calculus formula φ if: either (a) it occurs in $formula\_key(\varphi) \cup formula\_nonkey(\varphi)$, or (b) there exists one equation $e \Diamond_c e'$ ($\Diamond_c \equiv =$, $\Uparrow$, $\Downarrow$, or $\lhd$) in φ, such that it belongs to $cterms(e)$ (resp. $cterms(e')$) and every c-term of e' (resp. e) is ranged restricted in φ.*

In the above definition, $cterms(e)$ denotes the set of c-terms occurring in e. Range restricted c-terms are variables occurring in the scope of a relation predicate or c-terms compared (by means of syntactic and strong (in)equalities, and approximation equations) with variables in the scope of a relation predicate. Therefore, all of them take values from the schema instance.

Definition 13 (Safe Formulas). *A calculus formula φ against a database D is safe, if all c-terms and atomic formulas occurring in φ are range restricted and safe, respectively and, in addition, the only bounded variables are variables of $formula\_key(\varphi) \cup formula\_nonkey(\varphi) \cup approx(\varphi)$.*

For instance, the previous $\varphi_\mathtt{s}$ is *safe*, given that the constant `peter` is *range restricted* (by means of $\mathtt{y_1} = \mathtt{peter}$), and the variables $\mathtt{u},\mathtt{x}$ are also *range restricted* (by means of $\mathtt{retention\_for\_tax\ x} \lhd \mathtt{u}$ and $\mathtt{z_2} \Downarrow \mathtt{u}$). Once we have defined the conditions over the built formulas, we guarantee that they represent "queries" against a database. Negation can be used in combination with strong (in)equality relations; for instance, the calculus formula

$$\varphi_0 \equiv \neg \exists \mathtt{x_1}.\mathtt{x_2}.\mathtt{x_3}.\mathtt{x_4}.\mathtt{x_5}.\mathtt{person\_job}(\mathtt{x_1},\ldots,\mathtt{x_5}) \wedge \mathtt{x_1} = \mathtt{mary} \wedge \mathtt{x_5} \Downarrow \mathtt{y}$$

requests people who are not a `mary`'s boss. In this case, `y` is restricted to be a value of the column `boss` of the relation `person_job`. In this case, the answers are {`y/mary`} and {`y/F`}. Table 2 shows *(safe) calculus formulas* built from the *queries* presented in table 1. Now, we define the answers of a calculus formula. With this aim, we need to define the following notions.

Definition 14 (Denotation of Terms). *The denoted values for $e \in Term_{DC,IF}$ $(\mathcal{V})$ in an instance $\mathcal{D}$ of a database $D = (S, DC, IF)$ w.r.t. a substitution θ, represented by $\| e \|^{\mathcal{D}}\theta$, are defined as follows: $\| X \|^{\mathcal{D}}\theta =_{def} < X\theta >$, for $X \in \mathcal{V}$;*

$[\![c(e_1, \ldots, e_n)]\!]^{\mathcal{D}}\theta =_{def} < c([\![e_1]\!]^{\mathcal{D}}\theta, \ldots, [\![e_n]\!]^{\mathcal{D}}\theta) >$[3], *for all* $c \in DC^n$*; and* $[\![f\ e_1\ \ldots\ e_n\ ]\!]^{\mathcal{D}}\theta =_{def} f^{\mathcal{D}}\ [\![e_1]\!]^{\mathcal{D}}\theta\ \ldots\ [\![e_n]\!]^{\mathcal{D}}\theta$ *, for all* $f \in IF^n$

The denoted values for a term represent a cone (resp. ideal), containing the set of values which defines a non-deterministic (resp. deterministic) interpreted function.

Definition 15 (Active Domain of Terms). *The active domain of a term* $e \in Term_{DC,IF}\ (\mathcal{V})$ *in a calculus formula* φ *w.r.t. an instance* $\mathcal{D}$ *of database* $D = (S, DC, IF)$*, which is denoted by* $adom(e, \mathcal{D})$*, is defined as follows:*

1. $adom(x, \mathcal{D}) =_{def} \bigcup_{\psi \in Subst_{DC,\perp,\mathsf{F}}, (V_1,\ldots,V_i,\ldots,V_n) \in \mathcal{R}} V_i\psi$*, if there exists an atomic formula* $R(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n)$ *in* φ*;* $adom(x, \mathcal{D}) =_{def} adom(e, \mathcal{D})$ *if there exists an approximation equation* $e \lhd x$ *in* φ*; and* $< \perp >$*, otherwise*
2. $adom(c(e_1, \ldots, e_n), \mathcal{D}) =_{def} < c(adom(e_1, \mathcal{D}), \ldots, adom(e_n, \mathcal{D})) >$*, if* $c \in DC^n$
3. $adom(f\ e_1 \ldots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \ldots adom(e_n, \mathcal{D})$*, if* $f \in IF^n$

The active domain of key and non-key variables contains the complete set of values of the column representing the corresponding key and non-key attributes. In the case of approximation variables, the active domain contains the complete set of values of the interpreted function. For example, the active domain of $\mathtt{x_5}$ in $\mathtt{person\_job}(\mathtt{x_1}, \ldots, \mathtt{x_5})$ is $\{\mathtt{mary}, \mathtt{peter}, \mathsf{F}\}$. The active domain is used in order to restrict the answers of a calculus formula w.r.t the schema instance. For instance the previous formula φ_0 restricts $\mathtt{y}$ to be valued in the active domain of $\mathtt{x_5}$, which is $\{\mathtt{peter}, \mathtt{mary}, \mathsf{F}\}$, and, therefore, obtaining as answers $\theta_1 = \{\mathtt{y}/\mathtt{mary}\}$ and $\theta_2 = \{\mathtt{y}/\mathsf{F}\}$. Remark that the isolated equation $\neg \mathtt{x_5} \Downarrow \mathtt{y}$ holds for $\{\mathtt{x_5}/\mathtt{peter}, \mathtt{y}/\mathtt{lecturer}\}$, w.r.t. $\not\downarrow$. However the value $\mathtt{lecturer}$ is not in the active domain of $\mathtt{x_5}$.

Remark that we have to instantiate the schema instance, whenever it includes variables (see case (1) of the above definition).

Definition 16 (Satisfiability). *Given a calculus formula* $\{\bar{x} \mid \phi\}$*, the* satisfiability *of* ϕ *in an instance* $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ *under a substitution* θ*, such that* $dom(\theta) \subseteq free(\phi)$*, (in symbols* $(\mathcal{D}, \theta) \models_C \phi$*) is defined as follows:*

- $(\mathcal{D}, \theta) \models_C R(x_1, \ldots, x_n)$*, if there exists* $(V_1, \ldots, V_n) \in \mathcal{R}$ $(\mathcal{R} \in \mathcal{S})$*, such that* $x_i\theta \in V_i\psi$ *for every* $1 \leq i \leq n$*, where* $\psi \in Subst_{DC,\perp,\mathsf{F}}$.
- $(\mathcal{D}, \theta) \models_C x = t$*, if* $x\theta \equiv t\theta$*;* $(\mathcal{D}, \theta) \models_C t \Downarrow t'$*, if* $t\theta \downarrow t'\theta$ *and,* $t\theta, t'\theta \in adom(t, \mathcal{D}) \cup adom(t', \mathcal{D})$*;* $(\mathcal{D}, \theta) \models_C t \Uparrow t'$*, if* $t\theta \uparrow t'\theta$ *and,* $t\theta, t'\theta \in adom(t, \mathcal{D}) \cup adom(t', \mathcal{D})$*; and* $(\mathcal{D}, \theta) \models_C e \lhd x$*, if* $x\theta \in [\![e]\!]^{\mathcal{D}}\theta$
- $(\mathcal{D}, \theta) \models_C \phi_1 \wedge \phi_2$*, if* $\mathcal{D}$ *satisfies* ϕ_1 *and* ϕ_2 *under* θ
- $(\mathcal{D}, \theta) \models_C \exists x.\phi$*, if there exists* v*, such that* $\mathcal{D}$ *satisfies* ϕ *under* $\theta \cdot \{x/v\}$
- $(\mathcal{D}, \theta) \models_C \neg\phi$*, if* $\mathcal{D}$ *does not satisfy* ϕ *under the substitution* θ

[3] To simplify denotation, we write $\{c(t_1, \ldots, t_n) \mid t_i \in C_i\}$ as $c(C_1, \ldots, C_n)$ and $\{f(t_1, \ldots, t_n) \mid t_i \in C_i\}$ as $f(C_1, \ldots, C_n)$ where $C_i's$ are certain cones.

In the formula φ_0, $\texttt{adom}(\texttt{x}_5, \mathcal{D}) = \{\texttt{peter}, \texttt{mary}, \texttt{F}\}$ and $\texttt{adom}(\texttt{y}, \mathcal{D}) = \{\perp\}$. Moreover, $\theta_1 = \{\texttt{y}/\texttt{mary},\ \texttt{x}_5/\texttt{peter}\}$ and $\theta_2 = \{\texttt{y}/\texttt{F}, \texttt{x}_5/\texttt{peter}\}$ holds $\texttt{Y}\theta_1, \texttt{Y}\theta_2 \in \texttt{adom}(\texttt{x}_5, \mathcal{D}) \cup \texttt{adom}(\texttt{y}, \mathcal{D})$; and $\texttt{x}_5\theta_1 \not\Downarrow \texttt{y}\theta_1$ and $\texttt{x}_5\theta_2 \not\Downarrow \texttt{y}\theta_2$ are satisfied.

Given a calculus formula $\varphi \equiv \{x_1, \ldots, x_n \mid \phi\}$, we define the *set of answers* of φ w.r.t. an instance $\mathcal{D}$, denoted by $Ans(\mathcal{D}, \varphi)$, as follows: $Ans(\mathcal{D}, \{x_1, \ldots, x_n|\phi\}) = \{(x_1\theta, \ldots, x_n\theta)|\theta \in Subst_{DC,\perp,\texttt{F}}$ *and* $(\mathcal{D}, \theta) \models_C \phi\}$. Finally, the property of *domain independence* is defined as follows.

Definition 17 (Domain Independence). *A calculus formula φ is domain independent whenever: (a) if the instance $\mathcal{D}$ is finite, then $Ans(\mathcal{D}, \varphi)$ is finite and (b) given two ground instances $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ and $\mathcal{D}' = (\mathcal{S}, \mathcal{DC}', \mathcal{IF}')$ such that $\mathcal{DC}' \supseteq \mathcal{DC}$, and $\mathcal{IF}' \supseteq \mathcal{IF}$ w.r.t. $\mathcal{S}$, then $Ans(\mathcal{D}, \varphi) = Ans(\mathcal{D}', \varphi)$.*

The case (a) establishes that the set of answers is finite whenever $\mathcal{S}$ is finite; and (b) states that the output relation (i.e. answers) depends on the input schema instance $\mathcal{S}$, and not on the domain; that is, data constructors (i.e. $\mathcal{DC}$) and interpreted functions (i.e. $\mathcal{IF}$).

Theorem 1 (Domain Independence of Calculus Formulas). *Safe calculus formulas are domain independent.*

4 Calculus Formulas and Functional Logic Queries Equivalence

In this section, we establish the equivalence between the relational calculus and the functional-logic query language. With this aim, we need to define analogous safety conditions over functional-logic queries. The set of *query keys* of a key attribute $A_i \in Key(R)$ ($R \in S$) occurring in a term $e \in Term_D(\mathcal{V})$ and denoted by $query\_key(e, A_i)$, is defined as $\{t_i \mid H\ e_1 \ldots t_i \ldots e_k$ *occurs in* $e, H \in \{R\} \cup NonKey(R)\}$. Now, $query\_key(\mathcal{Q}) = \cup_{A_i \in Key(R)} query\_key(\mathcal{Q}, A_i)$ where $query\_key(\mathcal{Q}, A_i) = \cup_{e \diamondsuit_q e' \in \mathcal{Q}}\ (query\_key(e, A_i) \cup query\_key(e', A_i))$ ($\diamondsuit_q \equiv \bowtie$, $<>$, $\not\bowtie$, or $\not<>$).

A c-term t is *range restricted* in $\mathcal{Q}$, if: either (a) t belongs to $\cup_{s \in query\_key(\mathcal{Q})}$ *cterms*(s) or (b) there exists a constraint $e \diamondsuit_q e'$, such that t belongs to *cterms*(e) (resp. *cterms*(e')) and every c-term occurring in e' (resp. e) is *range restricted*.

Definition 18 (Safe Queries). *A query $\mathcal{Q}$ is safe if* all c-terms occurring in $\mathcal{Q}$ are range restricted.

For instance, let's consider the following query (corresponding to φ_s previously mentioned): $\mathcal{Q}_s \equiv \texttt{retention\_for\_tax X} \bowtie \texttt{salary(job\_id peter)}$. $\mathcal{Q}_s$ is *safe*, given that the constant `peter` is *range restricted*, and thus the variable `X` is also *ranged restricted*. Analogously to the calculus formulas, we need to define the denoted values and the active domain of a database term (which includes relation names and non-key attributes) in a functional-logic query.

Definition 19 (Denotation of Database Terms). *The denotation of $e \in Term_D(\mathcal{V})$ in an instance $\mathcal{D} = (\mathcal{S}, \mathcal{DC}, \mathcal{IF})$ of database $D = (S, DC, IF)$ under θ, is defined as follows:*

1. *$[\![R\ e_1\ \dots\ e_k\ ]\!]^{\mathcal{D}}\theta =_{def} < \mathbf{ok} >$, if $([\![e_1]\!]^{\mathcal{D}}\theta,\ \dots\ ,[\![e_k]\!]^{\mathcal{D}}\theta) = (V_1\psi, \dots, V_k\psi)$ and there exists a tuple $(V_1, \dots, V_k, V_{k+1}, \dots, V_n) \in \mathcal{R}$, where $\psi \in Subst_{DC,\perp,\mathsf{F}}$, $\mathcal{R} \in \mathcal{S}$, $k = nKey(R)$; $< \perp >$ otherwise, for all $R \in S$;*
2. *$[\![A_i\ e_1\ \dots\ e_k\ ]\!]^{\mathcal{D}}\theta =_{def} V_i\psi$, if $([\![e_1]\!]^{\mathcal{D}}\theta,\ \dots\ ,[\![e_k]\!]^{\mathcal{D}}\theta) = (V_1\psi, \dots, V_k\psi)$ and there exists a tuple $(V_1, \dots, V_k, V_{k+1}, \dots, V_i, \dots, V_n) \in \mathcal{R}$, where $\psi \in Subst_{DC,\perp,\mathsf{F}}$, $\mathcal{R} \in \mathcal{S}$, and $i > nKey(R) = k$; $< \perp >$ otherwise, for all $A_i \in NonKey(R)$;*
3. *And the rest of cases as in definition 14.*

Definition 20 (Active Domain of Database Terms). *The active domain of $e \in Term_D(\mathcal{V})$ w.r.t. an instance $\mathcal{D}$, and a query $\mathcal{Q}$, denoted by $adom(e, \mathcal{D})$, is defined as follows:*

- *$adom(t, \mathcal{D}) =_{def} \bigcup_{\psi \in Subst_{DC,\perp,\mathsf{F}},(V_1,\dots,V_i,\dots,V_n)\in\mathcal{R}} V_i\psi$, if $t \in query\_key(\mathcal{Q}, A_i)$, $A_i \in Key(R)$; and $< \perp >$ otherwise, for all $t \in CTerm_{\perp,\mathsf{F}}(\mathcal{V})$*
- *$adom(c(e_1, \dots, e_n), \mathcal{D}) =_{def} <c(adom(e_1, \mathcal{D}), \dots, adom(e_n, \mathcal{D}))>$, for all $c \in DC^n$*
- *$adom(f\ e_1 \dots e_n, \mathcal{D}) =_{def} f^{\mathcal{D}} adom(e_1, \mathcal{D}) \dots adom(e_n, \mathcal{D})$, for all $f \in IF^n$*
- *$adom(R\ e_1 \dots e_k, \mathcal{D}) =_{def} < \mathbf{ok} >$, for all $R \in S$*
- *$adom(A_i\ e_1 \dots e_k, \mathcal{D}) =_{def} \bigcup_{\psi \in Subst_{DC,\perp,\mathsf{F}},(V_1,\dots,V_i,\dots,V_n)\in\mathcal{R}} V_i\psi$, for all $A_i \in NonKey(R)$*

Both sets are also used for defining the set of query answers.

Definition 21 (Query Answers). *θ is an answer of $\mathcal{Q}$ w.r.t. $\mathcal{D}$ (in symbols $(\mathcal{D}, \theta) \models_Q \mathcal{Q}$) in the following cases:*

- *$(\mathcal{D}, \theta) \models_Q e \bowtie e'$, if there exist $t \in [\![e]\!]^{\mathcal{D}}\theta$ and $t' \in [\![e']\!]^{\mathcal{D}}\theta$, such that $t \downarrow t'$, and $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$.*
- *$(\mathcal{D}, \theta) \models_Q e <> e'$, if there exist $t \in [\![e]\!]^{\mathcal{D}}\theta$ and $t' \in [\![e']\!]^{\mathcal{D}}\theta$, such that $t \uparrow t'$, and $t, t' \in adom(e, \mathcal{D}) \cup adom(e', \mathcal{D})$.*
- *$(\mathcal{D}, \theta) \models_Q e \not\bowtie e'$ if $(\mathcal{D}, \theta) \not\models_Q e \bowtie e'$; and $(\mathcal{D}, \theta) \models_Q e \not<> e'$, if $(\mathcal{D}, \theta) \not\models_Q e <> e'$.*

Now, the set of answers of a safe query $\mathcal{Q}$ w.r.t. an instance $\mathcal{D}$, denoted by $Ans(\mathcal{D}, \mathcal{Q})$, is defined as follows: $Ans(\mathcal{D}, \mathcal{Q}) =_{def} \{(X_1\theta, \dots, X_n\theta) \mid Dom(\theta) \subseteq var(\mathcal{Q}), (\mathcal{D}, \theta) \models_Q \mathcal{Q}, var(\mathcal{Q}) = \{X_1, \dots, X_n\}\}$. With the previous definitions, we can state the following result.

Theorem 2 (Queries and Calculus Formulas Equivalence). *Let $\mathcal{D}$ be an instance, then:*

- *given a safe query $\mathcal{Q}$ against $\mathcal{D}$, there exists a safe calculus formula $\varphi_{\mathcal{Q}}$ such that $Ans(\mathcal{D}, \mathcal{Q}) = Ans(\mathcal{D}, \varphi_{\mathcal{Q}})$*
- *given a safe calculus formula φ against $\mathcal{D}$, there exists a safe query $\mathcal{Q}_\varphi$ such that $Ans(\mathcal{D}, \varphi) = Ans(\mathcal{D}, \mathcal{Q}_\varphi)$*

Proof Sketch:
The idea is to transform a safe query into the corresponding safe calculus formula, and viceversa, by applying the set of rules of table 3. The rules distinguish two parts $\varphi \oplus \mathcal{Q}$, where φ is a calculus formula and $\mathcal{Q}$ the query.

Table 3. Transformation Rules

$$\textbf{(1)}\ \frac{\phi \wedge \exists \bar{z}.\psi \oplus e \bowtie e', \mathcal{Q}}{\phi \wedge \exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Downarrow y \oplus \mathcal{Q}}$$

$$\textbf{(2)}\ \frac{\phi \wedge \neg\exists \bar{z}.\psi \oplus e \not\bowtie e', \mathcal{Q}}{\phi \wedge \neg\exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Downarrow y \oplus \mathcal{Q}}$$

$$\textbf{(3)}\ \frac{\phi \wedge \exists \bar{z}.\psi \oplus e \Diamond e', \mathcal{Q}}{\phi \wedge \exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Uparrow y \oplus \mathcal{Q}}$$

$$\textbf{(4)}\ \frac{\phi \wedge \neg\exists \bar{z}.\psi \oplus e \not\Diamond e', \mathcal{Q}}{\phi \wedge \neg\exists \bar{z}.\exists x.\exists y.\psi \wedge e \lhd x \ \wedge\ e' \lhd y \ \wedge\ x \Uparrow y \oplus \mathcal{Q}}$$

$$\textbf{(5)}\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge R\ e_1 \ldots e_k \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1.\ldots.\exists y_k.\psi \wedge R(y_1, \ldots, y_k, \ldots, y_n) \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_k \lhd y_k[x|\mathtt{ok}] \oplus \mathcal{Q}}$$
$\%\ \mathtt{R} \in \mathtt{S}$

$$\textbf{(6)}\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge A_i\ e_1 \ldots e_k \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1.\ldots.\exists y_k.\psi \wedge R(y_1, \ldots, y_k, \ldots, y_i, \ldots, y_n) \ \wedge\ e_1 \lhd y_1 \wedge \ldots \wedge e_k \lhd y_k \wedge y_i \lhd x \oplus \mathcal{Q}}$$
$\%\ \mathtt{A_i} \in \mathtt{NonKey(R)}$

$$\textbf{(7)}\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge f\ e_1 \ldots e_n \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots y_n.\psi \wedge f\ y_1 \ldots y_n \lhd x \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_n \lhd y_n \oplus \mathcal{Q}}$$

$$\textbf{(8)}\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge c(e_1, \ldots, e_n) \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\exists y_1 \ldots y_n.\psi \wedge c(y_1, \ldots, y_n) \lhd x \wedge e_1 \lhd y_1 \wedge \ldots \wedge e_n \lhd y_n \oplus \mathcal{Q}}$$
$\%\ \mathtt{c(e_1, \ldots, e_n)}$ `is not a cterm`

$$\textbf{(9)}\ \frac{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge t \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\psi \wedge x = t \oplus \mathcal{Q}}$$
$\%\ \mathtt{x} \in \mathtt{formula\_key}(\phi \wedge (\neg)\exists \bar{z}.\psi \wedge t \lhd x)$

$$\textbf{(10)}\ \frac{\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge t \lhd x \oplus \mathcal{Q}}{\phi \wedge (\neg)\exists \bar{z}.\psi[x|t] \oplus \mathcal{Q}}$$
$\%\ \mathtt{x} \notin \mathtt{formula\_key}(\phi \wedge (\neg)\exists \bar{z}.\exists x.\psi \wedge t \lhd x)$

5 Least Induced Database

To put an end, we will show how instances can be obtained from a set of conditional rewriting rules, by means of a *fixed point operator*, which computes the *least database induced* induced from a set of rules.

Definition 22 (Fixed Point Operator). *Given $\mathcal{A} = (\mathcal{S}^A, \mathcal{DC}^A, \mathcal{IF}^A)$ instance of a database schema $D = (S, DC, IF)$, we define a fixed point operator $T_{\mathcal{P}}(\mathcal{A}) = \mathcal{B} = (\mathcal{S}^B, \mathcal{DC}^A, \mathcal{IF}^B)$ as follows:*

– *For each schema $R(A_1, \ldots, A_n)$: $(V_1, \ldots, V_k, V_{k+1}, \ldots, V_n) \in \mathcal{R}^B, \mathcal{R}^B \in \mathcal{S}^{\mathcal{B}}$, iff $<$ **ok** $>\in T_{\mathcal{P}}(\mathcal{A}, R)$ $(V_1, \ldots, V_k)$, $k = nKey(R)$,and $V_i = T_{\mathcal{P}}(\mathcal{A}, A_i)$ $(V_1, \ldots, V_k)$ for every $i \geq nKey(R) + 1$,*
– *For each $f \in IF$ and $t_1, \ldots, t_n \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, $f^{\mathcal{B}}(t_1, \ldots, t_n) = T_{\mathcal{P}}(\mathcal{A}, f)$ $(t_1, \ldots, t_n)$, $f^{\mathcal{B}} \in \mathcal{IF}^{\mathcal{B}}$*

where given a symbol $H \in DS(D)$ and $s_1, \ldots s_n \in CTerm_{DC,\perp,\mathsf{F}}(\mathcal{V})$, we define:

$$\begin{aligned} T_{\mathcal{P}}(\mathcal{D}, H)(s_1, \ldots, s_n) =_{def} \{\ & [\![r]\!]^{\mathcal{D}}\theta \mid \textit{if there exist } H\ \bar{t}\ := r \Leftarrow C \textit{ and } \theta, \\ & \quad \textit{such that } s_i \in [\![t_i]\!]^{\mathcal{D}}\theta \textit{ and } (\mathcal{D}, \theta) \models_Q C\} \\ \cup \{\ & \mathsf{F} \mid \textit{if there exists } H\ \bar{t}\ := r \Leftarrow C, \textit{ such that} \\ & \quad \textit{for some } i \in \{1, \ldots, n\},\ s_i \neq t_i\} \\ \cup \{\ & \mathsf{F} \mid \textit{if there exist } H\ \bar{t}\ := r \Leftarrow C \textit{ and } \theta, \\ & \quad \textit{such that } s_i \in [\![t_i]\!]^{\mathcal{D}}\theta \textit{ and } (\mathcal{D}, \theta) \not\models_Q C\} \\ \cup \{\ & \perp \mid \textit{otherwise}\} \end{aligned}$$

Starting from the bottom instance, then the fixed point operator computes a chain of database instances $A \sqsubseteq A' \sqsubseteq A'', \ldots$ such that the fixed point is the least instance induced by a set of rules. Finally, the following result establishes that the instance computed by means of the proposed fixed point operator is the least one satisfying the set of conditional rewriting rules.

Theorem 3 (Least Induced Database).

– *The fixed point operator $T_{\mathcal{P}}$ has a least fixed point $\mathcal{L} = \mathcal{D}^{\omega}$ where $\mathcal{D}^0$ is the bottom instance and $\mathcal{D}^{k+1} = T_{\mathcal{P}}(\mathcal{D}^k)$*
– *For each safe query $\mathcal{Q}$ and θ: $(\mathcal{L}, \theta) \models_Q \mathcal{Q}$ iff $(\mathcal{D}, \theta) \models_Q \mathcal{Q}$ for each $\mathcal{D}$ satisfying the set of rules.*

6 Conclusions and Future Work

We have studied here how to express queries by means of an (extended) relational calculus in a functional logic language integrating databases. We have shown suitable properties for such language, which are summarized in the domain independence property. As future work, we propose two main lines of research: the study of an extension of our relation calculus to be used, also, as data definition language, and the implementation of the language.

References

1. S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Values. *VLDB*, 4(4):727–794, 1995.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
3. J. M. Almendros-Jiménez and A. Becerra-Terón. A Framework for Goal-Directed Bottom-Up Evaluation of Functional Logic Programs. In *Proc. of FLOPS*, LNCS 2024, pages 153–169. Springer, 2001.
4. J. M. Almendros-Jiménez and A. Becerra-Terón. A Relational Algebra for Functional Logic Deductive Databases. In *To Appear in Procs. of Perspectives of System Informatics*, LNCS. Springer, 2003.
5. J. M. Almendros-Jiménez, A. Becerra-Terón, and J. Sánchez-Hernández. A Computational Model for Funtional Logic Deductive Databases. In *Proc. of ICLP*, LNCS 2237, pages 331–347. Springer, 2001.
6. J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *JLP*, 1(40):47–87, 1999.
7. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *JLP*, 19,20:583–628, 1994.
8. P. Kanellakis and D. Goldin. Constraint Query Algebras. *Constraints*, 1(1–2):45–83, 1996.
9. F. J. López-Fraguas and J. Sánchez-Hernández. Proving Failure in Functional Logic Programs. In *Proc. of the CL*, LNCS 1861, pages 179–193. Springer, 2000.
10. P. Z. Revesz. Safe Query Languages for Constraint Databases. *TODS*, 23(1):58–99, 1998.

Automatic Visualization of Recursion Trees: a Case Study on Generic Programming

Alcino Cunha

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
`alcino@di.uminho.pt`

Abstract. Although the principles behind generic programming are already well understood, this style of programming is not widespread and examples of applications are rarely found in the literature. This paper addresses this shortage by presenting a new method, based on generic programming, to automatically visualize recursion trees of functions written in Haskell. Crucial to our solution is the fact that almost any function definition can be automatically factorized into the composition of a fold after an unfold of some intermediate data structure that models its recursion tree. By combining this technique with an existing tool for graphical debugging, and by extensively using Generic Haskell, we achieve a rather concise and elegant solution to this problem.

1 Introduction

A generic or polytypic function is defined by induction on the structure of types. It is defined once and for all, and can afterwards be re-used for any specific data type. The principles behind generic programming are already well understood [2], and several languages supporting this concept have been developed, suck as PolyP [13] or Generic Haskell [3]. Unfortunately, this style of programming is not widespread, and we rarely find in the literature descriptions of applications developed in these languages.

This paper addresses this shortage by presenting a case study on generic programming. The problem that we are trying to solve is to automatically and graphically visualize the recursion tree of a Haskell [14] function definition. This problem does not rise any particular difficulties, however it will be shown that the use of generic programming allows us to achieve a rather concise and elegant solution. This solution was integrated in a tool that is of practical interest for the area of program understanding, specially on an educational setting.

At the core of our solution lies an algorithm that automatically factorizes a recursive definition into the composition of a fold and an unfold of an intermediate data structure. This algorithm was first presented by Hu, Iwasaki, and Takeichi in [12], where it was applied to program deforestation. A well known side-effect of the factorization is that the intermediate data structure models the recursion tree of the original definition. The visualization of this structure is defined generically on top of GHood [18], a system to graphically trace Haskell programs.

This paper is structured as follows. In section 2 we informally show how the well-known fold and unfold functions on lists can be generalized to arbitrary data types. This generalization is essential for understanding the factorization algorithm. Section 3 introduces some theoretical concepts behind the idea of programming with recursion patterns. In section 4 we show how these recursion patterns can be defined once and for all using Generic Haskell. These generic definitions simplify the implementation, since we no longer have to derive the specific recursion patterns to operate on the intermediate data types. In section 5 we present the factorization algorithm very briefly. In section 6 we show how we can use GHood to observe intermediate data structures, and in section 7 we generalize the observation mechanism in order to animate any possible recursion tree. The last section presents some concluding remarks. We assume that the reader has some familiarity with the language Haskell.

2 Recursion Patterns Informally

Each inductive data type is characterized by a standard way of recursively consuming and producing its values according to its shape. The standard recursion pattern for consuming values is usually known as fold or catamorphism. In the case of lists this corresponds to the standard Haskell function `foldr`.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

The generalization to other data types is straightforward. Let us suppose that we want to fold over binary trees.

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

Similarly to `foldr`, this fold must receive as an argument the value to return when we reach a leaf, and a function to apply when consuming a node. This function has to process the result of two recursive calls since the data type is birecursive. In order to make explicit the duality with the (yet to be presented) unfolds, we group both parameters in a single function with the help of the data type `Maybe`.

```
foldT :: (Maybe (a,b,b) -> b) -> Tree a -> b
foldT g Leaf         = g Nothing
foldT g (Node x l r) = g (Just (x, foldT g l, foldT g r))
```

The dual of fold is the unfold or anamorphism. Although already known for a long time it is still not very used by programmers [6]. This recursion pattern encodes a standard way of producing a value of a given data type, and for lists it is defined as `unfoldr` in one of the standard libraries of Haskell.

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of Nothing    -> []
                          Just (a,b) -> a : unfoldr f b
```

The argument function `f` dictates the generation of the list. If it returns `Nothing` the generation stops yielding the empty list. Otherwise it should return both the value to put at the head of the list being constructed, and a seed to carry on with the generation. The generalization to trees is again straightforward.

```
unfoldT :: (b -> Maybe (a,b,b)) -> b -> Tree a
unfoldT h x = case (h x) of
                Nothing      -> Leaf
                Just (x,l,r) -> Node x (unfoldT h l) (unfoldT h r)
```

The composition of a fold after an unfold is known as a hylomorphism [16]. Note that since the fixed point operator can be defined as a hylomorphism, this recursion pattern can express any recursive function [17]. For a wide class of function definitions the intermediate data structures correspond to their recursion trees. For example, quick-sort can be easily implemented as a hylomorphism by using a binary search tree as intermediate data structure [1]. The unfold should build a search tree containing the elements of the input list, and the fold just traverses it inorder.

```
qsort :: (Ord a) => [a] -> [a]
qsort = (foldT g) . (unfoldT h)
    where h []     = Nothing
          h (x:xs) = Just (x, filter (<=x) xs, filter (>x) xs)
          g Nothing        = []
          g (Just (x,l,r)) = l ++ [x] ++ r
```

3 Data Types as Fixed Points of Functors

One of the tasks of the method presented in the previous section, is the explicit definition of the folds and unfolds for each new data type. However, it is possible to avoid this task by appealing to the theoretical concepts behind data types and the respective recursion patterns. The framework of the following presentation is the category $\mathcal{CPO}$ of complete partial orders and continuous functions.

The first insight is that data types can be modeled as least fixed points of functors. Given a monofunctor F which is locally continuous, there exists a data type μF and two strict functions $\mathtt{in}_F : F(\mu F) \to \mu F$ and $\mathtt{out}_F : \mu F \to F(\mu F)$ which are each others inverse. The data type μF is the least fixed point of F, the functor that captures the signature of its constructors. The functions $\mathtt{in}_F$ and $\mathtt{out}_F$ are used, respectively, to construct and destruct values of the data type μF. At least since the work of Meijer and Hutton presented in [17] it is well known how these concepts can be implemented directly in Haskell. First we define an explicit fixpoint operator with the keyword `newtype` to enforce the isomorphism.

```
newtype Mu f = In {out :: f (Mu f)}
```

For each new data type it is necessary to define the functor that captures its signature (and declare it as an instance of the class `Functor`), and then apply `Mu` in order to obtain the data type. For example, the `Tree` data type presented in the previous section could be defined as follows.

```
data FTree a b = Leaf | Node a b b

instance Functor (FTree a)
    where fmap f Leaf = Leaf
          fmap f (Node x l r) = Node x (f l) (f r)

type Tree a = Mu (FTree a)
```

A possible example of a tree with just two integer elements is

```
tree :: Tree Int
tree = In (Node 2 (In (Node 1 (In Leaf) (In Leaf))) (In Leaf))
```

Under some particular strictness conditions, the F-algebra $(\mu F, \mathtt{in}_F)$ is initial, and its carrier matches with the carrier of the final F-coalgebra $(\mu F, \mathtt{out}_F)$. This means that given any other F-algebra (B, g) or F-coalgebra (A, h), `fold` g and `unfold` h are the unique functions that make the following diagrams commute.

$$\begin{tikzcd}
\mu F \arrow[d, dotted, "\mathtt{fold}\, g"'] & F(\mu F) \arrow[l, "\mathtt{in}_F"'] \arrow[d, "F(\mathtt{fold}\, g)"] \\
B & FB \arrow[l, "g"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "h"] \arrow[d, dotted, "\mathtt{unfold}\, h"'] & FA \arrow[d, "F(\mathtt{unfold}\, h)"] \\
\mu F \arrow[r, "\mathtt{out}_F"'] & F(\mu F)
\end{tikzcd}$$

The unique definitions for folds and unfolds given by the diagrams can be directly translated to Haskell (in order to simplify the presentation we also give an explicit definition for hylomorphisms).

```
fold :: (Functor f) => (f b -> b) -> Mu f -> b
fold g = g . fmap (fold g) . out

unfold :: (Functor f) => (a -> f a) -> a -> Mu f
unfold h = In . fmap (unfold h) . h

hylo :: (Functor f) => (f b -> b) -> (a -> f a) -> a -> b
hylo g h = (fold g) . (unfold h)
```

These definitions can be seen as polytypic functions because they work for any data type, provided that it is modeled explicitly by the fixed point of a functor. The quick-sort example presented at the end of the previous section must be (slightly) adapted to this new methodology.

```
qsort :: (Ord a) => [a] -> [a]
qsort = hylo g h
    where h []     = Leaf
          h (x:xs) = Node x (filter (<=x) xs) (filter (>x) xs)
          g Leaf         = []
          g (Node x l r) = l ++ [x] ++ r
```

4 Generic Recursion Patterns

In order to use these recursion patterns, we still have to implement the map function for each functor. In order to avoid this task we will use Generic Haskell [3]. This language extends Haskell with polytypic features and originates on work by Ralf Hinze [8, 9], where generic functions are defined by induction on the structure of types, by providing equations for the base types and type constructors (like products and sums). Given this information, a generic definition can be specialized to any Haskell data type (these are internally converted into sums of products). The specialization proceeds inductively over the structure of the type, with type abstraction, type application, and type-level fixed-point being interpreted as their value-level counterparts.

Generic Haskell automatically converts each data type to an isomorphic type that captures its sum of products structure and records information about the presence of constructors. This structure type only represents the top-level structure of the original type. The types in the constructors do not change, including the recursive occurrences of the original type. The functions that convert data types into their structure types (and the other way round) are also automatically created. The constructors of structure types are

```
data a :+: b = Inl a | Inl b -- sum
data a :*: b = a :*: b       -- product
data Unit = Unit             -- unit
data Con a = Con a           -- constructor
```

For example, `Tree'` is the structure type of our first declaration of `Tree`.

```
data Tree a  = Leaf | Node a (Tree a) (Tree a)
type Tree' a = Con Unit :+: Con (a :*: (Tree a :*: Tree a))
```

Generic functions are then defined over the constructors of structure types and base types. The specific function for a data type is obtained by composing the specialization of the generic function to its structure type with the respective conversion functions. For example, a generic map function is predefined in the Generic Haskell libraries as follows[1].

[1] The delimiters `{|` `|}` enclose a type argument. In a generic function definition they enclose the type index for a given case. They are also used in the so-called generic application to require the specialization of a generic function to a specific type. When `Con` is supplied as a type index argument it includes an extra argument that

```
gmap {| Unit |} = id
gmap {| Int |}  = id
gmap {| :+: |} gmapA gmapB (Inl a)   = Inl (gmapA a)
gmap {| :+: |} gmapA gmapB (Inr b)   = Inr (gmapB b)
gmap {| :*: |} gmapA gmapB (a :*: b) = (gmapA a) :*: (gmapB b)
gmap {| Con c |} gmapA (Con a)       = Con (gmapA a)
```

Notice that different cases in `gmap` have different types (more precisely, the number of arguments equals the number of arguments of the type constructor). This is due to the fact that *polytypic values possess polykinded types* [9], a restriction that ensures that generic functions can be used with types of arbitrary kind. The type of `gmap` is specified in Generic Haskell as follows (the `type` keyword allows one to define a type by induction on the structure of its kind).

```
type Map {[ * ]} t1 t2 = t1 -> t2
type Map {[ k -> l ]} t1 t2 = forall u1 u2.
         Map {[ k ]} u1 u2 -> Map {[ l ]} (t1 u1) (t2 u2)

gmap {| t :: k |} :: Map {[ k ]} t t
```

For example, since the kind of `Tree` is `* -> *` (a constructor that receives a type as argument and produces a type), the type of the map function for binary trees can be determined by the following sequence of expansions.

```
Map {[ * -> * ]} Tree Tree
forall u1 u2 . Map {[ * ]} u1 u2 -> Map {[ * ]} (Tree u1) (Tree u2)
forall u1 u2 . (u1 -> u2) -> (Tree u1 -> Tree u2)
```

Since in Haskell all type variables are implicitly universally quantified, this type equals the expected one.

```
gmap {| Tree |} :: (a -> b) -> Tree a -> Tree b
```

In the case of the bifunctor `FTree`, of kind `* -> * -> *`, the map must receive two functions as arguments, one to be applied to the elements in the node, and another one for the recursive parameter.

```
gmap {| FTree |} :: (a -> b) -> (c -> d) -> FTree a c -> FTree b d
```

Besides generic functions, we can also define generic abstractions when a type variable (of fixed kind) can be abstracted from an expression. Typically this involves applying predefined generic functions to the type variable. For example, our recursion patterns can be defined by generic abstractions as follows [10].

is bound to a value providing information about the constructor name, its arity, etc. This information is necessary to implement, for example, a generic `show` function. Similarly, the delimiters `{[ ]}` used in type definitions enclose a kind argument.

```
fold {| f :: * -> * |} :: (f b -> b) -> Mu f -> b
fold {| f |} g = g . gmap {| f |} (fold {| f |} g) . out

unfold {| f :: * -> * |} :: (a -> f a) -> a -> Mu f
unfold {| f |} h = In . gmap {| f |} (unfold {| f |} h) . h

hylo {| f :: * -> * |} :: (f b -> b) -> (a -> f a) -> a -> b
hylo {| f |} g h = (fold {| f |} g) . (unfold {| f |} h)
```

Given these recursion patterns we no longer have to define the map functions for the data types we declare. For example, a quick-sort for integer lists can now be defined as follows (`g` and `h` are exactly the same as before).

```
qsort :: [Int] -> [Int]
qsort = hylo {| FTree Int |} g h
    where ...
```

However, there is a problem in defining the original polymorphic `qsort`. Given a list of type `[a]`, the intermediate data structure should be a binary tree with elements of type `a`, defined as the fixed point of `FTree a`. If Generic Haskell allowed the definition of scoped type variables (an extension to Haskell 98 described in [15]), one could define `qsort` as follows.

```
qsort :: (Ord a) => [a] -> [a]
qsort (l::[a]) = hylo {| FTree a |} g h l
    where ...
```

Unfortunately, since Generic Haskell does not support this extension we had to resort to a less elegant solution in order to allow for polymorphism. Instead of having a single definition for hylomorphisms, we define different functions to be used with monofunctors (`hylo1`), bifunctors (`hylo2`), and so on. As we have seen in the previous section for the `FTree` example, a polymorphic data type is obtained as the fixed point of a monofunctor, which in turn results from sectioning a bifunctor with the type variable. This means that the `Functor` instances should treat the type variable as a type constant (notice that in the case of `FTree` the contents of the node were left intact). When defining the generic recursion patterns for bifunctors this implies that we will have to pass the identity function as first argument to the generic map. For example, the fold for bifunctors is defined as follows.

```
fold2 {| f :: * -> * -> * |} :: (f c b -> b) -> Mu (f c) -> b
fold2 {| f |} g = g . gmap {| f |} id (fold2 {| f |} g) . out
```

The polymorphic `qsort` can then be defined as

```
qsort :: (Ord a) => [a] -> [a]
qsort = hylo2 {| FTree |} g h
    where ...
```

5 Deriving Hylomorphisms from Recursive Definitions

It is possible to derive automatically a hylomorphism from almost any explicit recursive definition of a function [12]. The main restrictions to the function definitions are that no mutual recursion is allowed, the recursive function calls should not be nested (thus excluding, for example, the usual definition of the Ackermann function), and, if the function has more than one argument, it should only induct over the last one (although it can be a tuple), leaving the remaining unchanged. This restrictions guarantee that the intermediate data type is a fixed point of a polynomial functor (sum of products), and, as a side-effect, that it models the recursion tree of the original definition.

We will informally explain how the algorithm works by applying it to the explicitly recursive definition of the quick-sort function. Our presentation assumes that all bounded variables are uniquely named. This restriction is trivially verified in this case.

```
qsort []     = []
qsort (x:xs) = qsort (filter (<=x) xs) ++ [x] ++
               qsort (filter (> x) xs)
```

The goal is to derive a functor `F`, and functions `g` and `h` in order to obtain the following hylomorphism (note that after determining `F` one should choose the appropriate `hylo` according to its kind).

```
qsort = hylo {| F |} g h
```

The first step is to identify, for the right hand side of each clause, the recursive calls (shown in italic) and all variables that occur free in those terms outside of the recursive calls (shown underlined).

```
qsort []     = []
qsort (x:xs) = qsort (filter (<=x) xs) ++ [x] ++
               qsort (filter (> x) xs)
```

The definition of `h` is similar to that of `qsort`, but the right hand sides are replaced by new data constructors applied to the free variables and the arguments of recursive calls.

```
h []     = F1
h (x:xs) = F2 x (filter (<=x) xs) (filter (>x) xs)
```

In order to define `g` we first replace the recursive calls in the right hand sides of `qsort` by fresh variables, and use as arguments the right hand sides of `h`, but with the arguments of recursive calls replaced by the new variables.

```
g F1           = []
g (F2 x r1 r2) = r1 ++ [x] ++ r2
```

The arguments of the functor `F` are all the free variables plus a single recursive variable `r`. The constructors have the same definition as the arguments of `g`, but with all the fresh variables replaced by `r`.

```
data F x r = F1 | F2 x r r
```

Putting it all together, and using the adequate `hylo` (in this case `F` is a bifunctor), we get the same definition as before.

```
qsort = hylo2 {| F |} g h
    where h []     = F1
          h (x:xs) = F2 x (filter (<=x) xs) (filter (>x) xs)
          g F1           = []
          g (F2 x r1 r2) = r1 ++ [x] ++ r2
```

As expected, the intermediate data type models the recursion tree of the original definition. In this example we got binary trees since quick-sort is a birecursive function.

We implemented this algorithm straightforwardly using some libraries for parsing and pretty-printing Haskell, combined with a state monad for managing unique names and other global information. Note that the generic definitions of the recursion patterns can be used for every new functor `F`.

6 Observing Recursion Trees

GHood [18] is a graphical animation tool built on top of Hood [7] (*Haskell Object Observation Debugger*). Hood is a portable debugger for full Haskell, based on the observation of intermediate data structures. Essentially, it introduces the following combinator with a similar signature to `trace` (a debugging primitive offered by all major Haskell distributions), but with a more complex behavior.

```
observe :: (Observable a) => String -> a -> a
```

This function just returns the second argument, but as a side-effect it stores it into some persistent structure for later rendering. It behaves like an identity function that can remember its argument. The string parameter is just a label that allows one to distinguish between different observations in the same program. The main advantage of `observe` over `trace` is that it can be effectively used without changing the strictness properties of the observed program.

Instances of `Observable` for the standard types are predefined. Implementing new instances of this class is very simple due to the high-level combinators and monads included in the library. As an example, we present the implementation for lists that is predefined in the Hood libraries.

```
instance (Observable a) => Observable [a] where
  observer (a:as) = send ":"  (return (:) << a << as)
  observer []     = send "[]" (return [])
```

The function `send` collects temporal information (when the observation was done) that is not used by Hood. GHood uses this information to produce animations. Its graphical visualization system is based on a simple layout algorithm.

Since the derivation of hylomorphisms exposes the recursion tree as an intermediate data structure, it is enough to place an observation point in the hylomorphism definition in order to visualize it.

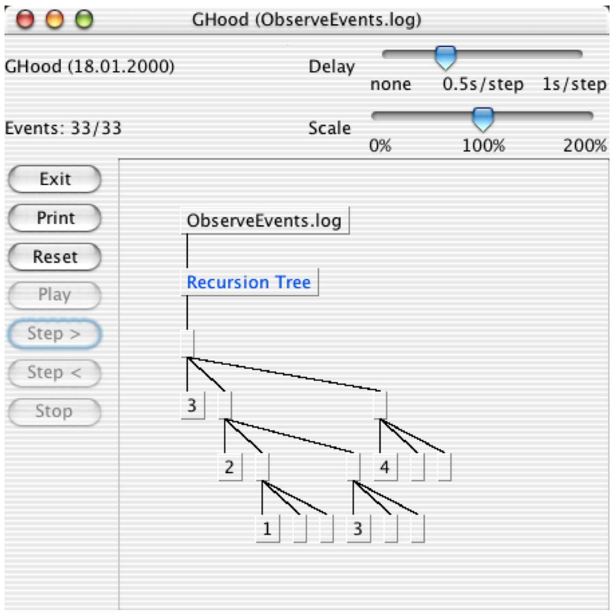


Fig. 1. Recursion tree of `qsort [3,2,4,3,1]`.

```
hylo {| f :: * -> * |} :: (f b -> b) -> (a -> f a) -> a -> b
hylo {| f |} g h = (fold   {| f |} g).(observe "Recursion Tree").
                   (unfold {| f |} h)
```

It is also necessary to implement instances of the class `Observable` for all possible intermediate data types. Since it is not known beforehand what kind of functor will be derived, it is necessary to provide a generic definition for `observer`. In the next section we provide that definition. For example, in the case of the functor `F` derived in the previous section, the final instances should behave as follows.

```
instance (Observable a) => Observable (Mu (F a)) where
    observer (In x) p = In (observer x p)

instance (Observable a, Observable b) => Observable (F a b) where
    observer F1          = send "" (return F1)
    observer (F2 x l r) = send "" (return F2 << x << l << r)
```

There are a couple of remarks to be made about these definitions. We bypass the observation of the fixpoint operator `Mu`. The goal is that the final animation should look the same as if the data type had been explicitly declared in a recursive fashion. The constructors of the data type are not displayed because their derived names are meaningless. Using these definitions, the recursion tree of `qsort [3,2,4,3,1]` is visualized as shown on figure 1.

```
type Observer {[ * ]} t = t -> ObserverM t
type Observer {[ k -> l ]} t = forall u . Observer {[ k ]} u ->
                                        Observer {[ l ]} (t u)

gobserverm {| t :: k |} :: Observer {[ k ]} t
gobserverm {| Unit |} = return
gobserverm {| Int |}  = return
gobserverm {| :+: |} oA oB (Inl a)   = do {a' <- (oA a); return (Inl a')}
gobserverm {| :+: |} oA oB (Inr b)   = do {b' <- (oB b); return (Inr b')}
gobserverm {| :*: |} oA oB (a :*: b) = do {a' <- (oA a); b' <- (oB b);
                                         return (a' :*: b')}
gobserverm {| Con c |} oA (Con a)    = do {a' <- (oA a); return (Con a')}
```

Fig. 2. Generic embedding into `ObserverM`

7 Generic Observations

In order to explain how we can define a generic observer we will first expand `<<` in the previous instance.

```
instance (Observable a, Observable b) => Observable (F a b) where
    observer F1         = send "" (return Leaf)
    observer (F2 x l r) = send "" (do {x' <- thunk x;
                                       l' <- thunk l;
                                       r' <- thunk r;
                                       return (Node x' l' r')}})
```

In this definition we can see that `send` runs a state monad (`ObserverM`) in order to evaluate a term and simultaneously collect information for the renderer, and `thunk` is invoked for each child in a node. Even without presenting more details, it is clear that the monadic code follows the structure of the type, and so it is possible to define generically a function to embed a value into `ObserverM`, as shown in figure 2.

Notice that if we parameterize this function with a monofunctor `f` we get a function with type `(a -> ObserverM a) -> f a -> ObserverM (f a)`. By passing `thunk` as parameter we can get a generic definition of `observer` for monofunctors that behaves as expected.

```
gobserver {| f :: *->* |} :: Observable a => f a -> Parent -> f a
gobserver {| f |} x p = send "" (gobserverm {| f |} thunk x) p
```

Technically speaking, `gobserverm` is a monadic map for `ObserverM`. Given a functor F, and a monad M, the monadic map should transform functions of type $A \to M\ B$ into functions of type $F\ A \to M\ (F\ B)$. This concept was introduced by Fokkinga in [5]. For example, in the standard `Prelude` of Haskell, the function `mapM` implements the monadic map for lists. However, likewise to the regular map function, given a polynomial functor the monadic map can be defined by induction on its structure, thus being suitable for polytypic implementation. In fact, Generic Haskell provides a library `MapM` where this function is implemented with two different versions: one that evaluates the products left-to-right (`mapMl`) and another that evaluates them right-to-left (`mapMr`). If the monad is strong and commutative both yield the same result, but in general that is not the case. If we abstract the monad `ObserverM` in `gobserverm` we obtain the function `mapMl`. Given this equivalence, we can implement the generic observer as follows.

```
gobserver {| f :: *->* |} :: Observable a => f a -> Parent -> f a
gobserver {| f |} x p = send "" (mapMl {| f |} thunk x) p
```

As was the case for recursion patterns, we have to define different generic observers for monofunctors (`gobserver1`), bifunctors (`gobserver2`), etc. The instance implementation for functor `F` presented in the previous section can now be simply obtained as follows.

```
instance (Observable a, Observable b) => Observable (F a b) where
    observer = gobserver2 {| F |}
```

In the presence of an extension to the type system allowing for the specification of polymorphic predicates in an instance declaration, as presented in [11], we could also have a single definition of the `Observable` instance for `Mu`. That instance would look like

```
instance (forall b . (Observable b) => Observable (f b))
                                    => Observable (Mu f) where
    observer (In x) p = In (observer x p)
```

8 Conclusions and Future Work

The techniques presented in this paper were included in an application that, given a Haskell module, tries to derive hylomorphisms for all the functions declared in that module. When successful, each original definition is replaced by the corresponding new one, and the data type that models the recursion tree is declared, together with the appropriate instances of `Observable`. The resulting module is written in Generic Haskell and should be compiled into regular Haskell. The execution of each transformed function triggers, as a side-effect, a visualization of its recursion tree.

The main contribution of this paper is a new approach to visualizing recursion trees, that makes intensive use of generic programming in order to make the task of putting together previously developed tools and techniques easier. Other specific contributions are the clarification of the use of generic recursion patterns in polymorphic definitions, and the generic definition of observations to be used with `GHood`. In the past [4], we developed a preliminary solution to this problem, with a proprietary observation mechanism based on monadic recursion patterns. However, it had several problems, namely, it changed the strictness properties of the original definitions and it had no support for polymorphism.

Essentially, we have used generic programming to overcome the limitations of the Haskell `deriving` mechanism, which currently supports only a limited range of classes. There has been some research towards developing specific mechanisms to overcome these limitations. These mechanisms could have been used to achieve a similar effect. Hinze and Peyton Jones proposed *Derivable Type Classes* [11], a system based on the same theoretical concepts as Generic Haskell, but that only allows generic definitions in instance declarations. Unfortunately, it is not yet fully implemented in any Haskell distribution. An older system is DrIFT [19], a preprocessor that parses a Haskell module for special commands that trigger the generation of new code. This is a rather *ad hoc* mechanism, which is not as theoretically sound as generic programming.

In the future we intend to develop a new animation backend to replace GHood, in order to increase the understanding of the functions being observed. First, it should allow us to visualize the consumption of the intermediate data structure, showing how the final result is obtained from the recursion tree. It should also allow more control on the view of nested hylomorphisms. Sometimes, the parameters of a hylomorphism are also hylomorphisms, and to simplify the presentation these should only be visualized as requested.

Acknowledgments

I would like to thank José Barros, Jorge Sousa Pinto, José Bacelar Almeida, Alexandra Barros, and all the anonymous referees for their helpful comments on previous versions of this paper. Thanks also to João Saraiva for calling up the idea of hylomorphism visualization, and Andres Löh for his help with Generic Haskell.

References

1. Lex Augusteijn. Sorting morphisms. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer Verlag, 1999.
2. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming – an introduction. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer Verlag, 1999.
3. Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.
4. Alcino Cunha, José Barros, and João Saraiva. Deriving animations from recursive definitions. In *Draft Proceedings of the 14th International Workshop on the Implementation of Functional Languages (IFL'02)*, 2002.
5. Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94–28, University of Twente, June 1994.
6. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 273–279. ACM Press, 1998.

7. Andy Gill. Debugging Haskell by observing intermediate data structures. In G. Hutton, editor, *Proceedings of the 4th ACM SIGPLAN Haskell Workshop*, 2000.
8. Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 119–132. ACM Press, 2000.
9. Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction (proceedings of MPC'00)*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.
10. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In Eerke Boiten and Bernhard Möller, editors, *Mathematics of Program Construction (proceedings of MPC'02)*, volume 2386 of *LNCS*, pages 148–174. Springer-Verlag, 2002.
11. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *ENTCS*. Elsevier, 2001.
12. Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.
13. Patrik Jansson and Johan Jeuring. Polyp – a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
14. Simon Peyton Jones and John Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. February 1999.
15. Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. To be submitted to The Journal of Functional Programming, March 2002.
16. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.
17. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, 1995.
18. Claus Reinke. GHood - graphical visualisation and animation of Haskell object observations. In Ralf Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, volume 59 of *ENTCS*. Elsevier, 2001.
19. Noel Winstanley and John Meacham. *DrIFT User Guide (version 2.0rc3)*, 2002.

Cost-Sensitive Debugging of Declarative Programs [⋆]

D. Ballis[1] M. Falaschi[1] C. Ferri[2]
J. Hernández-Orallo[2] M.J. Ramírez-Quintana[2]

[1] DIMI, University of Udine, Via dell Scienze 206, 33100 Udine, Italy.
{ballis,falaschi}@dimi.uniud.it
[2] DSIC, Univ. Politècnica de València, Camí de Vera s/n, 46022 València, Spain.
{cferri,jorallo,mramirez}@dsic.upv.es

Abstract. Diagnosis methods in debugging aim at detecting bugs of a program, either by comparing it with a correct specification or by the help of an oracle (usually the user herself). Debugging techniques for declarative programs usually exploit the semantical properties of programs (and specifications) and generally try to detect one or more "buggy" rules. In this way, rules are split apart in an absolute way: either they are correct or not. However, in many situations, not every error has the same consequences, an issue that is ignored by classical debugging frameworks. In this paper, we generalise debugging by considering a cost function, i.e. a function that assigns different cost values to each kind of error and different benefit values to each kind of correct response. The problem is now redefined as assigning a real-valued probability and cost to each rule, by considering each rule more or less "guilty" of the overall error and cost of the program. This makes possible to *rank* rules rather than only separate them between right and wrong. Our debugging method is also different from classical approaches in that it is probabilistic, i.e. we use a set of ground examples to approximate these rankings.

Keywords: Declarative Debugging, Cost Matrices, Software Testing, Cost-sensitive Evaluation.

1 Introduction

Debugging has always been an undesired stage in software development. Much research in the area of software development has been devoted to avoid errors but, in the general framework of programming, the errors are still there and, hence, must be detected and corrected.

The problem of debugging can be presented in several ways. In the worst case, we only know that the program does not work correctly, and little can be done automatically. Sometimes we are given several wrong (and optionally right) computations. In other situations, we take for granted that we have an

[⋆] This work has been partially supported by CICYT under grant TIC2001-2705-C03-01 and Acción Integrada Hispano-Italiana HI2000-0161.

oracle (generally the user) whom we can ask about the correct computation of any example. In the best case, we have a formal specification and the program to be corrected.

Even in the best case (we know the specification) the diagnosis of which parts of the program are wrong is not trivial. Successful techniques in the diagnosis of bugs are usually associated with the type of programming language used. Programming languages based on rules and, especially, declarative languages allow the use of powerful (formal) techniques to address the problem. Several approaches have been developed for logic programming [4], functional programming [9] and logic and functional integrated languages [1, 2], in order to debug programs according to different observable properties, e.g., correct answers, computed answers, call patterns, finite failure, etc. Furthermore, some of these methods are also able to diagnose lost answers (the completeness problem).

However, many formal techniques derived so far are based on the idea of comparing specification and program, also assuming that all the predicates or functions implemented by the program are also defined by the specification, to which they can be compared, or assuming that auxiliary functions are correct.

So we would like to cope with large programs, where the specification of the auxiliary functions is not known, and taking into account the program usage distribution and the costs of each kind of error.

Let us illustrate with an example the kind of problems we want to address.

Example 1. Consider a taxi company that wants to use a program for sending the appropriate vehicle type (car, van, minibus or special service) depending on the number of people in the group that has made the call. The maximum capacities of each vehicle are 4 for cars, 8 for vans and 12 for minibusses. The specification I given by the company is then as follows[1]:

$\texttt{taxi(N)} \rightarrow \texttt{car} \Leftarrow \texttt{N} \leq \texttt{4}$
$\texttt{taxi(N)} \rightarrow \texttt{van} \Leftarrow \texttt{4} < \texttt{N} \leq \texttt{8}$
$\texttt{taxi(N)} \rightarrow \texttt{minibus} \Leftarrow \texttt{8} < \texttt{N} \leq \texttt{12}$
$\texttt{taxi(N)} \rightarrow \texttt{special} \Leftarrow \texttt{12} < \texttt{N}$

A programmer just implements this problem as the following program R_1:

$r1 : \texttt{cap(car)} \rightarrow \texttt{4}$
$r2 : \texttt{cap(van)} \rightarrow \texttt{9}$
$r3 : \texttt{cap(minibus)} \rightarrow \texttt{12}$
$r4 : \texttt{interval(X,N,Z)} \rightarrow (\texttt{X} < \texttt{N} \leq \texttt{Z})$
$r5 : \texttt{taxi(N)} \rightarrow \texttt{car} \Leftarrow \texttt{interval(0,N,cap(car))}$
$r6 : \texttt{taxi(N)} \rightarrow \texttt{van} \Leftarrow \texttt{interval(cap(car),N,cap(van))}$
$r7 : \texttt{taxi(N)} \rightarrow \texttt{minibus} \Leftarrow \texttt{interval(cap(van),N,cap(minibus))}$
$r8 : \texttt{taxi(N)} \rightarrow \texttt{special} \Leftarrow \texttt{cap(minibus)} < \texttt{N}$

As can be seen, the programmer has made a mistake on the capacity of the van. However, the semantics of the auxiliary function cap measuring the"capacity" of a vehicle is not in the specification and, consequently, classical approaches cannot be applied unless assuming that the auxiliary functions are correct, which in this case it is not true.

[1] In the example, $<$ and $\leq$ are the usual predefined boolean functions modelling inequality relations among naturals.

Nonetheless, the previous example can also be used to show that even detecting errors is not sufficient for improving the quality of a program.

Example 2. Consider the alternative program R_2:

$r1 : \texttt{cap(car)} \rightarrow \texttt{3}$
$r2 : \texttt{cap(van)} \rightarrow \texttt{7}$
$r3 : \texttt{cap(minibus)} \rightarrow \texttt{12}$
$r4 : \texttt{interval(X,N,Z)} \rightarrow (\texttt{X} < \texttt{N} \leq \texttt{Z})$
$r5 : \texttt{taxi(N)} \rightarrow \texttt{car} \Leftarrow \texttt{interval(0,N,cap(car))}$
$r6 : \texttt{taxi(N)} \rightarrow \texttt{van} \Leftarrow \texttt{interval(cap(car),N,cap(van))}$
$r7 : \texttt{taxi(N)} \rightarrow \texttt{minibus} \Leftarrow \texttt{interval(cap(van),N,cap(minibus))}$
$r8 : \texttt{taxi(N)} \rightarrow \texttt{special} \Leftarrow \texttt{N} > \texttt{cap(minibus)}$

Apparently, this program is more buggy than R_1 since rules $r1$ and $r2$ seem to be wrong. However, in the context where this program is going to be used, it is likely that R_2 is better than R_1. But why?

The issue here is that in most situations, not every error has the same consequences. For instance, a wrong medical diagnosis or treatment can have different costs and dangers depending on which kind of mistake has been done. Obviously, costs of each error are problem dependent, but it is not common the case that they are uniform for a single problem. Note that it might even be cheaper not to correct a limited bug and invest that time in correcting other costlier bugs.

For the previous example, the cost of each error could be represented as a "cost matrix" and its values could have been estimated by the company[2]:

		I			
		car	*van*	*minibus*	*special*
	car	0	200	200	200
R	*van*	4	0	200	200
	minibus	10	6	0	200
	special	30	15	5	0

Obviously the cost of sending a car when a van is needed (200) is much higher than the contrary situation (4), since in the first case the vehicle is not big enough to carry all the people. For this particular matrix and a non-biased distribution of examples, the program R_2 is better than the program R_1, even though apparently it has more buggy rules. Nonetheless a different distribution of examples (e.g. if groups of 4 people are much more common than groups of 9 people) could turn R_1 into a better program. Consequently, in order to know which rules are more problematic or costlier, we need to know:

- The cost of each kind of error given by a cost function.
- The distribution of cases that the program has to work with.

The previous example also shows that if we do not have the specification of the main function and all the auxiliary functions, it is impossible in general to

[2] This is usually a simplification, since the cost may also depend on the size, the age or even the extra number of people.

talk about correct and incorrect rules. In these cases, it is only sensible to talk about rules that participate more or less in the program errors or costs, giving a rate rather than a sharp classification between good and bad rules. Another common situation is during the development of the software product when there is not enough time to correct all the errors before the date of the purchase of the product. In this case, it could be very useful to have a rank of errors in order to correct the most important ones first.

Taking into account the previous considerations, we are going to devise a new debugging schema (for terminating programs) with the following requirements:

– It should be able to work with samples of examples, with oracles or with a correct specification, without the need of correct specifications for auxiliary functions.
– It should consider a user-defined or predefined cost function in order to perform cost-sensitive debugging.
– The previously obtained costs will be used to rank the rules from more costly to less costly rather than strictly separating between buggy or non-buggy rules.
– It should be possible to use a tunable trade-off between efficiency of the method and the quality of its ranking.

The paper is organised as follows. In Section 2, we give some notation and we formally introduce the cost-sensitive diagnosis problem. Section 3 deals with an estimation of the rule error probability. Section 4 is mainly devoted to the illustration of the cost-sensitive diagnosis framework. Scalability and applicability of the approach are discussed in Section 5. Section 6 concludes the paper.

2 Notation and Problem Statement

In this work, we will concentrate on declarative languages, although most of the ideas introduced are valid for other rule-based languages. First, let us briefly introduce some notation and next we will state the problem more formally.

2.1 Notation

Throughout this paper, V will denote a countably infinite set of variables and Σ is a *signature* denoting a set of function symbols, each of which has a fixed associated arity. $\mathcal{T}(\Sigma \cup V)$ and $\mathcal{T}(\Sigma)$ denote the non-ground term algebra and the term algebra built on $\Sigma \cup V$ and Σ, respectively. By $Var(t)$ we intend the set of variables occurring in term t. Let $\Sigma_\perp = \Sigma \bigcup \{\perp\}$, whereas $\perp$ is a new fresh constant. Given an equation e, we will denote the left-hand side (resp. the right-hand side) of e by $lhs(e)$ (resp. by $rhs(e)$).

A *conditional term rewriting system* (CTRS for short) is a pair (Σ, R), where R is a finite set of rewrite rule schemes of the form $(\lambda \to \rho \Leftarrow C)$, $\lambda, \rho \in \mathcal{T}(\Sigma \cup V)$, $\lambda \notin V$ and $Var(\rho) \subseteq Var(\lambda)$. The condition C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations.

We will often write just R instead of (Σ, R). Given a CTRS (Σ, R), we assume that the signature Σ is partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{F}$, where $\mathcal{F} = \{f \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in R\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{F}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{F}$ are called *defined functions*. Elements in $\mathcal{T}(\mathcal{C})$ are called *values* (or ground constructor terms). In the remainder of this paper a *program* is a CTRS. A term t is a *normal form* w.r.t. a program R, if there is no term s such that $t \to_R s$, where $\to_R$ denotes the usual conditional rewrite relation [7]. Let $nf_R(t)$ denote the set of normal forms of the term t w.r.t. R and $cnf_R(t) = \{s|s \in nf_R(t)$ and $s \in \mathcal{T}(\mathcal{C})\}$. Moreover, let $nf_{R,r}(t) = \{s|s \in nf_R(t)$ and r occurs in $t \to_R^* s\}$ and $cnf_{R,r}(t) = \{s|s \in nf_{R,r}(t)$ and $s \in \mathcal{T}(\mathcal{C})\}$. A CTRS R is *terminating* if there are no infinite sequences of the form $t_1 \to_R t_2 \to_R t_3 \to_R \ldots$ A CTRS R is *confluent* if, for each term s such that $s \to_R^* t_1$ and $s \to_R^* t_2$, there exists a term u such that $t_1 \to_R^* u$ and $t_2 \to_R^* u$. If R is confluent then the normal form of a term t is unique. A *sample* S is a set of examples of the form $\langle e, n\rangle$ where e is a ground equation and $n \in \mathbb{N}$ is a unique identifier. For the sake of simplicity, we will often denote the example $\langle e, n\rangle$ by e_n or e.

2.2 A Cost-Sensitive Debugging Schema

A cost-sensitive debugging problem is defined as follows:

- a program R to be debugged;
- one main function definition f, selected from R;
- a correctness reference, either an intensional specification I or a sample S, or an oracle O;
- a cost function defined from $\mathcal{T}(\mathcal{C}_\perp) \times \mathcal{T}(\mathcal{C})$ to the set of real numbers;
- a probability distribution function p of ground terms that have f as outermost function symbol.

Let us define more precisely some of the previous components. First of all, the program R is assumed to be terminating in order to ensure effectiveness of our debugging framework. However, R could be non-confluent. In that case, we assume the following worst-case error situation: if one of all the normal forms computed by R is not correct w.r.t. the correctness reference, we consider program R incorrect.

Secondly, f is the main function definition to which the cost measures and errors refer to. In the case that more than one function definition f must be taken into account, we could debug them separately.

Thirdly, our cost-sensitive diagnosis framework exploits a correctness reference representation based on a sample S, which could be generated according to distribution p. Example generation could be done by means of a generative grammar method, which is able to automatically yield a set of ground equations satisfying a given probability distribution requirement, as we will discuss in section 5. However, correctness reference can also be represented by the following alternative forms: an intensional program I (a specification, which at least defines function f), an oracle O (an entity that can be queried about the correct normal form of any well-constructed term whose outermost symbol is f). Our framework can deal with all these three representations, since both I and O can be used to randomly generate a sample S, according to a given distribution p. Hereinfater, we will work with S.

Fourthly, we have a function $ECF_f : \mathcal{T}(\mathcal{C}_\perp) \times \mathcal{T}(\mathcal{C}) \longrightarrow I\!R$, called the Error Cost Function, such that the first argument of this function is intended to be one of the normal forms computed by the program R for a term $f(t_1, \ldots, t_n)$ and the second one is intended to be the correct normal form. In the particular case that the set of computed values is a set of constants (nominal type), the Error Cost Function can be expressed as a bidimensional matrix, which makes its understanding easier, as seen in the introduction.

As we will see, in many cases ECF_f and p are not known. We will also discuss reasonable assumptions (symmetric cost function and universal distribution) for these situations later.

2.3 Kinds of Error

First of all, we assume that our sample S fullfils the following properties: (i) for each $l = r \in S$, $r \in \mathcal{T}(\mathcal{C})$, (ii) there are not two equations e and e' such that $lhs(e) = lhs(e')$ and $rhs(e) \neq rhs(e')$. The reason why we require (i) concerns the fact that generally programmers are interested in computing values. So, we will rank rules w.r.t. values. Besides, (ii) is needed in order to force a good behaviour of our correctness reference: we do not want that a ground function call can calculate more than one value. In particular, (ii) implies a *confluent* intensional specification I.

For this work we will consider two kinds of errors: correctness errors and completeness errors.

Definition 1 (Correctness Error). *The program R has a* correctness error *on example e, denoted by $\Box_e$, when there exists a term $t \in cnf_R(lhs(e))$ such that $t \neq rhs(e)$.*

Definition 2 (Completeness Error). *The program R has a* completeness error *on example e, denoted by Δ_e, when there exists a term $t \in nf_R(lhs(e))$ such that $t \notin \mathcal{T}(\mathcal{C})$.*

Note that there may be cases when there is both a correctness error and a completeness error. In general we will use the symbols $\Box$ and Δ when we do not refer to any particular example.

For simplicity, the error function ECF_f is defined only for values (ground constructor terms) plus the additional element $\perp$ representing all the non-constructor normal forms. More precisely, the first argument of ECF_f has to be a value or $\perp$, while the second argument is constrained to be a value.

3 Estimating the Error Probability

Consider a program R that is not correct w.r.t. some specification. In some cases there is clearly one rule that can be modified to give a correct program. In other cases, a program cannot be corrected by changing only one rule and the possibilities of correction (and hence the number of rules that might be altered) become huge.

Consequently, we are interested in a rating of rules that assigns higher values to rules that are more likely to be guilty of program failure. More precisely, we want to derive the probability that given an error, the cause is a particular rule. If we enumerate the rules of a program from r_1 to r_n and, as we defined in Section 2.3, we denote by $\Box$ a mismatch between the normal form computed by R and the correct normal form (correctness error), then we would like to obtain the probability that given a correctness error, a particular rule is involved, i.e. $P(r_i|\Box)$.

This probability could be computed directly or, as will be shown later, it could be obtained by the help of Bayes theorem, i.e.:

$$P(r_i|\Box) = \frac{P(r_i) \cdot P(\Box|r_i)}{P(\Box)}.$$

So, we only have to obtain the following probabilities: $P(r_i)$, $P(\Box|r_i)$ and $P(\Box)$. In the sequel, let $card(L)$ be the cardinality of a sample L. The probability of error $P(\Box)$ is easy to be estimated by using the sample S. Let us denote by $E \subseteq S$ the sample which gives rise to a correctness error using R. Formally, $E = \{e \in S \,|\, \exists\, t \in \mathit{cnf}_R(\mathit{lhs}(e)) \text{ and } t \neq rhs(e)\}$. As we said, if R is not confluent it is sufficient that one normal form is different from the correct value to consider that to be an error. Consequently:

$$P(\Box) \approx \frac{card(E)}{card(S)}$$

i.e., the number of cases that are wrongly covered by the program w.r.t. the size of the sample.

The probability that a rule is used, $P(r_i)$, is also easy to be estimated. Let us denote by $U_i \subseteq S$ the following sample: $U_i = \{e \in S \,|\, \exists\, t \in \mathit{cnf}_{R,r_i}(\mathit{lhs}(e))\}$. If an example uses more than once rule r_i, r_i is only reckoned once. Then we have:

$$P(r_i) \approx \frac{card(U_i)}{card(S)}$$

Finally, we have to approximate the probability that there is an error whenever a rule r_i is used, that is, $P(\Box|r_i)$. Let us denote by $E_i \subseteq U_i$ the sample from U_i rising a correctness error w.r.t. R. Then,

$$P(\Box|r_i) \approx \frac{card(E_i)}{card(U_i)}.$$

If we take a look at the previous formulae, it is straightforward noting that:

$$P(r_i|\Box) \approx \frac{card(E_i)}{card(E)}$$

which is consistent with the definition of $P(r_i|\Box)$ as "probability that given an error, the cause is a particular rule". Moreover, since $card(E)$ is the same for all the rules, we would have that $P(r_i|\Box)$ only depends on $card(E_i)$.

However, consider the following example:

r_1 : $\texttt{f(X, [])} \rightarrow \texttt{0}$
r_2 : $\texttt{f(true, [A|B])} \rightarrow \texttt{sumlist([A|B])}$
r_3 : $\texttt{f(false, [A|B])} \rightarrow \texttt{sizelist([A|B])}$
r_4 : $\texttt{sumlist([A|B])} \rightarrow \texttt{A} + \texttt{sumlist[B]}$
r_5 : $\texttt{sumlist([])} \rightarrow \texttt{0}$
r_6 : $\texttt{sizelist([A|B])} \rightarrow \texttt{0} + \texttt{sizelist[B]}$
r_7 : $\texttt{sizelist([])} \rightarrow \texttt{0}$

and $S = \{\langle f(true, [3, 2, 5]) = 10, 1\rangle, \langle f(false, []) = 0, 2\rangle, \langle f(false, [2]) = 1, 3\rangle\}$. With this we would have that $P(r_6|\Box) = 1$. This is so easily detected because f is implemented in a "divide-and-conquer" fashion, and examples are distributed in such a way that even auxiliary functions not given in the specification can be debugged. However, this may not be the case for programs where the recursive clauses are deeply interrelated (we would also need the definition of the auxiliary functions in these cases).

When there are few examples, it could happen that a faulty rule never rises a correctness error, because it might not appear in any example rewrite sequence. Therefore it may not be detected as incorrect. In this situation, the use of some smoothing techniques, such as *Laplace smoothing* or the *m-estimate* [3], for correcting error rule probabilities could be helpful.

3.1 Advantages and Caveats of using $P(r_i|\Box)$

Let us study the kinds of problems where the previous approach works. In some cases the rating is clearly intuitive, for instance, when we have non-recursive function definitions with a fine granularity of the rules or recursive definitions where the error is in the base case. Consider, e.g., the following wrong program for the function *even*:

r_1 : $\texttt{even(0)} \rightarrow \texttt{false}$
r_2 : $\texttt{even(s(s(X)))} \rightarrow \texttt{even(X)}$

Here r_1 is clearly given the highest $P(r_i|\Box)$ whenever the sample contains "even(0) = true".

However, there are some cases where there is a tie between the error probabilities of two or more rules. One can imagine to use $P(\Box|r_i)$ in order to untie, but this does not always work properly. For instance, let us consider the following example that computes the product of two natural numbers by the use of an auxiliary function *multaux*:

$r1$: $\texttt{mult(0, y)} \rightarrow \texttt{0}$
$r2$: $\texttt{mult(s(X), Y)} \rightarrow \texttt{s(multaux(X, Y, Y))}$
$r3$: $\texttt{multaux(X, Y, 0)} \rightarrow \texttt{mult(X, Y)}$
$r4$: $\texttt{multaux(X, Y, s(Z))} \rightarrow \texttt{s(multaux(X, Y, Z))}$

In this case, if the sample S just contains examples for *mult* but not for *multaux* (probably because we have a specification of *mult* using *add*, as usual) then we will have $P(\square|r_2) = P(\square|r_3) = P(\square|r_4) = 1$. However, $P(r_2|\square) = P(r_3|\square) = 1$. We cannot in general distinguish between r_2 and r_3, because whenever r_2 is used r_3 is also used and viceversa. If we had examples for *multaux*, we would have that $P(\square|r_2) = 1 > P(\square|r_3) = P(\square|r_4)$, clarifying that since *multaux* seems to be correct, the error can only be found in r_2.

The following result about incorrectness detection can be proven.

Theorem 1. *Let S be a sample and R be a program not rising correctness error w.r.t. S. Consider a program R' which differs from R just by one rule r_i. Then, $\exists\, e' \in S$ such that $cnf_R(lhs(e')) \subset cnf_{R'}(lhs(e'))$. Then, the diagnosis method assigns the greatest $P(r_i|\square)$ to rule r_i.*

Proof. Since there is (at least) $e' \in S$ such that $cnf_R(lhs(e')) \subset cnf_{R'}(lhs(e'))$, we have that there exits (at least) a term $t \in cnf_{R'}(lhs(e'))$ and $t \notin cnf_R(lhs(e'))$. Since R is correct w.r.t. S, then $rhs(e') \in cnf_R(lhs(e'))$ and $t \neq rhs(e')$. Thus, R' rises a correctness error on example e'. Clearly, there may be more than one $e' \in S$ on which R' gives rise to a correctness error. So, let $E \subseteq S$ be the sample of such examples. Since R' only differs from R by one rule r_i and R does not give rise to any correctness error, we have that $E_i = E$. Hence,

$$P(r_i|\square) \;=\; \frac{card(E_i)}{card(E)} \;=\; 1.$$

Consequently, any other rule of R' must have less or equal probability. And this proves the claim. $\square$

Finally, the previous measure detects faulty rules and it is able to *rank* them. However, even though the distribution of examples is taken into account (the probabilities are estimated from this distribution), sometimes we obtain rule rankings containing ties. Therefore we are sometimes not allowed to localise error sources with a sufficient precision degree. In the next section, we will try to refine the ranking process by considering rule costs.

4 A Cost-Sensitive Diagnosis

In this section, we define a cost-sensitive diagnosis for functional logic programs. For this purpose we require a cost function $ECF_f : \mathcal{T}(\mathcal{C}_\perp) \times \mathcal{T}(\mathcal{C}) \longrightarrow I\!R$ defined for the outputs of function f.

By using this function we could compute the cost assigned to each rule, i.e. which proportion of the overall cost of the program is blamed to each rule.

Definition 3 (Rule Correctness Cost). *The* rule correctness cost*, denoted by $Cost_{r_i}$ is computed in the following way:*

$$Cost_{r_i} = \frac{\sum_{e \in S} Cost_{r_i}(e)}{card(S)}$$

where $Cost_{r_i}(e)$ is defined as follows:

$$Cost_{r_i}(e) = \sum_{t \in cnf_{R,r_i}(lhs(e))} ECF_f(t, rhs(e))$$

Example 3. Consider the following example:

r_1 : `even(0)` $\rightarrow$ `true`
r_2 : `even(s(X))` $\rightarrow$ `even(X)`

and $S = \{\langle even(0) = true, 1\rangle, \langle even(s(0)) = false, 2\rangle, \langle even(s(s(0))) = true, 3\rangle, \langle even(s(s(s(0)))) = false, 4\rangle\}$. And consider the ECF_{even} defined by the following cost matrix.

		I	
		$false$	$true$
R	$false$	-1	1
	$true$	1	-1

Consequently, using this cost matrix, we have that r_2 is costlier than r_1.

However, there are two important questions to be answered here. What is the relation between the *Cost* measure and the error probability? And secondly, which cost matrix should be used in the case we are not given one? The first question is answered by the following proposition, which states that the rule correctness cost method is a "generalisation" of error probability method.

Proposition 1. *Let R be a confluent CTRS, r_i be a rule belonging to R and S be a sample. Consider the cost function $ECF_f(X, Y) = \kappa$ if $X \neq Y$ and 0 otherwise, where $\kappa \in \mathcal{R}, \kappa > 0$. Then, $Cost_{r_i} = \kappa \cdot P(r_i|\Box) \cdot P(\Box)$.*

Proof. Let $e \in S$. Consider the rule correctness cost per example $Cost_{r_i}(e)$ which is defined as

$$Cost_{r_i}(e) = \sum_{t \in cnf_{R,r_i}(lhs(e))} ECF_f(t, rhs(e)). \tag{1}$$

Since R is confluent, the normal form of a given term (whenever it exists) is unique. So, Equation 1 becomes

$$Cost_{r_i}(e) = ECF_f(t, rhs(e)) = \begin{cases} \kappa \text{ if } t \neq rhs(e), t \in \mathcal{T}(\mathcal{C}), \kappa \in \mathbb{R}, \kappa > 0 \\ 0 \text{ otherwise.} \end{cases} \tag{2}$$

where t is the unique normal form of $lhs(e)$. By summing each rule correctness cost per example, we get the following relation:

$$card(E_i) = \frac{1}{\kappa} \sum_{e \in S} Cost_{r_i}(e) \tag{3}$$

As $E \subseteq S$, $card(S) = card(E) + c$, where $c \geq 0$ is a natural constant. By exploiting relations above and probability definitions given in Section 3, we finally get the desired result.

$$Cost_{r_i} = \frac{\sum_{e \in S} Cost_{r_i}(e)}{card(S)} = \kappa \cdot \frac{card(E_i)}{card(S)} = \kappa \cdot \frac{card(E_i)}{card(E)} \cdot \frac{card(E)}{card(S)} = \kappa \cdot P(r_i|\Box) \cdot P(\Box).$$

So, our claim is proven. $\Box$

The interesting thing (and the answer to the second question) comes when we use cost functions which do not subsume error probabilities.

As we have seen in the previous example, instead of the cost function that does not take into account hits (i.e. correct constructor normal forms), we prefer the symmetrical cost function $ECF_f(X,Y) = 1$ if $X \neq Y$ and -1 otherwise (we assign 0 when X is not a value). The rationale of this choice is based on the idea that not only errors have to be taken into account but also hits. Hence, a very useful part of a program may have participated in an error but should be less blamed for that than a less useful part of a program that has also participated in the error.

According to all the machinery we set up, a by-default modus operandi could be as follows.

Modus Operandi.

- First of all, discard all the incompleteness errors. The true equations corresponding to these errors (lhs is the normal form generated by R and rhs is the correct normal form) are output on a new set E_Δ.
- Compute the $p(r_i|\Box)$ for all the rules and give a first ranking of rules.
- In case of tie use $Cost_{r_i}$ with the symmetrical cost function to refine the ranking.

Let us see in the taxi assignment example of Section 1, whether this rule of thumb works in practice.

Example 4. Consider the following program.

r_1 : $\texttt{cap(car)} \to \texttt{4}$
r_2 : $\texttt{cap(van)} \to \texttt{9}$
r_3 : $\texttt{cap(minibus)} \to \texttt{12}$
r_4 : $\texttt{interval(X,N,Z)} \to (\texttt{X} < \texttt{N} \leq \texttt{Z})$
r_5 : $\texttt{taxi(N)} \to \texttt{car} \ \Leftarrow \ \texttt{interval(0,N,cap(car))}$
r_6 : $\texttt{taxi(N)} \to \texttt{van} \ \Leftarrow \ \texttt{interval(cap(car),N,cap(van))}$
r_7 : $\texttt{taxi(N)} \to \texttt{minibus} \ \Leftarrow \ \texttt{interval(cap(van),N,cap(minibus))}$
r_8 : $\texttt{taxi(N)} \to \texttt{special} \ \Leftarrow \ \texttt{cap(minibus)} < \texttt{N}$

and the specification I shown in the introduction. Let us use the following sample: $S = \{\langle taxi(1) = car, 1\rangle$, $\langle taxi(1) = car, 2\rangle$, $\langle taxi(2) = car, 3\rangle$, $\langle taxi(3) = car, 4\rangle$, $\langle taxi(5) = van, 5\rangle$, $\langle taxi(7) = van, 6\rangle$, $\langle taxi(9) = minibus, 7\rangle$, $\langle taxi(11) = minibus, 8\rangle$, $\langle taxi(20) = special, 9\rangle\}$. Note that repeated examples exist and they should be taken into account. By applying the probability estimation of Section 3 and the symmetrical cost function, the following probabilities and costs are computed.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
$P(r_i\|\Box)$	1	1	0	1	0	1	0	0
$Cost_{r_i}$	$-\frac{5}{9}$	$-\frac{2}{9}$	$-\frac{5}{9}$	$-\frac{8}{9}$	$-\frac{4}{9}$	$-\frac{1}{9}$	$-\frac{1}{9}$	$-\frac{1}{9}$

We note there is a tie among the probabilities of rules r_1, r_2, r_4 and r_6. This rule ranking can be refined by taking into account rule correctness costs. Indeed, we have that r_6 is rated first and r_2 is rated second.

5 Generation of Examples, Scalability and Applications

In order to approximate in an appropriate way the ratings introduced before, we need a representative sample of examples. If we are given the sample S we can use it directly. However, if we have a specification I or an oracle O then we need to obtain random examples from it. A first idea in these latter cases is to generate terms by using the fix-point of the immediate consequence operator [6] until iteration k, i.e. ${T_R}^k$. However, the problem of this approach is that the number of examples cannot be told in advance with k and, more importantly, the complexity of the ${T_R}^k$ procedure depends on the size and complexity of the program. Consequently, it seems more reasonable to generate some set or multiset of terms (as *lhs* of examples) and use them to obtain (jointly with I or O) the *rhs* of the examples. If we are given a probability distribution over the terms or the examples, this is fairly clear, we can generate them with an appropriate generative grammar. However, which distribution should we assume if we are not given one? We have to determine a distribution on all the ground terms of the co-domain of the function. Since there may be infinite possible terms, a uniform distribution is not a good solution for this.

A way to obtain a representative sample with less assumptions but still feasible is the use of a universal distribution over the terms, which is defined as the distribution that gives more probability to short terms and less probability to large terms [8]. This is also beneficial for computing normal forms, since the size of the examples will not be, in general, too large, and for many programs, this would also mean relatively short rewriting chains.

It may be interesting to discuss whether short examples are more likely to include extreme or strange values (as it is usually recommended by techniques such as "Extreme Value Analysis"). Note that base cases, cases with extreme values, exceptional cases, etc. (of a single function) are precisely those which can be reached with a few rewriting steps in an overwhelming majority of applications (one can make up a very special program that can only have a special behaviour after a thousand calls to the same function, but this is not a good practice or it is not a very common example). In this way we would have that for many programs, if we generate a sufficiently large sample, almost all the small peculiar cases would be considered.

Regarding scalability, the generation of a sample of n terms according to the universal distribution can be considered in $\mathcal{O}(n)$. It is the computation of the *rhs* of each of these terms that may be the most computationally expensive part. As we have said, in many situations, short terms will have (in general) shorter derivations than large terms, and this will affect positively on the possibility of generating large amounts of examples for many applications. It is reasonable to think that larger programs will require more examples, although it is problem-dependent to know if the relation is sublinear, linear or exponential. The good thing of our method is that we can begin to estimate probabilities in an incremental way and we can stop when the cardinalities of each of the rule probabilities and costs are large enough to have good approximations. In other words, for each particular program, we can tune a specific compromise between time complexity and good estimation of probabilities and costs.

With respect to applicability, the first area where our approach is especially beneficial is in those cases where we have a relevant cost function or matrix. For instance, medical diagnosis, fault-detection, etc., are clear examples where our approach can take advantage of the cost information. Just as an example, it is more important to detect an error that alerts that a system is not working (when it is) than the reverse situation. A second area of application is the development of programs for which we are given many use cases as a specification or we can generate many of these. There is no need of a full specification in these cases. A third area of application is prototyping, when we want to detect the most evident errors first in order to show some basic functionalities. Additionally to these three specific areas, we think that our approach could also be beneficial in general, possibly depending on the kind of program to be debugged.

Finally, an alternative way of generating examples could be to use the universal distribution for the *derivations*. Hence, terms would be more probable the shorter their program derivation is. Nonetheless, we think that this approach would be less efficient (we have to consider both the *lhs* and the *rhs* wrt. the program) and more difficult to implement than the previous approach.

6 Conclusions

In this paper, we have shown that the analysis of rule use depending on the distribution of examples can also be used for detecting bugs and, more importantly, to priorise them. Developing a debugging ranking of rules must also take into account the context and distribution where the program is being used. Some cases handled by a program are much more frequent than others. This means that errors in frequent cases are corrected first than errors in less frequent cases or even cases that almost never occur. This information can be given by a probability distribution and hence used by our diagnosis framework when constructing the sample (instead of the general, by-default universal distribution). Moreover, this information can be complemented with the cost function, because it may also be the case that rare situations may have more cost than common situations.

These considerations are, to the best of our knowledge, quite new in the typical discourse of program debugging, but are well-known in what is called cost-sensitive learning. Other techniques, such as ROC analysis [5] could also be applied for diagnosis.

The efficiency of our method is proportional to the number of examples used. The bigger the sample the more precise the estimation of probabilities and costs becomes, but the efficiency decreases (linearly).

Another advantage of this framework, inherited from its simplicity, is that it can also be applied to other declarative and even non-declarative rule-based languages, provided the rules have enough granularity.

One way to advance in some complex cases (especially recursive ones) could be to explore rule dependencies. Another issue to be explored is that the probabilities and costs could be computed exactly and not approximated, when we are given the specification. This would turn this method into a purely semantical one instead of a statistical one. As future work, we also plan to extend our cost analysis in order to deal with completeness errors and nonterminating programs.

Finally, in this work we have only considered detection and ranking of bugs, but not their correction. In many cases, inductive techniques (i.e. learning) should be used for this. Systems that are able to work with cost functions would be more coherent and could be coupled with this approach.

Acknowledgements

We would like to thank the WFLP'03 anonymous referees for their helpful comments which have helped to significantly improve this paper.

References

1. M. Alpuente, F. J. Correa, and M. Falaschi. Debugging Scheme of Functional Logic Programs. In M. Hanus, editor, *Proc. of International Workshop on Functional and (Constraint) Logic Programming, WFLP'01*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
2. R. Caballero-Roldán, F.J. López-Fraguas, and M. Rodríquez Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Fifth International Symposium on Functional and Logic Programming*, volume 2024 of *LNCS*, pages 170–184. Springer-Verlag, 2001.
3. B. Cestnik. Estimating probabilities: A crucial task in machine learning. In *Proc. Ninth European Conference on Artificial Intelligence*, pages 147–149, London, 1990. Pitman.
4. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
5. J.A. Hanley and B.J. McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology, 143*, pages 29–36, 1982.
6. S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *LNAI*. Springer Verlag, 1989.
7. J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.
8. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications.* 2nd Ed. Springer-Verlag, 1997.
9. H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(1):337–370, July 1994.

Model Checking Erlang Programs – LTL-Propositions and Abstract Interpretation

Frank Huch

Christian-Albrechts-University of Kiel,Germany
fhu@informatik.uni-kiel.de

Abstract. We present an approach for the formal verification of Erlang programs using abstract interpretation and model checking. In previous work we defined a framework for the verification of Erlang programs using abstract interpretation and LTL model checking. The application of LTL model checking yields some problems in the verification of state propositions, because propositions are also abstracted in the framework. In dependence of the number of negations in front of state propositions in a formula they must be satisfied or refuted. We show how this can automatically be decided by means of the abstract domain.
The approach is implemented as a prototype and we are able to prove properties like mutual exclusion or the absence of deadlocks and lifelocks for some Erlang programs.

1 Introduction

Growing requirements of industry and society impose greater complexity of software development. Consequently, understandability, maintenance and reliability cannot be warranted. This gets even harder when we leave the sequential territory and develop distributed systems. Here many processes run concurrently and interact via communication. This can yield problems like deadlocks or lifelocks. To guarantee the correctness of software, formal verification is needed.

In industry the programming language Erlang [1] is used for the implementation of distributed systems. For formal verification, we have developed a framework for abstract interpretations [4, 11, 15] for a core fragment of Erlang [7]. This framework guarantees that the transition system defined by the abstract operational semantics (*AOS*) includes all paths of the standard operational semantics (*SOS*). Because the AOS can sometimes have more paths than the SOS, it is only possible to prove properties that have to be satisfied on all paths, like in linear time logic (*LTL*). If the abstraction satisfies a property expressed in LTL, then also the program satisfies it, but not vice versa. If the AOS is a finite transition system, then model checking is decidable [13, 16]. For finite domain abstract interpretations and an additional flow-abstraction [9] this finite state property can be guaranteed for Erlang programs which do not create an unbound number of processes and use only mailboxes of restricted size.

However, the application of LTL model checking to the AOS is not so straightforward. LTL is usually defined by means of simple state propositions. In our approach of verification by model checking, the validity of a state proposition depends on the chosen abstraction. This must be taken into account in the implementation of the model checking algorithm. Furthermore, negation in LTL formulas results in additional problems. The equivalence of not validating a state proposition and refuting it does not hold in the context of abstraction. We show how safe model checking with respect to the abstract interpretation can be implemented.

The paper is organized as follows: Section 2 introduces the programming language Erlang and Section 3 shortly presents our framework for abstract interpretation. In Section 4 we add propositions to Erlang programs which can be used for formal verification by LTL model checking, shortly introduced in Section 5. Then, Section 6 motivates how abstract state propositions can be decided, which is formalized in Section 7. Finally, we present a concrete verification in Section 9 and conclude in Section 10.

2 Erlang

Erlang is a strict functional programming language. It is extended with concurrent processes. In Erlang, variables start with an uppercase letter. They can be bound to arbitrary values, with no type restrictions. Basic values are atoms (starting with a lowercase letter), e.g. `true,false`. More complex data structures can be created by constructors for tuples of any arity (`{...}`$/n$) and constructors for lists (`[.|.]`and `[]`). It is also possible to use atoms and these constructors in pattern matching.

In Erlang, a new process can be created by `spawn(`f, `[`$a_1, \ldots, a_n$`]`). The process starts with the evaluation of $f(a_1, \ldots, a_n)$. Since Erlang is strict the arguments of `spawn` are evaluated before the new process is created. The functional result of `spawn` is the process identifier (*pid*) of the newly created process. With p`!`v arbitrary values (including pids) can be sent to other processes. The processes are addressed by their pids (p). A process can access its own pid with the Erlang function `self/0`. The messages sent to a process are stored in its mailbox and the process can access them conveniently with pattern matching in a `receive`-statement. Especially, it is possible to ignore some messages and fetch messages from further behind. For more details see [1].

For the formal verification we restrict to a core fragment of Erlang (called *Core Erlang*) which contains the main features of Erlang. The syntax of Core Erlang is defined as follows:

$$\begin{array}{lll}
p & ::= & f\texttt{(}X_1\texttt{,}\ldots\texttt{,}X_n\texttt{)} \texttt{ -> } e\texttt{.} \mid p\ p \\
e & ::= & \phi\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mid X \mid pat \texttt{ = } e \mid \texttt{self} \mid e_1\texttt{,}e_2 \mid e_1\texttt{!}e_2 \mid \\
 & & \texttt{case } e \texttt{ of } m \texttt{ end} \mid \texttt{receive } m \texttt{ end} \mid \texttt{spawn(}f\texttt{,}e\texttt{)} \\
m & ::= & p_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}p_n\texttt{->}e_n \\
pat & ::= & c\texttt{(}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{)} \mid X
\end{array}$$

where ϕ represents predefined functions, functions of the program and constructors (c).

Example 1. A database process for storing key-value pairs can be defined as follows:

```
main() -> DB = spawn(dataBase,[[]]),
          spawn(client,[DB]), client(DB).

dataBase(L) -> prop(top),
   receive
      {allocate,Key,P} ->
          prop({allocate,P}),
          case lookup(Key,L) of
             fail -> P!free,
                     receive
                        {value,V,P} -> prop({value,P}),
                                       dataBase([{Key,V}|L])
                     end;
             {succ,V} -> P!prop(allocated), dataBase(L)
          end;
      {lookup,Key,P} -> prop(lookup),
                        P!lookup(Key,L), dataBase(L)
   end.
```

At this point the reader should ignore the applications of the function `prop`. In Section 9 we will prove mutual exclusion for the database. For this purpose we will need the state propositions introduced by `prop`.

The program creates a database process holding a state (its argument `L`) in which the key-value pairs are stored. The database is represented by a list of key-value pairs. The communication interface of the database is given by the messages `{allocate,Key,P}` for allocating a new key and `{lookup,Key,P}` for retrieving the value of a stored key. In both patterns `P` is bound to the requesting client pid to which the answer is sent.

The allocation of a new key is done in two steps. First the key is received and checked. If there is no conflict, then the corresponding value can be received and stored in the database. This exchange of messages in more than one step has to guarantee mutual exclusion on the database. Otherwise, it could be possible that two client processes send keys and values to the database and they are stored in the wrong combination. A client can be defined accordingly [7]. In Section 9 we will prove that the database combined with two accessing clients satisfies this property.

2.1 Semantics

In [7] we presented a formal semantics for Core Erlang. In the following we will refer to it as standard operational semantics (*SOS*). It is an interleaving semantics over a set of processes $\Pi \in \mathit{States}$. A process ($\mathit{proc} \in \mathit{Proc}$) consists of a pid ($\pi \in \mathit{Pid} := \{\texttt{@n} \mid n \in \mathbb{N}\}$), a Core Erlang evaluation term e and a word over constructor terms, representing the mailbox.

The semantics is a non-confluent transition system with interleaved evaluations of the processes. Only communication and process creation have side effects to the whole system. For the modeling of these actions two processes are involved. To give an impression of the semantics, we present the rule for sending a value to another process:

$$\frac{v_1 = \pi' \in \mathit{Pid}}{\Pi, (\pi, E[v_1 \,!\, v_2], \mu)(\pi', e, \mu') \xrightarrow{!v_2} \Pi, (\pi, E[v_2], \mu)(\pi', e, \mu' : v_2)}$$

E is the context of the leftmost-innermost evaluation position, with v_1 a pid and v_2 an arbitrary constructor term. The value v_2 is added to the mailbox μ' of the process π' and the functional result of the send action is the sent value. For more details see [10].

3 Abstract Interpretation of Core Erlang Programs

In [7] we developed a framework for abstract interpretations of Core Erlang programs. The abstract operational semantics (*AOS*) yields a transition system which includes all paths of the SOS. In an abstract interpretation $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ for Core Erlang programs $\widehat{A}$ is the abstract domain which should be finite for our application in model checking. The abstract interpretation function $\widehat{\iota}$ defines the semantics of predefined function symbols and constructors. Its codomain is $\widehat{A}$. Therefore, it is for example not possible to interpret constructors freely in a finite domain abstraction. $\widehat{\iota}$ also defines the abstract behaviour of pattern matching in equations, `case`, and `receive`. Here the abstraction can yield additional non-determinism because branches can get undecidable in the abstraction. Hence, $\widehat{\iota}$ yields a set of results which defines possible successors. Furthermore, an abstract interpretation contains a partial order $\sqsubseteq$, describing which elements of $\widehat{A}$ are more precise than others[1]. We do not need a complete partial order or a lattice because we do not compute any fixed point. Instead, we just evaluate the operational semantics with respect to the abstract interpretation. This yields a finite transition system, which we use for (on the fly) model checking. An example for an abstraction of numbers with an ordering of the abstract representations is: $\mathbb{N} \sqsubseteq \{v \mid v \leq 10\} \sqsubseteq \{v \mid v \leq 5\}$. It is more precise to know, that a value is ≤ 5, than ≤ 10 than any number. The last component of $\widehat{\mathcal{A}}$ is the abstraction function. α which maps every concrete value to its most precise abstract representation. Finally, the abstract interpretation has to fulfill five properties, which relate an abstract interpretation to the standard interpretation. They also guarantee that all paths of the SOS are represented in the AOS, for example in branching. An example for these properties is the following:

(P1) For all $\phi/n \in \Sigma \cup C, v_1, \ldots, v_n \in T_C(Pid)$ and
$\widetilde{v}_i \sqsubseteq \alpha(v_i)$ it holds that $\phi_{\widehat{\mathcal{A}}}(\widetilde{v}_1, \ldots, \widetilde{v}_n) \sqsubseteq \alpha(\phi_{\mathcal{A}}(v_1, \ldots, v_n))$.

It postulates, that evaluating a predefined function or a constructor on abstract values which are representations of some concrete values yields abstractions of the evaluation of the same function on the concrete values. The other properties postulate correlating properties for pattern matching in equations, `case`, and `receive`, and the pids represented by an abstract value. More details and some example abstractions can be found in [7, 8].

4 Adding Propositions

The semantics of Core Erlang is defined as a labeled transition system where the labels represent the performed actions of the system. We want to prove properties of the system with model checking. It would be possible to specify properties using the labels. Nevertheless, it is more convenient to add propositions to the states of this

[1] Since we do not need a cpo, we use a partial order for the abstract domain. Therefore, the orientation of our abstract domain is upside down. The least precise abstract value is the least element of the abstract domain.

transition system. With these state propositions, properties can be expressed more easily. As names for the propositions we use arbitrary Core Erlang constructor terms which is very natural for Erlang programmers.

For the definition of propositions we assume a predefined Core Erlang function `prop/1` with the identity as operational semantics. Hence, adding applications of `prop` does not effect the SOS nor the AOS. Nevertheless, as a kind of side-effect the state in which `prop` is evaluated has the argument of `prop` as a valid state proposition. We mark this with the label prop in the AOS:

$$\frac{-}{\Pi, (\pi, E[\texttt{prop}(v)], \mu) \overset{\textsf{prop}}{\longrightarrow}_{\widehat{\mathcal{A}}} \Pi, (\pi, E[v], \mu)}$$

The valid state propositions of a process and a state over abstract values can be evaluated by the function prop:

Definition 1. *(Proposition of Processes and States*[2]*)*
The proposition of a process is defined by $\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{Proc}_{\widehat{\mathcal{A}}} \longrightarrow \mathcal{P}(\widehat{A})$:

$$\mathsf{prop}_{\widehat{\mathcal{A}}}((\pi, E[e], \mu)) := \begin{cases} \{\widehat{v}\} & , \textit{if } e = \texttt{prop}(\widehat{v}) \textit{ and } \widehat{v} \in \widehat{A} \\ \emptyset & , \textit{otherwise} \end{cases}$$

The propositions of a state $\mathsf{prop}_{\widehat{\mathcal{A}}} : \widehat{State}_{\widehat{\mathcal{A}}} \longrightarrow \mathcal{P}(\widehat{A})$ *are defined as the union of all propositions of its processes:*

$$\mathsf{prop}_{\widehat{\mathcal{A}}}(\Pi) = \bigcup_{\pi \in \Pi} \mathsf{prop}_{\widehat{\mathcal{A}}}(\pi)$$

In Example 1 we have added four propositions to the database which have the following meanings:

`top`	the main state of the database process
`{allocate,P}`	the process with pid `P` tries to allocate a key
`{value,P}`	the process with pid `P` enters a value into the database
`lookup`	a reading access to the database

In most cases, propositions will be added in a sequence as for example the proposition `top`. However, defining propositions by an Erlang function it is also possible to mark existing (sub-)expressions as propositions. As an example, we use the atom `allocated` which is sent to a requesting client, as a state proposition.

5 Linear Time Logic

The abstract operational semantics is defined by a transition system. We want to prove properties (described in temporal logic) of this transition system using model checking. We use *linear time logic* (*LTL*) [5] in which properties have to be satisfied on every path of a given transition system.

[2] For both functions we use the name $\mathsf{prop}_{\widehat{\mathcal{A}}}$. The concrete instance of this overloading will be clear from the application of $\mathsf{prop}_{\widehat{\mathcal{A}}}$. We will also omit the abstract interpretation in the index, if it is clear from the context. As for the abstract domain, we use a hat to distinguish the abstract variants (which contain abstract instead of concrete values) from the defined concrete structures (e.g. *Proc*).

Definition 2. *(Syntax of Linear Time Logic (LTL)) Let Props be a set of state propositions. The set of* LTL-formulas *is defined as the smallest set with:*

• $Props \subseteq \text{LTL}$		*state propositions*
• $\varphi, \psi \in \text{LTL} \Longrightarrow$	– $\neg\varphi \in \text{LTL}$	*negation*
	– $\varphi \wedge \psi \in \text{LTL}$	*conjunction*
	– $X\varphi \in \text{LTL}$	*in the next state φ holds*
	– $\varphi\ U\ \psi \in \text{LTL}$	*φ holds until ψ holds*

An LTL-formula is interpreted with respect to an infinite path. The propositional formulas are satisfied, if the first state of a path satisfies them. The next modality $X\varphi$ holds if φ holds in the continuation of the path. Finally, LTL contains a strong until: If φ holds until ψ holds and ψ finally holds, then $\varphi\ U\ \psi$ holds. Formally, the semantics is defined as:

Definition 3. *(Path Semantics of LTL) An infinite word over sets of propositions $\pi = p_0 p_1 p_2 \ldots \in \mathcal{P}(Props)^\omega$ is called a path. A path π satisfiess an LTL-formula φ ($\pi \models \varphi$) in the following cases:*

$$\begin{array}{rcll}
p_0\pi & \models & P & \textit{iff } P \in p_0 \\
\pi & \models & \neg\varphi & \textit{iff } \pi \not\models \varphi \\
\pi & \models & \varphi \wedge \psi & \textit{iff } \pi \models \varphi \textit{ and } \pi \models \psi \\
p_0\pi & \models & X\varphi & \textit{iff } \pi \models \varphi \\
p_0 p_1 \ldots & \models & \varphi\ U\ \psi & \textit{iff } \exists i \in \mathbb{N} : p_i p_{i+1} \ldots \models \psi \textit{ and } \forall j < i : p_j p_{j+1} \ldots \models \varphi
\end{array}$$

Formulas are not only interpreted with respect to a single path. Their semantics is extended to Kripke Structures:

Definition 4. *(Kripke Structure) $\mathcal{K} = (S, Props, \longrightarrow, \tau, s_0)$ with S a set of states, Props a set of propositions, $\longrightarrow \subseteq S \times S$ the transition relation, $\tau : S \longrightarrow \mathcal{P}(Props)$ a labeling function for the states, and $s_0 \in S$ the initial state is called a* Kripke Structure. *Instead of $(s, s') \in \longrightarrow$ we usually write $s \longrightarrow s'$.*

A state path of $\mathcal{K}$ *is an infinite word $s_0 s_1 \ldots \in S^\omega$ with $s_i \longrightarrow s_{i+1}$ and s_0 the initial state of $\mathcal{K}$. If $s_0 s_1 \ldots$ is a state path of $\mathcal{K}$ and $p_i = \tau(s_i)$ for all $i \in \mathbb{N}$, then the infinite word $p_0 p_1 \ldots \in \mathcal{P}(Props)^\omega$ is a* path of $\mathcal{K}$.

Definition 5. *(Kripke-Structure-Semantics of LTL) Let $\mathcal{K} = (S, \rightarrow, \tau, s_0)$ be a Kripke structure. It satisfies an LTL-formula φ ($\mathcal{K} \models \varphi$) iff for all paths π of $\mathcal{K}$: $\pi \models \varphi$.*

The technique of *model checking* automatically decides, if a given Kripke structure satisfies a given formula. For finite Kripke structures and the logic LTL model checking is decidable [13]. For the convenient specification of properties in LTL we define some abbreviations:

Definition 6. *(Abbreviations in LTL)*

ff	$:=$	$\neg P \wedge P$	*the boolean value true*
tt	$:=$	$\neg ff$	*the boolean value false*
$\varphi \vee \psi$	$:=$	$\neg(\neg\varphi \wedge \neg\psi)$	*disjunction*
$\varphi \rightarrow \psi$	$:=$	$\neg\varphi \vee \psi$	*implication*
$F\ \varphi$	$:=$	$tt\ U\ \varphi$	*finally φ holds*
$G\ \varphi$	$:=$	$\neg F \neg\varphi$	*globally φ holds*
$F^{\infty}\varphi$	$:=$	$G\ F\ \varphi$	*infinitely often φ holds*
$G^{\infty}\varphi$	$:=$	$F\ G\ \varphi$	*only finally often φ does not hold*

The propositional abbreviations are standard. $F\varphi$ is satisfied if there exists a position in the path, where φ holds. If in every position of the path φ holds, then $G\varphi$ is satisfied. The formulas φ which have to be satisfied in these positions of the path are not restricted to propositional formulas. They can express properties of the whole remaining path. This fact is used in the definition of $F^{\infty}\varphi$ and $G^{\infty}\varphi$. The weaker property $F^{\infty}\varphi$ postulates, that φ holds infinitely often on a path. Whereas $G^{\infty}\varphi$ is satisfied, if φ is satisfied with only finitely many exceptions. In other words there is a position, from where on φ always holds.

For the verification of Core Erlang programs we use the AOS respectively the SOS of a Core Erlang program as a Kripke structure. We use the transition system which is spawned from the initial state (`@0`,`main()`,()). As labeling function for the states we use the function `prop` from the previous section.

6 Abstraction of Propositions

We want to verify Core Erlang programs with model checking. The framework for abstract interpretations of Core Erlang programs guarantees, that every path of the SOS is also represented in the AOS. If the resulting AOS is finite, then we can use simple model checking algorithms to check, if it satisfies a property φ expressed in LTL. If φ is satisfied in the AOS, then φ also holds in the SOS. In the other case model checking yields a counter example which is a path in the AOS on which φ is not satisfied. Due to the fact that the AOS contains more paths than the SOS, the counter example must not be a counter example for the SOS. The counter path can be a valid path in the abstraction but not in the SOS. Therefore, in this case it only yields a hint, that the chosen abstraction is too coarse and must be refined.

The application of model checking seems to be easy but proving state propositions yields problems as the following example shows:

Example 2. `main() -> prop(42).`

A possible property of the program could be F **42** (finally 42). To prove this property we use the AOS with an abstract interpretation, for instance the even-odd interpretation, which only contains the values **even** (representing all even numbers), **odd** (representing all odd numbers), and **?** (representing all values). With this abstraction the AOS yields the following transition system:

$$(\texttt{@0}, \texttt{main()}, ()) \longrightarrow (\texttt{@0}, \texttt{prop(42)}, ()) \longrightarrow (\texttt{@0}, \texttt{prop}(\textbf{even}), ()) \xrightarrow{\textsf{prop}} (\texttt{@0}, \textbf{even}, ())$$

Only the state $(\texttt{@0}, \texttt{prop}(\textbf{even}), ())$ has a property, namely **even**. **42** is an even number, but it is not the only even number. For safeness, this property cannot be proven. For instance, we could otherwise also prove the property F **40**.

It is only possible to prove properties for which the corresponding abstract value exclusively represents this value. However, it does not make much sense to abstract from special values and express properties for these values afterwards. Therefore, we only use propositions of the abstract domain, like

$$F\ \textbf{even} \quad \text{(finally even)}$$

In the AOS the state (`@0`,`prop(`**even**`)`,()) has the property **even**. Therefore, the program satisfies this property. Now we consider a more complicated example:

Example 3. `main() -> prop(84 div 2).`
This system satisfies the property too because $(84 \div 2) = 42$. On the other hand, in the even-odd abstraction we only get:

$$\begin{tikzcd}
(\texttt{@0}, \texttt{main}(), ()) \arrow[r] & (\texttt{@0}, \texttt{prop}(84\ \texttt{div}\ 2), ()) \arrow[r] & (\texttt{@0}, \texttt{prop}(\textbf{even}\ \texttt{div}\ 2), ()) \arrow[d] \\
(\texttt{@0}, ?, ()) & (\texttt{@0}, \texttt{prop}(?), ()) \arrow[l] & (\texttt{@0}, \texttt{prop}(\textbf{even}\ \texttt{div}\ \textbf{even}), ()) \arrow[l]
\end{tikzcd}$$

with $\textsf{prop}((\texttt{@0},\texttt{prop(?)},())) = \{?\}$ and $\emptyset$ as propositions of the other states. The result of the division of two even values must not be even. In a safe abstraction we cannot be sure, that the property F **even** is satisfied. Hence, model checking must yield, that it does not hold. For instance, the program

```
  main() -> prop(42 div 2).
```

has a similar AOS but the property is not satisfied ($42 \div 2 = 21$). Therefore, a property is satisfied in a state, if the property of the state is at least as precise, as the expected property:

$$p_0 p_1 \ldots \models \widetilde{v} \text{ iff } \exists \widetilde{v}' \in p_0 \text{ with } \widetilde{v} \sqsubseteq \widetilde{v}'$$

However, this is not correct in all cases, as the following example shows. We want to prove that the program satisfies the property $G\neg$**even** (always not even) Therefore, one point is to check that the state (`@0`,`prop(?)`,()) models $\neg$**even**. With the definition from above we can conclude:

$$(\texttt{@0},\texttt{prop(?)},()) \not\models \textbf{even} \qquad \text{and hence} \qquad (\texttt{@0},\texttt{prop(?)},()) \models \neg\textbf{even}.$$

However, this is wrong. In Example 3 the property is not satisfied. The SOS has the property 42, which is an even value. The reason is the non-monotonicity of $\neg$. Considering abstraction, the equivalence

$$\pi \models \neg\varphi \text{ iff } \pi \not\models \varphi$$

does not hold! $\pi \not\models \varphi$ only means that $\pi \models \varphi$ is not safe. In other words, there can be a concretization which satisfies φ but we cannot be sure that it holds for all concretizations. Therefore, negation has to be handled carefully.

Which value of our abstract domain would fulfill the negated proposition $\neg$**even**? Only the proposition **odd** does. The values **even** and **odd** are incomparable and no value exists, which is more precise than these two abstract values. This connection can be generalized as follows:

$$p_0 p_1 \ldots \models \neg\widetilde{v} \text{ if } \forall \widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist}$$

Note, that this is no equivalence anymore. The non-existence of $\widetilde{v} \sqcup \widetilde{v}'$ does only imply that $p_0 p_1 \ldots \models \neg\widetilde{v}$. It does not give any information for the negation $p_0 p_1 \ldots \models \widetilde{v}$. This (double) negation holds, if $\exists \widetilde{v}' \in p_0$ with $\widetilde{v} \sqsubseteq \widetilde{v}'$.

On a first sight refuting a state proposition seems not to be correct for arbitrary abstract interpretations. Consider the abstract domain

$$\begin{tikzcd}
\leq\mathbf{0} & & \geq\mathbf{0} \\
& \mathbf{num} \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

where the abstract value $\leq\mathbf{0}$ represents the concrete values $\{\mathbf{0}, -\mathbf{1}, -\mathbf{2}, \ldots\}$, $\geq\mathbf{0}$ represents $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$, and **num** represents $\mathbb{Z}$. The represented concrete values of $\leq\mathbf{0}$ and $\geq\mathbf{0}$ overlap in the value $\mathbf{0}$. Therefore, it would be incorrect that a state with the proposition $\leq\mathbf{0}$ satisfies the formula $\neg\geq\mathbf{0}$.

However, this abstraction is not possible. Any abstraction function $\alpha : A \longrightarrow \widehat{A}$ yields one single abstract representation for a concrete value. Without loss of generality, let $\alpha(\mathbf{0}) = \geq\mathbf{0}$. Abstract values which represent the concrete value $\mathbf{0}$ can only be the result of the use of the abstract interpretation function $\widehat{\iota}$. However, all these results $\widetilde{v}$ must be less precise: $\widetilde{v} \sqsubseteq \alpha(\mathbf{0}) = \geq\mathbf{0}$ because of the properties claimed by our framework. Hence, this abstract domain can be defined but the value $\leq\mathbf{0}$ does only represent the values $\{-\mathbf{1}, -\mathbf{2}, \ldots\}$. The name of the abstract value is not relevant. However, for understandability it should be renamed to $<\mathbf{0}$.

Alternatively, the abstract domain can be refined. The two overlapping abstract values can be distinguished by a more precise abstract value:

$$\begin{tikzcd}
<\mathbf{0} & & \mathbf{0} & & >\mathbf{0} & & & \mathbf{0} & \\
& \leq\mathbf{0} \arrow[ul, dash] \arrow[ur, dash] & & \geq\mathbf{0} \arrow[ul, dash] \arrow[ur, dash] & & \text{or} & \leq\mathbf{0} \arrow[ur, dash] & & \geq\mathbf{0} \arrow[ul, dash] \\
& & \mathbf{num} \arrow[ul, dash] \arrow[ur, dash] & & & & & \mathbf{num} \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

In both cases we must define $\alpha(\mathbf{0}) = \mathbf{0}$ because otherwise we have the same situation as before and the concrete value $\mathbf{0}$ is not represented by both abstract values $\leq\mathbf{0}$ and $\geq\mathbf{0}$.

7 Concretization of Propositions

With the advisement of the previous section we can now formalize whether a state proposition is satisfied or refuted, respectively. Similar results have been found by Clark, Grumberg, and Long [3] and Knesten and Pnueli [12]. In their results Knesten and Pnueli handle a simple toy language in which only propositions on integer values can be made. In our framework state propositions are arbitrary Erlang values and their validity has to be decided with respect to an arbitrary abstract domain. The same holds for the paper of Clark et. al which also does not consider a real programming language. In following, we present how these ideas can be transfered to formal verification of the real programming language Erlang.

First we define the concretization of an abstract value. This is the set of all concrete values which have been abstracted to the value, or a more precise value.

Definition 7. *(Concretization of Abstract Values)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ *be an abstract interpretation. The concretization function* $\gamma : \widehat{A} \longrightarrow \mathcal{P}(T_{\mathcal{C}}(Pid))$ *is defined as* $\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}$.

For the last example we get the following concretizations:

$$\begin{array}{lclclcl} \gamma(\mathbf{0}) & = & \{\mathbf{0}\} & \qquad & \gamma(\geq\mathbf{0}) & = & \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\} \\ \gamma(\leq\mathbf{0}) & = & \{\mathbf{0}, -\mathbf{1}, -\mathbf{2}, \ldots\} & & \gamma(\mathbf{num}) & = & \mathbb{Z} \end{array}$$

Lemma 1. *(Connections between* γ *and* α*)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ *be an abstract interpretation and* γ *the corresponding concretization function. Then the following properties hold:*

1. $\forall v \in \gamma(\widetilde{v}) : \widetilde{v} \sqsubseteq \alpha(v)$ *2.* $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \widetilde{v}$

Proof.
1. $v \in \gamma(\widetilde{v})$ iff $v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}$ iff $\widetilde{v} \sqsubseteq \alpha(v)$
2. $\bigsqcap\{\alpha(v) \mid v \in \gamma(\widetilde{v})\} = \bigsqcap\{\alpha(v) \mid v \in \{v' \mid \widetilde{v} \sqsubseteq \alpha(v')\}\} = \bigsqcap\{\alpha(v) \mid \widetilde{v} \sqsubseteq \alpha(v)\} = \widetilde{v}$

With the concretization function we can define whether a state proposition of a state satisfies a proposition in the formula or refutes it.

Definition 8. *(Semantics of a State Proposition)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ *be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:*

$$\begin{array}{l} p \models \widetilde{v} \;\; \textit{if} \;\; \exists \widetilde{v}' \in p \textit{ with } \gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v}) \\ p \not\models \widetilde{v} \;\; \textit{if} \;\; \forall \widetilde{v}' \in p \textit{ holds } \gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset \end{array}$$

Similarly to these definitions for the concretization, we can decide whether a state proposition if satisfied or refuted for abstract values. For finite domain abstractions, this can be decided automatically.

Lemma 2. *(Deciding Propositions in the abstract domain)*
Let $\widehat{\mathcal{A}} = (\widehat{A}, \widehat{\iota}, \sqsubseteq, \alpha)$ be an abstract interpretation. A set of abstract state propositions satisfies or refutes a proposition of a formula in the following cases:

$$\begin{array}{l} p \models \widetilde{v} \;\; \textit{if} \;\; \exists \widetilde{v}' \in p \textit{ with } \widetilde{v} \sqsubseteq \widetilde{v}' \\ p \not\models \widetilde{v} \;\; \textit{if} \;\; \forall \widetilde{v}' \in p \textit{ holds } \widetilde{v} \sqcup \widetilde{v}' \textit{ does not exist} \end{array}$$

Proof. We show: $\widetilde{v} \sqsubseteq \widetilde{v}'$ implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$ and the non-existence of $\widetilde{v} \sqcup \widetilde{v}'$ implies $\gamma(\widetilde{v}) \cap \gamma(\widetilde{v}') = \emptyset$:

- $\widetilde{v} \sqsubseteq \widetilde{v}'$
  $\gamma(\widetilde{v}) = \{v \mid \widetilde{v} \sqsubseteq \alpha(v)\}$ and $\gamma(\widetilde{v}') = \{v \mid \widetilde{v}' \sqsubseteq \alpha(v)\}$.
  $(\widehat{A}, \sqsubseteq)$ is a partial order. Hence, it is transitive. This implies $\gamma(\widetilde{v}') \subseteq \gamma(\widetilde{v})$.
- $\widetilde{v} \sqcup \widetilde{v}'$ does not exist $\Longrightarrow \gamma(\widetilde{v} \sqcup \widetilde{v}') = \emptyset \Longrightarrow \{v \mid (\widetilde{v} \sqcup \widetilde{v}') \sqsubseteq \alpha(v)\} = \emptyset$
  $\Longrightarrow \{v \mid \widetilde{v}' \sqsubseteq \alpha(v) \text{ and } \widetilde{v} \sqsubseteq \alpha(v)\} = \emptyset \Longrightarrow \gamma(\widetilde{v}') \cap \gamma(\widetilde{v}) = \emptyset$

Note, that we only show an implication. We can define unnatural abstract domains in which a property is satisfied or refuted with respect to Definition 8 but using only the abstract domain, we cannot show this. We consider the following abstract domain:

$$\mathbf{num} \sqsubseteq \mathbf{zero} \sqsubseteq \mathbf{0} \qquad \text{with } \alpha(v) = \begin{cases} \mathbf{0} & \text{, if } v = \mathbf{0} \\ \mathbf{num} & \text{otherwise} \end{cases}$$

The abstract value **zero** is superfluous because it represents exactly the same values, as the abstract value **0**. However, this abstract domain is valid. Using the definition of the semantics of a state proposition from Definition 8, we can show that $\{\mathbf{zero}\} \models \mathbf{0}$ because $\gamma(\mathbf{zero}) = \gamma(\mathbf{0}) = \{\mathbf{0}\}$. However, $\mathbf{zero} \sqsubseteq \mathbf{0}$ and we cannot show that $\{\mathbf{zero}\} \models \mathbf{0}$ just using the abstract domain.

The same holds for refuting a state proposition:

$$\begin{tikzcd}[column sep=small] & \mathbf{0} \arrow[dl, dash] \arrow[dr, dash] & \\ \leq\mathbf{0} \arrow[dr, dash] & & \geq\mathbf{0} \arrow[dl, dash] \\ & \mathbf{num} & \end{tikzcd} \qquad \text{with } \alpha(v) = \begin{cases} \geq\mathbf{0} & \text{, if } v \geq \mathbf{0} \\ \leq\mathbf{0} & \text{otherwise} \end{cases}$$

In this domain the abstract value **0** is superfluous. Its concretization is empty. Hence, $\gamma(\leq\mathbf{0}) = \{-\mathbf{1}, -\mathbf{2}, \ldots\}$ and $\gamma(\geq\mathbf{0}) = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}$. $\gamma(\leq\mathbf{0}) \cap \gamma(\geq\mathbf{0}) = \emptyset$ and $\leq\mathbf{0} \models \neg\geq\mathbf{0}$. However, this proposition cannot be refuted with this abstract domain because $\leq\mathbf{0} \sqcup \geq\mathbf{0} = \mathbf{0}$ exists.

These examples are unnatural because the domains contain superfluous abstract values. Nobody will define domains like these. Usually, the concretization of an abstract value is nonempty and differs from the concretizations of all other abstract values. In this case deciding propositions in the abstract domain is complete with respect to the semantics of propositions. Although it is not complete in general, it is safe. If we can prove a property with the abstract values, then it is also correct for its concretizations.

8 Proving LTL Formulas

So far we have discussed whether a state proposition is satisfied or refuted. However, in LTL negation is not only allowed in front of state propositions. Arbitrary sub-formulas can be negated. For the decision of arbitrary LTL formulas with respect to an abstract interpretation two approaches are possible: In the first approach, all negations can be pushed into the formula until they only occur in front of state propositions. Since there exists no equivalent representation of $\neg(\varphi U \psi)$ which uses negation only in front of φ and ψ, we must extend LTL with the release modality $\varphi R \psi$, the dual modality of until: $\neg(\varphi U \psi) \sim \neg\varphi R \neg\psi$. Its semantics is defined as:

$$p_0 p_1 \ldots \models \varphi R \psi \text{ iff } \forall i \in \mathbb{N} : p_i p_{i+1} \ldots \models \psi \text{ or } \exists j < i : p_j p_{j+1} \ldots \models \varphi$$

There is no intuitive semantics of release, except that it can be used for the negation of until. However, it can also be automatically verified in model checking.

Furthermore, we must add $\vee$ to LTL. After pushing the negations in front of the state propositions we can use standard model checking algorithms in combination with Lemma 2 for deciding state propositions.

Standard model checking algorithms work with a similar idea but they do not need the release modality. For example, in [16] an alternating automaton is constructed, that represents the maximal model which satisfies the formula. The states correspond to the possible sub-formulas and their negations. For every negation in the formula the automaton switches to the corresponding state which represents the positive or negative sub-formula. With this alternation negations are pushed into the automaton representing the formula, like in the first approach. We use a similar idea as in the second approach and decide the state propositions in dependence of the negation in front. We distinguish positive (marked with $+$) and negative (marked with $-$) state propositions in dependence on an even respectively odd number of negations in front of them. The advantage of this approach is that the formula is not transformed and its semantics stays more intuitive. The two kinds of state propositions can then be decided as follows:

$$\begin{array}{lll} p_0 p_1 \ldots \models^{+} \widetilde{v} & \text{if} & \exists \widetilde{v}' \in p_0 \text{ with } \widetilde{v} \sqsubseteq \widetilde{v}' \\ p_0 p_1 \ldots \not\models^{-} \widetilde{v} & \text{if} & \forall \widetilde{v}' \in p_0 \text{ holds } \widetilde{v} \sqcup \widetilde{v}' \text{ does not exist} \end{array}$$

9 Verification of the Database

Now we want to verify the system of Example 1. A database process and two clients are executed. We want to guarantee, that the process which allocates a key also sets the corresponding value:

> If a process π allocates a key, then no other process π' sets a value before π sets a value, or the key is already allocated.

This can for arbitrary processes be expressed in LTL as follows:

$$\bigwedge_{\substack{\pi \in Pid \\ \pi' \neq \pi}} G\ ({}^{-}\{\mathbf{allocate}, \pi\} \longrightarrow (\neg\, {}^{-}\{\mathbf{value}, \pi'\})\ U\ ({}^{+}\{\mathbf{value}, \pi\} \vee\ {}^{+}\mathbf{allocated}))$$

In our system only a finite number of pids occurs. Therefore, this formula can be translated into a pure LTL-formula as a conjunction of all possible permutations of possible pids which satisfy the condition. This is

$$(\pi, \pi') \in \{(\texttt{@0}, \texttt{@1}), (\texttt{@0}, \texttt{@2}), (\texttt{@1}, \texttt{@2}), (\texttt{@1}, \texttt{@0}), (\texttt{@2}, \texttt{@0}), (\texttt{@2}, \texttt{@1})\}.$$

We have already marked the propositions in the formula with respect to the negations in front of them. Considering this marking the proposition $^{-}\{\textbf{allocate}, \pi\}$ must be refuted. This is for example the case for the abstract values **top** and **{lookup,?}**. However, **?** and **{allocate**,p**}** with p the pid of the accessing client do not refute the proposition. The right side of the implication must be satisfied. Similar conditions must hold for the other propositions.

We can automatically verify this property using a finite domain abstraction, in which only the top-parts of depth 2 of the constructor terms are considered. The deeper parts of a constructor term are cut off and replaced by **?**. For more details see [8]. Our framework guarantees that the property also holds in the SOS and we have proven mutual exclusion for the database program.

10 Related Work and Conclusion

There exist two other approaches for the formal verification of Erlang: EVT [14] is a theorem prover especially tailored for Erlang. The main disadvantage of this approach is the complexity of proves. Especially, induction on infinite calculations is very complicated and automatization is not support yet. Therefore, a user must be an expert in theorem proving and EVT to prove system properties. We think, for the practical verification of distributed systems push-button techniques are needed. Such a technique is model checking, which we use for the automated verification of Erlang programs. This approach is also pursued by part of the EVT group in [2]. They verified a distributed resource locker written in Erlang with a standard model checker. The disadvantage of this approach is that they can only verify finite state systems. However, in practice many systems have an infinite (or for model checkers too large) state space. As a solution, we think abstraction is needed to verify larger distributed systems. A similar approach for Java is made by the Bandera tool [6]. They provide abstraction, but the user is responsible to define consistent abstractions by explicitly defining abstracted methods in the source code. They use a standard model checkers in which the presented problems of abstraction are not considered. Our approach for the formal verification of Erlang programs uses abstract interpretation and LTL model checking. The main idea is the construction of a finite model by the AOS by means of a finite domain abstract interpretation. The abstraction is safe in the sense, that all path of the SOS are also represented in the AOS.

For convenient verification we have added state propositions to the AOS. Considering the abstract interpretation, problems in the semantics of these propositions arise. In this paper we solved these problems by distinguishing positive and negative propositions of the formula. For these we can decide, if a state satisfies or refutes it by means of the abstract domain. Finally, we used this technique in the formal verification of the database process: we proved mutual exclusion for two accessing clients.

References

1. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
2. Thomas Arts and Clara Benac Earle. Development of a verified Erlang program for resource locking. In *Formal Methods in Industrial Critical Systems*, Paris, France, July 2001.
3. Edmund Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.
5. Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM SIGACT and SIGPLAN, ACM Press, 1980.
6. John Hatcliff, Matthew B. Dwyer, and Shawn Laubach. Staging static analyses using abstraction-based program specialization. *LNCS*, 1490:134–148, 1998.
7. Frank Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).
8. Frank Huch. Verification of Erlang programs using abstract interpretation and model checking – extended version. Technical Report 99–02, RWTH Aachen, 1999.
9. Frank Huch. Model checking erlang programs - abstracting recursive function calls. In *Proceedings of WFLP'01, ENTCS*, volume 64. Elsevier, 2002.
10. Frank Huch. *Verification of Erlang Programs using Abstract Interpretation and Model Checking.* PhD thesis, RWTH Aachen, Aachen, 2002.
11. Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science.* Oxford University Press, 1994. 527–629.
12. Yonit Kesten and Amir Pnueli. Modularization and abstraction: The keys to practical formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *The 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS 1998)*, volume 1450 of *LNCS*, pages 54–71. Springer, 1998.
13. Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, 1985. ACM SIGACT-SIGPLAN, ACM Press.
14. Thomas Noll, Lars-åke Fredlund, and Dilian Gurov. The Erlang verification tool. In *Proceedings of TACAS'01*, volume 2031 of *LNCS*, pages 582–585. Springer, 2001.
15. David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. *LNCS*, 1503:351–380, 1998.
16. Moshe Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *LNCS*, pages 238–266. Springer, New York, NY, USA, 1996.

An Open System to Support Web-based Learning[⋆] (Extended Abstract)

Michael Hanus   Frank Huch

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,fhu}@informatik.uni-kiel.de

Abstract. In this paper we present a system, called *CurryWeb*, to support web-based learning. Characteristic features of this system is openness and self-responsible use. Openness means that there is no strong distinction between instructors and students, i.e., every user can learn with the system or add new learning material to it. Self-responsible use means that every user is responsible for selecting the right material to obtain the desired knowledge. However, the system supports this selection process by structuring all learning material hierarchically and as a hypergraph whose nodes and edges are marked with educational objectives and educational units, respectively.
The complete system is implemented with the declarative multi-paradigm language Curry. In this paper we describe how the various features of Curry support the high-level implementation of this system. This shows the appropriateness of declarative multi-paradigm languages for the implementation of complex web-based systems.

Keywords: functional logic programming, WWW, applications

1 Overview of CurryWeb

CurryWeb is a system to support web-based learning that has been mainly implemented by students during a practical course on Internet programming. A key idea of this system was to support self-responsibility of the users (inspired by similar ideas in the development of open source software). Thus, every user can insert learning material on a particular topic. The quality of the learning material can be improved by providing feedback through critics and ranking, or putting new (hopefully better) material on the same topic.

In order to provide a structured access to the learning material, CurryWeb is based on the following notions:

Educational Objectives: An educational objective names a piece of knowledge that can be learned or is required to read and understand some learning material. Since it is not possible to give this notion a formal meaning,

⋆ This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD under the programme Acciones Integradas Hispano-Alemanas.

each educational objective has a distinguished name. Its meaning can be informally described by a document.

Educational Unit: An educational unit is the smallest piece of learning material in the system. Each educational unit has *prerequisites* that are required to be able to understand this unit and *objectives* that can be learned by studying this unit. Both prerequisites and objectives are sets of educational objectives. Thus, the educational units together with their prerequisites and objectives form a hypergraph where educational objectives are the nodes and each educational unit is a hyperedge (a directed edge with $m \geq 0$ source and $n \geq 0$ target nodes).

Courses: A course is a sequence of elements where each element is an educational unit or a course. There is exactly one root course from which all other courses are reachable. Thus, the courses structure all educational units as a tree where a leaf (educational unit) can belong to more than one course.

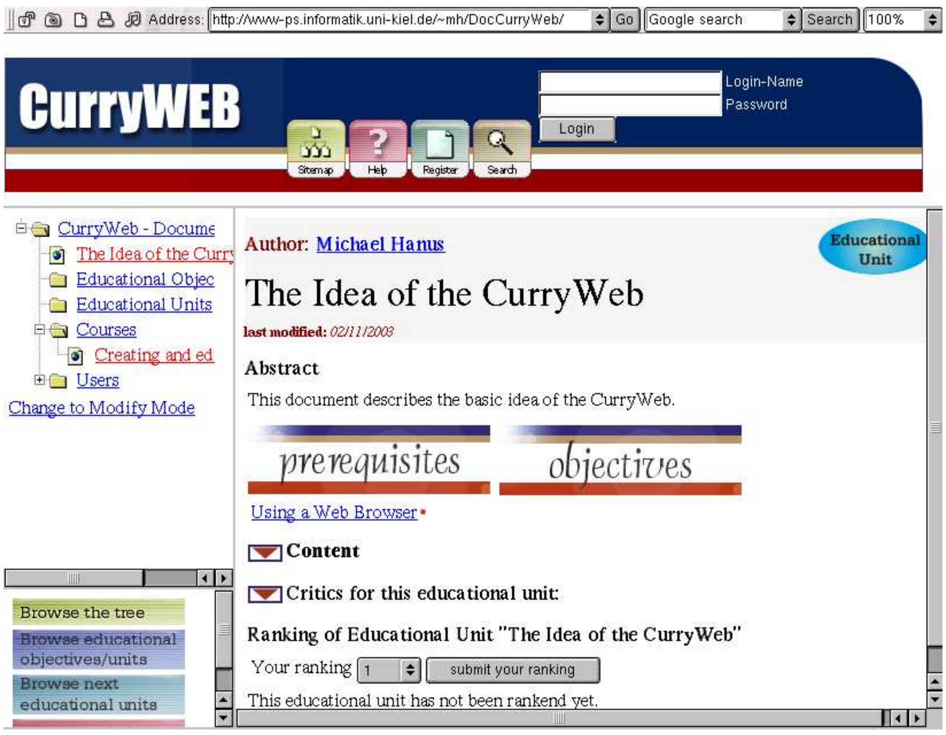
The course and hypergraph structure also implies two basic methods to navigate through the system. The courses provide a hierarchical navigation as in Unix-like file systems. Often more useful is the navigation through the hypergraph of educational objectives: a user can select one or more educational objectives and search for educational units that can be studied based on this knowledge or which provide this knowledge. Moreover, the system also offers the usual string search functions on titles or contents of documents.

Finally, CurryWeb also manages *users*. All documents can be anonymously accessed but any change to the system's contents (e.g., critics and ranking of educational units, inserting new material) can only be done by registered users in order to provide some minimal safety. Registration as a new user is always possible and automatically performed (initially, a random password is sent by email to the new user). Additionally, there is also a super user who can change the rights of individual users which are classified into three levels:

– *Standard users* can insert new documents and update their own documents, write critics and rank other educational units.
– *Administrators* are users who can modify all documents.
– *Super users* can modify everything, i.e., they can modify all documents including user data (e.g., deleting users).

Each document in the system (e.g., educational unit, educational objective, course description) belongs to a user (the *author* of a document). Documents can only be changed by the author or his *co-authors*, i.e., other users who received from the author the right to change this document.

To give an impression of the use of CurryWeb, Fig. 1 shows a snapshot of a selected educational unit. The top frame contains the access to basic functions (login/logout, search, site map, etc.) of the system. The upper left frame is the navigation window showing the course structure or the educational objectives, depending on the navigation mode selectable in the lower left frame. The navigation mode shown in Fig. 1 is the course structure, i.e., the upper left frame shows the tree structure of courses. A user can open or close a course folder



Author: Michael Hanus

The Idea of the CurryWeb

last modified: 02/11/2003

Abstract

This document describes the basic idea of the CurryWeb.

Critics for this educational unit:

Ranking of Educational Unit "The Idea of the CurryWeb"

Your ranking 1

This educational unit has not been rankend yet.

Fig. 1. A snapshot of the CurryWeb interface

by clicking on the corresponding icons. By selecting the "Modify Mode", the visualization of the course structure is extended with icons to insert new courses or educational units in the tree. If the navigation mode is based on educational objectives, the upper left frame contains a list of educational objectives in which the user can select some of them in order to search for appropriate educational units.

The right frame is the content frame where the different documents (educational units, course descriptions, descriptions of educational objectives, user settings, etc.) are shown and can be modified (if this is allowed).

An important feature of this web-based system is the hyperlink structure of all entities. For instance, the name of the author of a document is shown with a link to the author's description, or the prerequisites and objectives of an educational unit are linked to the description of the corresponding educational objective. If the author or co-author of a document is logged into the system, the document view is extended with an edit button which allows to load a web page for changing that document.

The rest of this paper is structured as follows. The next section provides a short overview of the main features of Curry as relevant for this paper. Section 3 sketches the model for web programming in Curry. Section 4 describes the implementation of CurryWeb and highlights the features of Curry exploited to implement this system in a high-level manner. Finally, Section 5 contains our conclusions.

2 Basic Elements of Curry

In this section we review those elements of Curry which are necessary to understand the implementation of CurryWeb. More details about Curry's computation model and a complete description of all language features can be found in [4, 9].

Curry is a modern multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [11], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("f e").

A Curry *program* consists of the definition of functions and data types on which the functions operate. Functions are evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation mode of functions which can be either *flexible* or *rigid.* Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule ("*narrowing*"). Calls to rigid functions are suspended if a demanded argument is uninstantiated ("*residuation*").

Example 1. The following Curry program defines the data types of Boolean values and polymorphic lists (first two lines) and functions for computing the concatenation of lists and the last element of a list:

```
data Bool   = True | False
data List a = []   | a : List a

conc :: [a] -> [a] -> [a]
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last :: [a] -> a
last xs | conc ys [x] =:= xs   = x   where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[1] Since `conc` is flexible,[2] the equation "`conc ys [x] =:= xs`" is solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

In general, functions are defined by (*conditional*) *rules* of the form "$f\, t_1 \ldots t_n$ `|` c `=` e `where` vs `free`" with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in $t_1, \ldots, t_n$. The condition and the `where` parts can be omitted if c and vs are empty, respectively. The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable. A *constraint* is any expression of the built-in type `Success`. Each Curry system provides at least equational constraints of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable patterns. However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [7].

The operational semantics of Curry, precisely described in [4, 9], is based on an optimal evaluation strategy [1] and a conservative extension of lazy functional programming and (concurrent) logic programming. However, the application considered in this paper does not require all these features of Curry. Beyond the standard features of functional languages (e.g., laziness, higher-order functions, monadic I/O), logical variables are essential to describe the structure of dynamic web pages appropriately by exploiting the functional logic programming pattern [2] "locally defined global identifier".

3 Programming Dynamic Web Pages in Curry

This section surveys the model supported in Curry for programming dynamic web pages. This model exploits the functional and logic features of Curry and is available through the library `HTML` which is part of the PAKCS distribution [7]. The ideas of this library and its implementation are described in detail in [5].

If one wants to write a program that generates an HTML document, one must decide about the representation of such documents inside a program. A textual representation (as often used in CGI scripts written in Perl or with the Unix shell) is very poor since it does not avoid certain syntactical errors (e.g., unbalanced parenthesis) in the generated document. Thus, it is better to

[1] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[2] As a default, all functions except for I/O actions and external functions are flexible.

introduce an abstraction layer and model HTML documents as elements of a specific data type together with a wrapper function that is responsible for the correct textual representation of this data type. Since HTML documents have a tree-like structure, they can be represented in functional or logic languages in a straightforward way [3, 10, 12]. For instance, we can define the type of HTML expressions in Curry as follows:

```
data HtmlExp = HtmlText String
             | HtmlStruct String [(String,String)] [HtmlExp]
             | HtmlElem   String [(String,String)]
```

Thus, an HTML expression is either a plain string or a structure consisting of a tag (e.g., `b`,`em`,`h1`,`h2`,...), a list of attributes (name/value pairs), and a list of HTML expressions contained in this structure (because there are a few HTML elements without a closing tag, like `<hr>` or `<br>`, there is also the constructor `HtmElem` to represent these elements).

Since writing HTML documents in the form of this data type might be tedious, the `HTML` library defines several functions as useful abbreviations (`htmlQuote` transforms characters with a special meaning in HTML, like `<`, `>`, `&`, `"`, into their HTML quoted form):

```
htxt   s      = HtmlText (htmlQuote s)       -- plain string
h1     hexps = HtmlStruct "h1"  [] hexps    -- main header
bold   hexps = HtmlStruct "b"   [] hexps    -- bold font
italic hexps = HtmlStruct "i"   [] hexps    -- italic font
verbatim s   = HtmlStruct "pre" [] [htxt s] -- verbatim text
hrule        = HtmlElem "hr" []             -- horizontal rule
...
```

Thus, the following function defines a “Hello World” document consisting of a header and two words in italic and bold font, respectively:

```
hello = [h1 [htxt "Hello World"],
         italic [htxt "Hello"],
         bold [htxt "world!"]]
```

A *dynamic web page* is an HTML document (with header information) that is computed by a program at the time when the page is requested by a client (usually, a web browser). For this purpose, there is a data type

```
data HtmlForm = HtmlForm String [FormParam] [HtmlExp]
```

to represent complete HTML documents, where the first argument to `HtmlForm` is the document’s title, the second argument are some optional parameters (e.g., cookies, style sheets), and the third argument is the document’s content. As before, there is also a useful abbreviation:

```
form title hexps = HtmlForm title [] hexps
```

The intention of a dynamic web page is to represent some information that depends on the environment of the web server (e.g., stored in data bases). Therefore, a dynamic web page has always the type "`IO HtmlForm`", i.e., it is an I/O action[3] that retrieves some information from the environment and produces a web document. Thus, we can define the simplest dynamic web page as follows:

```
helloPage = return (form "Hello" hello)
```

This page can be easily installed on the web server by the command "`makecurrycgi`" of the PAKCS distribution [7] with the name "`helloPage`" as a parameter.

Dynamic web pages become more interesting by processing user input during the generation of a page. For this purpose, HTML provides various input elements (e.g., text fields, text areas, check boxes). A subtle point in HTML programming is the question how the values typed in by the user are transmitted to the program generating the answer page. This is the purpose of the Common Gateway Interface (CGI) but the details are completely hidden by the `HTML` library. The programming model supported by the `HTML` library can be characterized as programming with call-back functions.[4] A web page with user input and buttons for submitting the input to a web server is modeled by attaching an *event handler* to each submit button that is responsible to compute the answer document. In order to access the user input, the event handler is a function from a "CGI environment" (holding the user input) into an I/O action that returns an HTML document. A *CGI environment* is simply a mapping from CGI references, which identify particular input fields, into strings. Thus, each event handler has the following type:

```
type EventHandler = (CgiRef -> String) -> IO HtmlForm
```

What are the elements of type `CgiRef`, i.e., the *CGI references* to identify input fields? In traditional web programming (e.g., raw CGI, Perl, PHP), one uses strings to refer to input elements. However, this has the risk of programming errors due to typos and does not support abstraction facilities for composing documents (see [5] for a more detailed discussion). Here, logical variables are useful. Since it is not necessary to know the concrete representation of a CGI reference, the type `CgiRef` is abstract. Thus, we use logical variables as CGI references. Since each input element has a CGI reference as a parameter, the logical variables of type `CgiRef` are the links to connect the input elements with the use of their contents in the event handlers. For instance, the following program implements a (dangerous!) web page where a client can submit a file name. As a result, the contents of this file (stored on the web server) are shown in the answer document:[5]

---

[3] The I/O concept of Curry is identical to the monadic I/O concept of Haskell [14].

[4] This model has been adapted to Haskell in [13], but, due to the use of a purely functional language, it is less powerful than our model which exploits logical variables.

[5] The predefined right-associative infix operator $f \,\$\, e$ denotes the application of f to the argument e.

```
getFile = return $ form "Question"
          [htxt "Enter local file name:", textfield fileref "",
           button "Get file!" handler]
 where
  fileref free

  handler env = do contents <- readFile (env fileref)
                   return $ form "Answer"
                    [h1 [htxt ("Contents of " ++ env fileref)],
                     verbatim contents]
```

Since the locally defined name `fileref` (of type `CgiRef`) is visible in the right-hand side of the definition of `getFile` as well as in the definition of `handler`, it is not necessary to pass it explicitly to the event handler. Note the simplicity of retrieving values entered into the form: since the event handlers are called with the appropriate CGI environment containing these values, they can easily access these values by applying the environment to the appropriate CGI reference, like `(env fileref)`. This structure of CGI programming is made possible by the functional as well as logic programming features of Curry.

This programming model also supports the implementation of interaction sequences by nesting event handlers or recursive calls to event handlers. Since CGI references are locally defined without any concrete value, one can also compose different forms without name clashes for input elements. These advantages are discussed in [5] in more detail.

4 Implementation

In this section we describe some details of the implementation of CurryWeb and emphasize how the functional logic features of Curry have been exploited in this application.

CurryWeb, as described in Section 1, is completely implemented in Curry using various standard libraries of PAKCS [7], in particular, the `HTML` library introduced in the previous section. The source of the complete implementation is around 8000 lines of code. The implementation is based on the CGI protocol, i.e., it requires only a standard web server configured to execute CGI programs. All data is stored in XML format in files. The access to these files is hidden by access functions in the implementation, i.e., each fundamental data type of the application (e.g., educational unit, educational objective) has functions to store an entry or to read an entry (with a particular key). By changing the implementation of these functions, the file-based information retrieval can be easily replaced by a data base if this becomes necessary for efficiency reasons.[6]

The functionality of each fundamental notion of CurryWeb (educational unit, educational objective, course, user) is implemented in a separate module pro-

[6] We have not used a data base since the current implementation of PAKCS do not offer a data base interface and the hierarchical and semi-structured data format of XML is more appropriate for our application.

viding an abstract data type. Since CGI programs do not run permanently but are started for every request of a client, we separate all data in independent structures. For all structures we use unique identifiers to describe the connections between them. Hence, we get more efficient access to the data relevant for a single request.

Some common functionality is factorized by the notion of a *document*. Each document has an author, a modification date, and some contents. The latter can be a URL (reference to an externally stored document), a list of HTML expressions, or an XML expression (according to some specific format for representing documents). Documents are used at various places. For instance, an educational objective has a document to describe its meaning, a course has a document for the abstract, and an educational unit has two documents: one for the abstract and one for the contents.

Since the implementation of most parts of the system is not difficult (except for the algorithms on the hypergraph to select educational units for a given set of educational objectives), in the following paragraphs we mainly describe how the various features of Curry have helped to implement this system.

4.1 Types

As mentioned in Section 2, Curry is a polymorphically typed language. Types are not only useful to detect many programming errors at compile time, but they are also useful for documentation. The type of a function can be considered as a partial specification of its meaning. In this project, we have exploited this idea by specifying the type signatures of functions of the base modules before implementing them. This was useful to check the consistency of parts of the program immediately when they have been implemented. Moreover, a web-based documentation of these modules can be automatically generated by the tool `CurryDoc` [6]. `CurryDoc` is a documentation generator influenced by the popular tool `javadoc` but applied to Curry in an extended form. In order to apply this tool, the program must be compilable even it is not yet completely implemented. Therefore, we have initially defined all functions with their type signature and a rule with a call to the error function as its right-hand side. For instance, the preliminary code of a function `updateFile` could be as follows:

```
--- An action that updates the contents of a file.
--- @param f - the function to transform the contents
--- @param file - the name of the file
updateFile :: (String -> String) -> String -> IO ()
updateFile f file = error "updateFile: not yet implemented"
```

This is sufficient to generate the on-line documentation (documentation comments start with "`---`") or import it in other modules. Later, the `error` call in the right-hand side is replaced by the actual code.

The polymorphic type system was also useful to implement new higher abstractions for HTML programming. For instance, we have developed generic functions supporting the creation of index pages. They can be wrapped around arbitrary lists of HTML expressions to categorize them by various criteria (e.g., put all items starting with the same letter in one category). Another module implements the visualization and manipulation of arbitrary trees as a CGI script (e.g., used for the course tree in the upper left frame in Fig. 1) according to a mapping of nodes and leaves into HTML expressions.

4.2 Logical Variables

We have already mentioned in Section 3 that logical variables are useful in web programming as links between the input fields and the event handlers processing the web page. Using logical variables, i.e., unknown values, instead of concrete values like strings or numbers improves compositionality: one can easily combine two independent HTML expressions with local input fields, CGI references and event handlers into one web page without the risk of name clashes between different CGI references. This is useful in our application. For instance, an educational unit has a document for the abstract and a document for the contents. Due to the use of logical variables, we can use the same program code for editing these two documents in a single web page for editing educational units. This will be described in more detail in the next section.

4.3 Functional Abstraction

As mentioned above, the document is a central structure that is used at various places in the implementation. Thus, it makes sense to reuse most of the functionality related to documents. For instance, the code for editing documents should be written only once even if it is used in different web pages or twice in one web page (like in educational units which have two separate documents). In the following we explain our solution to reach this goal.

The implementation of a web page for editing a document consists at least of two parts: a description of the edit form (containing various input fields) as an HTML expression and an event handler that stores the data modified by the user in the edit form. Both parts are linked by variables of type `CgiRef` to pass the information from the edit form to the handler. For the sake of modularity, we hide the concrete number and names of these variables by putting them into one data structure (e.g., a list) so that these parts can be expressed as functions of the following type (where `Doc` denotes the data type of documents):

```
editForm    :: Doc -> [CgiRef] -> HtmlExp

editHandler :: Doc -> [CgiRef] -> (CgiRef -> String) -> IO ()
```

The concrete CGI references are specified in the code of `editForm` by pattern matching (and similarly used in the code of `editHandler`):

```
editForm doc [cr1,cr2,...] =
                   ... textfield cr1 ... textfield cr2 ...
```

Note that the answer type of the associated handler is "`IO ()`" rather than "`IO HtmlForm`". This is necessary since one can only associate a single event handler to each submit button of a web page. In order to combine the functionality of different handlers (like `editHandler`) in the handler of a submit button, we execute the individual (sub-)handlers and return a single answer page (see below).

To use the code of `editForm` and `editHandler`, no knowledge about the CGI references used in these functions is required since one can use a single logical variable for this purpose. For instance, a dynamic web page for editing a complete educational unit contains two different calls to `editForm` for the abstract and the contents document of this unit. Thus, the code implementing this page and its handler has the following structure:

```
eUnitEdit ... = ... return $ form "Edit Educational Unit" [
   ... editForm abstract arefs,... editForm contents crefs,...
   button "Submit changes" handler]
 where
   arefs,crefs free

   handler env = do ...
                    editHandler abstract arefs env
                    editHandler contents crefs env
                    ...
                    return $ answerpage
```

The logical variables `arefs` and `crefs` are used to pass the appropriate CGI references from the forms to the corresponding handlers that are called from the handler of this web page. Thanks to the modular structure, the implementation of the document editor can be implemented independently of its use in the editor for educational units (and similarly for the editors for courses, educational objectives, etc).

There are many other places in our application where this technique can be applied (e.g., changing co-authors of a document is a functionality implemented in the user management but used in the document editor). Since the different modules have been implemented by different persons, this technique was important for the independent implementation of these modules.

4.4 Laziness and Logic Programming

Due to the use of the `HTML` library, logical variables are heavily used to refer to input elements, like text fields, check boxes etc. In some situations, the number of these elements depends on the current data. For instance, if one wants to navigate through the educational objectives, the corresponding web page has a check button for each educational objective to select the desired subset. Thus, one needs an arbitrary but fixed number of logical variables as CGI references for these check buttons. For this purpose, we define a function

```
freeVarList = x : freeVarList   where x free
```

that evaluates to an infinite list of free variables.[7] Thanks to the lazy evaluation strategy of Curry, one can deal with infinite data structures as in non-strict functional languages like Haskell. Thus, the expression (`take` *n* `freeVarList`) evaluates to a list of *n* fresh logical variables.

In some situations one can also use another programming technique. The function `zipEqLength` combines two lists of elements of the same length into one list of pairs of corresponding elements:

```
zipEqLength :: [a] -> [b] -> [(a,b)]
zipEqLength [] [] = []
zipEqLength (x:xs) (y:ys) = (x,y) : zipEqLength xs ys
```

Due to the logic programming features of Curry, we can also evaluate this function with an unknown second argument. For instance, the expression `(zipEqLength [1,2,3] l)` instantiates the logical variable `l` to a list `[(1,x1),(2,x2),(3,x3)]` containing pairs with fresh logical variables `x`*i*. This provides a different method to generate CGI references for a list of elements.

Note that the concrete use of such lists of logical variables is usually encapsulated in functions generating HTML documents (with check boxes) and event handlers for processing user selections, as described in Section 4.3.

4.5 Partial Functions

In a larger system dealing with dynamic data, there are a number of situations where a function call might not have a well-defined result. For instance, if we look up a value associated to a key in an association list or tree, the result is undefined if there is no corresponding key. In purely functional languages, a partially defined function might lead to an unintended termination of the program. Therefore, such functions are often made total by using the `Maybe` type that represents undefined results by the constructor `Nothing` and defined results *v* by (`Just` *v*). If we use such functions as arguments in other functions, we have to check the arguments for the occurrence of the constructor `Nothing` before passing the argument. This often leads to a monadic programming style which is still more complicated than a simple application of partially defined functions.

A functional logic language offers an appropriate alternative. Since dealing with failure is a standard feature of such languages, it is not necessary to "totalize" partial functions with the type `Maybe`. Instead, one can leave the definition of functions unchanged and catch undefined expressions at the appropriate points. For doing so, the following function is useful. It evaluates its argument and returns `Nothing` if the evaluation is not successful, or (`Just` *s*) if *s* is the first

[7] Note that `freeVarList` is not a cyclic structure but a function according to the definition of Curry. Since the logical variable `x` is locally defined in the body of `freeVarList` and each application of a program rule is performed with a variant containing fresh variables (as in logic programming), `freeVarList` evaluates to `x1:x2:x3:...` where `x1`,`x2`,`x3`,... are fresh logical variables.

solution of the evaluation (the definition is based on the operator `findall` to encapsulate non-deterministic computations, see [8]):

```
catchNoSolutions :: a -> Maybe a
catchNoSolutions e = let sols = findall (\x -> x =:= e) in
                  if sols==[] then Nothing else (Just (head sols))
```

From our experience, it is a good programming style to avoid the totalization of most functions and use the type `Maybe` (or a special error type) only for I/O actions since a failure of them immediately terminates the program.

5 Conclusions

We have presented CurryWeb, a web-based learning system that is intended to be organized by its users. The learning material is structured in courses and by educational objectives (prerequisites and objectives). The complete system is implemented with the declarative multi-paradigm language Curry. We have emphasized how the various features of Curry supported the development of this complex application. Most parts of the system were implemented by students without prior knowledge to Curry or multi-paradigm programming. Thus, this project has shown that multi-paradigm declarative languages are useful for the high-level implementation of non-trivial web-based systems.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
2. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. 6th Int. Symp. on Functional and Logic Programming*, pp. 67–87. Springer LNCS 2441, 2002.
3. D. Cabeza and M. Hermenegildo. Internet and WWW Programming using Computational Logic Systems. In *Workshop on Logic Programming and the Internet*, 1996. See also `http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`.
4. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. 24th ACM Symp. on Principles of Programming Languages*, pp. 80–93, 1997.
5. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
6. M. Hanus. CurryDoc: A Documentation Tool for Declarative Programs. In *Proc. 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, pp. 225–228. Research Report UDMI/18/2002/RR, University of Udine, 2002.
7. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2003.
8. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.

9. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
10. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, Vol. 10, No. 1, pp. 1–18, 2000.
11. S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. `http://www.haskell.org`, 1999.
12. P. Thiemann. Modelling HTML in Haskell. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 263–277. Springer LNCS 1753, 2000.
13. P. Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In *4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002)*, pp. 192–208. Springer LNCS 2257, 2002.
14. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

Graphical Representations and Infinite Virtual Worlds in a Logic and Functional Programming Course

J. E. Labra Gayo

Department of Computer Science, University of Oviedo
C/ Calvo Sotelo S/N, 3307,
Oviedo, Spain
labra@lsi.uniovi.es

Abstract. The assignment scheme of our *Logic and Functional Programming* course has adopted the generation of graphical representations, quadtrees and octrees to teach the different features of this kind of languages: higher order functions, lazy evaluation, polymorphism, logical variables, constraint satisfaction, etc. The use of standard XML vocabularies: SVG for graphics and X3D for virtual worlds enables to have numerous visualization tools. We consider that the incorporation of these exercises facilitates teaching and improves the motivation of students.

1 Introduction

The *Logic and Functional Programming* course is given as an elective course of 60 hours to third year undergraduate students in Computer Science Engineering at the University of Oviedo. The rest of programming courses is based on imperative languages like Pascal, C, C++ and Java. In this way, our course is the first, and sometimes the only, contact that our students have with declarative languages. Given its elective nature, the course survival depends on the number of registered students. Apart from the intrinsic difficulty of programming courses, the choice of students is conditional on several aspects related with this kind of languages: poor programming environments, lack of graphical and intuitive debugging and tracing systems, lack of libraries or difficulties to link with libraries written in different languages, etc.

With the main goal of increasing students motivation and course scope we have developed the IDEFIX Project [17] since course 2000/2001, which is focused on the development of an Internet based platform for teaching multiparadigm programming languages.

The course devotes 45 hours to theoretical presentations and 15 hours to programming assignments. In this paper, we just describe the programming assignment scheme, which is based on the use of XML vocabularies that include bidimensional graphics in SVG and virtual worlds in X3D. These assignments follow an incremental presentation scheme using the taxonomy of cognitive goals [15]:

firstly, we present a correct program that the students have to compile and execute. Secondly, we ask them to make some modifications to the program so they learn by imitation a basic understanding of program construction. Finally, we ask them to create a new program by themselves.

The paper begins with a brief overview of these XML vocabularies. In the next section we present the first basic exercise consisting of the development a XML generation library. In section 3 we present the next exercise which allows to generate graphical function representations using higher order functions. In section 4 we study the quadtree recursive structure. In section 5 we present octrees and in section 6 we present the manipulation of infinite virtual worlds using lazy evaluation. In section 7 we describe the creation of polymorphic datatypes. Section 8 describes the use of logic programming features (logical variables and non-determinism) and in the next section we solve the quadtree coloring problem using constraints.

Notation. Along the paper, we use Haskell and Prolog syntax. We suppose that the reader is familiar with these languages.

2 XML Vocabularies

XML [5] is becoming a standard for Internet information exchange and representation. One of the first assignments is to develop a small library for XML generation. This library will allow the students to have a common base for the rest of exercises and as it only requires string manipulation, it can serve as a simple familiarization with declarative languages.

The library must contain the following functions:

– empty e as generates an empty element e with attributes as. It could be implemented just as:
– gen e es generates element e with subelements es
– genAs e as es generates element e with attributes as and subelements es

The above functions could be implemented as:

```
vacio :: String → [(String,String)] → String
empty e as = "<" ++ e ++ putAtts as ++ "/>"

gen :: String → String → String
gen e c = genAs e [] c

genAs :: String → [(String,String)] → String → String
genAs e as c = "<" ++ e ++ putAtts as ++ ">" ++
               c ++
               "</" ++ e ++ ">"

putAtts = concat . map f
  where f (v,n) = "␣" ++ v ++ "=\"" ++ n ++ "\""
```

One of the specific XML vocabularies that will be used is SVG [3] which is becoming the main standard for Internet graphics. Also, VRML (Virtual Reality Modelling Language) can be considered the main standard to represent Internet virtual worlds. However, as this standard precedes XML and does not follow its syntax, since year 2000 the Web3D consortium has developed X3D [2], which can be considered as a new VRML adapted to XML plus some new features.

3 Higher order functions: Graphical Representations

The second assignment is the development of graphical representations of functions. We present them the function plotF which stores in a SVG file the graphical representation of a function:

```
plotF :: (Double → Double) → String → IO ()
plotF f fn = writeFile fn (plot f)

plot f = gen "svg" (plotPs ps)
 where
  plotPs        = concat . map mkline
  mkline (x,x')= line (p x) (p x') c
  ps            = zip ls (tail ls)
  ls            = [0..sizeX]
  p x           = (x0+x, sizeY − f x)

line (x,y) (x',y') c = empty "line"
              [("x1",show x) ,("y1",show y),
               ("x2",show x'),("y2",show y')]

sizeX = 500
sizeY = 500
x0 = 10
```

Notice that plotF does Input/Output actions. Traditionally, most of the courses that teach purely declarative languages try to delay the presentation of the IO system. We consider that a more incremental presentation of this system avoids future misunderstandings. Since when delayed it in previous years, a lot of students found strange its introduction in a language that they were told to be pure. The source code of the plotF function also serves as an example that uses the recursive combinators zip, map, foldr, etc. that characterize Haskell.

In this assignment, the proposed exercises use the concept of higher order functions, which is a key concept of functional programming languages. For example, we ask the student to build a function plotMean that writes in a file the graphical representation of two functions and their mean. In figure 1 we show the mean of (\x →10 ∗ **sin** x) and (\x →10 + **sqrt** x).

The code to represent the mean function is as simple as:

```
mean f g = plotF (\x → (f x + g x)/2)
```

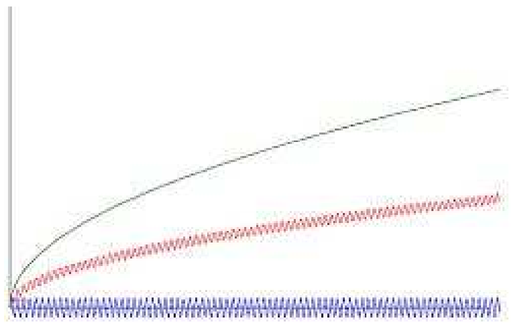


Fig. 1. Graphical representation of 2 functions and their mean

4 Recursive Datatypes: Quadtrees

The next didactic unit is centered on the definition of recursive datatypes and their manipulation. The traditional assignments used the predefined datatype of lists and defined a new datatype of binary trees. Usually, the students had serious difficulties to understand these recursive definitions and are usually not motivated by these exercises [11, 20]. In order to increase their motivation we use the concept of quadtrees [19], which are recursive structures that can represent pictures and have numerous practical applications. In a quadtree, pictures are represented by a unique color or by the division of the picture in four quadrants which are quadtrees. The generalization of quadtrees to three dimensions are called octrees, as they divide the cube in eight parts. These structures are widely used in computer science because they allow to optimize the internal representation of scenes and tridimensional databases for geographical information systems. In Haskell, a quadtree can be represented using the following datatype:

```
data Color = RGB Int Int Int
data QT = B Color
        | D QT QT QT QT
```

An example of a quadtree could be:

```
exQT = D r g g (D r g g r)
  where r = B (RGB 255 0 0)
        g = B (RGB 0 255 0)
```

The above example is represented in figure 2.

Quadtree visualization is made using SVG files which are generated using the functions empty, gen and genAs implemented by students for XML file generation.

```
type Point = (Int,Int)
type Dim   = (Point,Int)
```

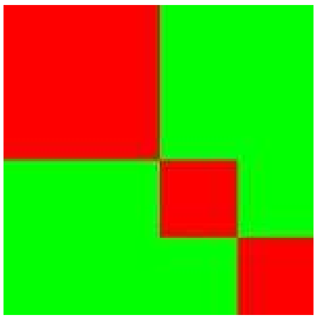


Fig. 2. Example of quadtree

```
wQt :: QT → FileName → IO ()
wQt q fn = writeFile fn (gen "svg" (sh ((0,0),500) q))

sh :: Dim → QT → String
sh ((x,y),d) (B c) = rect (x,y) (x+d+1,y+d+1) c
sh ((x,y),d) (D ul ur dl dr) =
  let d2 = d `div` 2
  in if d <= 0 then ""
     else sh ((x, y), d2)           ul ++
          sh ((x + d2, y), d2)      ur ++
          sh ((x, y + d2), d2)      dl ++
          sh ((x + d2, y + d2), d2) dr

rect :: Point → Point → Color → String
rect (x,y) (x',y') (RGB r g b) =
 empty "rect"
     [("x",show x),("y",show y),
      ("height",show (abs (x - x'))),
      ("width", show (abs (y - y'))),
      ("fill", "rgb(" ++ show r ++ "," ++
                         show g ++ "," ++
                         show b ++ "," )]
```

5 Octrees and Virtual Worlds

An octree is a generalization of a quadtree to three dimensions. The Haskell representation of octrees could be:

```
 data OT = Empty
          | Cube Color
          | Sphere Color
          | D OT OT OT OT OT OT OT OT
```

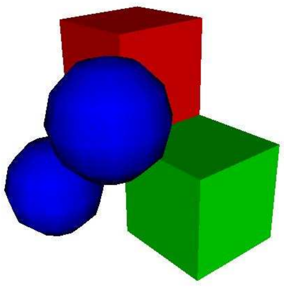


Fig. 3. Example of octree

The above representation declares that an octree can be empty, or it can be a cube, or a sphere or a division of eight octrees. An example of an octree could be:

```
exOT :: OT
exOT = D e s s e r e e g
  where e = Empty
        s = Sphere (RGB 0 0 255)
        r = Cube   (RGB 255 0 0)
        g = Cube   (RGB 0 255 0)
```

We present the students the function wOT ::OT →FileName →**IO** () which stores a representation of an octree in a X3D file.

Figure 3 shows a screen capture of the exOT octree. Although in this paper, we present a printed version, the system generates a 3D virtual model that the students can navigate through.

6 Lazy Evaluation and Infinite Worlds

A nice feature of Haskell is lazy evaluation, which allows the programmer to declare and work with infinite data structures. The students can define infinite octrees like the following:

```
inf :: OT
inf = D inf e s e e e r inf
  where e = Empty
        s = Sphere (RGB 0 0 255)
        r = Cube   (RGB 255 0 0)
```

Figure 4 presents the visualization of the previous octree.

It is also possible to define functions that manipulate infinite octrees. For example, the following function takes an octree as argument and generates a new octree repeating it.

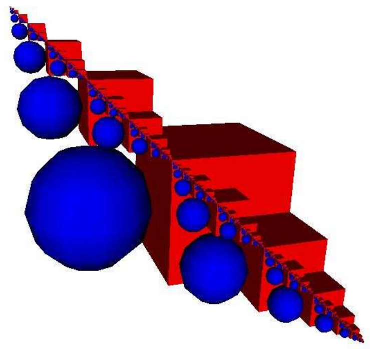


Fig. 4. Example of infinite octree

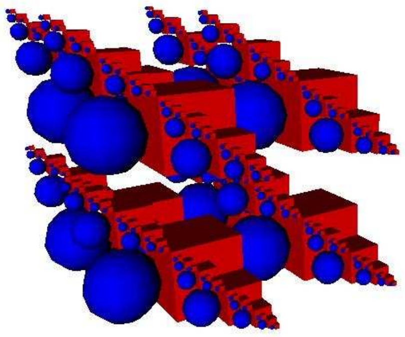


Fig. 5. Repetition of infinite octree

```
repeat :: OT → OT
repeat x = D x x x x x x x x
```

When applying the **repeat** function to the octree inf we obtain the octree that appears in figure 5.

7 Polymorphism: Height Quadtrees

The Haskell type system allows the programmer to declare polymorphic datatypes like lists. Although traditional quadtrees contain color information in each quadrant, it is possible to define a generalization of quadtrees that may contain values of different types. The new declaration could be:

```
data QT a = B a
          | D (QT a) (QT a) (QT a) (QT a)
```

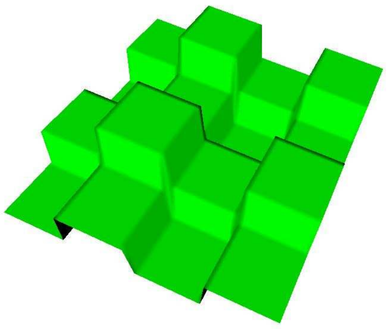


Fig. 6. Heights quadtree

The quadtrees presented in previous sections would be values of type QT Color. This generalization enables the definition of other kinds of quadtrees, like those that contain height information (a value of type **Float**). Those quadtrees can be useful to represent terrain information. For example, the following quadtree:

```
qth :: QT Float
qth = let x = D (B 10) (B 20) (B 0) (B 10)
      in D x x x x
```

is represented in figure 6.

Haskell promotes the use of generic functions like the predefined **foldr** function for lists. A similar function can also be defined for quadtrees.

```
foldQT:: (b → b → b → b → b) →
         (a → b) → QT a → b
foldQT f g (B x) = g x
foldQT f g (D a b c d) = f (foldQT f g a) (foldQT f g b)
                           (foldQT f g c) (foldQT f g d)
```

It is possible to define numerous functions using the foldQT function. For example, to calculate the list of values of a quadtree, we can define:

```
values :: QT a → [a]
values = foldQT (\a b c d → a ++ b ++ c ++ d)
                (\x → [x])
```

The depth of a quadtree can also be defined as:

```
depth :: QT a → Int
depth = foldQT (\a b c d → 1 + maximum [a,b,c,d])
               (\_ → 1)
```

The foldQT belongs to a set of functions that traverse and transform a recursive data structure into a value. These functions are also called catamorphisms and are one of the research topics of generic programming [4].

8 Non-determinism: Coloring Quadtrees

One of the difficulties of our course is the introduction of two paradigms, logic and functional, in a short period of time. We are currently employing two different programming languages, Haskell and Prolog, although we would like to use a logic-functional language like Curry in the future. In order to facilitate the shift between paradigms and languages, we ask the students similar assignments in both parts of the course, so they can observe the main differences. In this way, the first assignments in the logic programming part also work with quadtrees and octrees. However, an important feature of logic programming is the use of logical variables and predicates that admit several solutions obtained by backtracking.

As an example, it is possible to define the predicate col(Xs,Q1,Q2) which is true when Q2 is the quadtree formed by filling the quadtree Q1 with colours from the list Xs.

```
col(Xs,b(_),b(X)):-elem(X,Xs).
col(Xs,d(A,B,C,D),d(E,F,G,H)):-
    col(Xs,A,E),col(Xs,B,F), col(Xs,C,G),col(Xs,D,H).
```

where **elem**(X,Xs) is a predefined predicate that is satisfied when X belongs to the list Xs. Notice that when we ask to fill a quadtree with several colours, we can obtain multiple solutions.

```
?-col([0,1],d(b(_),b(_),b(_),b(_)),V).
V = d(b(0),b(0),b(0),b(0)) ;
V = d(b(0),b(0),b(0),b(1)) ;
V = d(b(0),b(0),b(1),b(0)) ;
. . .
```

The traditional problem to color a map in such a way that no two adjacent regions have the same color can be posed using quadtrees. In figure 7 we present a possible solution to color a quadtree representing a rhombus.

A naive solution using logic programming consists of the generation of all the possible quadtrees and the corresponding test using the following predicate noColor.

```
noColor(b(_)).
noColor(d(A,B,C,D)):-
  noColor(A), noColor(B),noColor(C), noColor(D),
  right(A,Ar), left(B,Bl), diff(Ar,Bl),
  right(C,Cr), left(D,Dl), diff(Cr,Dl),
  down(A,Ad), up(C,Cu), diff(Ad,Cu),
  down(B,Bd), up(D,Du), diff(Bd,Du).

up(b(X),l(X)).
up(d(A,B,_,_),f(X,Y)):-up(A,X), up(B,Y).

down(b(X),l(X)).
down(d(_,_,C,D),f(X,Y)):-down(C,X), down(D,Y).
```

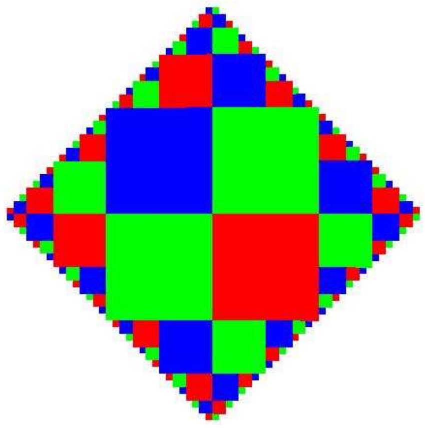


Fig. 7. Solution of the quadtree coloring problem

```
left(b(X),l(X)).
left(d(A,_,C,_),f(X,Y)):- left(A,X), left(C,Y).

right(b(X),l(X)).
right(d(_,B,_,D),f(X,Y)):- right(B,X), right(D,Y).

diff(l(X),l(Y)):-X\=Y.
diff(l(X),f(A,B)):- notElem(X,A), notElem(X,B).
diff(f(A,B),l(X)):- notElem(X,A), notElem(X,B).
diff(f(A,B),f(C,D)):- diff(A,C), diff(B,D).

notElem(X,l(Y)):-X\=Y.
notElem(X,f(A,B)):- notElem(X,A), notElem(X,B).
```

9 Constraints

Declarative languages offer an ideal framework for constraint programming. Although an in-depth presentation of the field is out of our course scope, we considered that a short introduction could be useful to the students, because most of them, will have no more contact with that field in other courses. Following with the quadtrees subject, we can define a solution of the map coloring problem using the logic programming extension with finite domains offered by some implementations. As an example, the following fragment solves the problem using the CLP(FD) syntax implemented in GNU Prolog [7].

```
colC(M,N,b(_),b(X)):- fd_domain(X,M,N).
colC(M,N,d(A,B,C,D),d(E,F,G,H)):-
    col(M,N,A,E), col(M,N,B,F),
    col(M,N,C,G), col(M,N,D,H).
```

The new version of noColor is almost the same changing calls to predicate \= by calls to predicate #\=. However, the constraint programming implementation solves the problem for N=3 in 0.01sg while the naive solution does it in 74.5sg.

10 Related Work

The lack of popularity of declarative languages [21] has motivated the search for practical and attractive applications which highlight the main features of these languages. As an example, P. Hudak's textbook [12] presents an introduction to functional programming using multimedia based examples. That book uses specific libraries to generate and visualize the proposed exercises. Our approach is similar in its goal, but we have adopted standard XML vocabularies to generate the graphical elements. This approach offers several advantages: existence of numerous visualization tools, portability and independence of specific libraries. Furthermore, as a side effect, the students of our course could benefit from the use of these XML technologies in other fields of their future professional activity.

The declarative representations of quadtrees have already been studied in [6, 8, 10, 22]. Recently, C. Okasaki [18] has taken as a starting point the Haskell representation of a quadtree to define an efficient implementation of square mattrices using nested datatypes. The consistency of his representation is maintained thanks to the Haskell type system.

In the imperative field there have been several works that emphasize the use of quadtrees as good examples for programming assignments [14, 13, 16]. However, most of these papers are centered on the image compression application of quadtrees. Several algorithms for quadtree coloring and their application to the schedule of parallel computations are described in [9].

11 Conclusions

In this paper, we propose a programming assignment scheme for our *logic and functional programming* course that intends to favour the graphical visualization of results and to present at the same time the main features of these languages.

Although we have not made a systematic research about the reaction of our students to this scheme, our first impressions are highly positive, with a significant decrease on the abandonment percentage compared to previous courses. Nevertheless, these kind of statements should rigorously be contrasted, checking, for example, that the students that take the new course really solve programming problems better than other students.

The generation of virtual worlds supposed an incentive for our IDEFIX project [17], among the future lines of research we are considering the development of virtual communities like ActiveWorlds [1] where the students can visit a cummunity, create their own worlds and chat with other students.

References

1. Activeworlds web page. http://www.activeworlds.com.
2. Extensible 3d. http://www.web3d.org.
3. Svg – Scalable Vector Graphics. http://www.w3.org/Graphics/SVG/.
4. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming - an introduction. In S. Swierstra, P. Henriques, and Jose N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*. Springer, 1999.
5. Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (1.0). http://www.w3.org/TR/REC-xml, Oct. 2000. 2nd. Edition.
6. F. W. Burton and J. G. Kollias. Functional programming with quadtrees. *IEEE Software*, 6(1):90–97, Jan 1989.
7. P. Codognet and D. Díaz. Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3), Jun 1996.
8. S. Edelman and E. Shapiro. Quadtrees in concurrent prolog. In *Proc. International Conference on Parallel Processing*, pages 544–551, 1985.
9. D. Eppstein, M. W. Bern, and B. Hutchings. Algorithms for coloring quadtrees. *Algorithmica*, 32(1), Jan 2002.
10. J. D. Frens and D. S. Wise. Matrix inversion using quadtrees implemented in gofer. Technical Report 433, Computer Science Department, Indiana University, May 1995.
11. J. Good and P. Brna. Novice difficulties with recursion: Do graphical representations hold the solution? In *European Conference on Artificial Intelligence in Education*, Lisboa, Portugal, Oct 1996.
12. P. Hudak. *The Haskell School of Expression: Learning Functional Programming through multimedie*. Cambridge University Press, 2000.
13. R. Jiménez-Peris, S. Khuri, and M. Patiño-Martínez. Adding breadth to cs1 and cs2 courses through visual and interactive programming projects. *ACM SIGCSE Bulletin*, 31(1), Mar 1999.
14. J. B. Fenwick Jr., C. Norris, and J. Wilkes. Scientific experimentation via the mathing game. *SIGCSE Bulletin*, 34(1), Mar 2002. 33th SIGCSE Technical Symposium on Computer Science Education.
15. J. Kaasbøll. Exploring didactic models for programming. In *Norsk Informatikk-Konferanse*, Høgskolen I Agder, 1998.
16. S. Khuri and H. Hsu. Interactive packages for learning image compression algorithms. *ACM SIGCSE Bulletin*, 32(3), Sep 2000.
17. J. E. Labra, J. M. Moreno, A. M. Fernández, and Sagastegui H. A generic e-learning multiparadigm programming language system: IDEFIX project. In *ACM 34th SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, USA, Feb 2003.
18. Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. *ACM SIGPLAN Notices*, 34(9):28–35, 1999. International Conference on Functional Programming.
19. Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, Jun 1984.
20. J. Segal. Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, (22):385–411, 1995.
21. P. Wadler. Why no one uses functional programming languages? *ACM SIGPLAN Notices*, 33(8):23–27, Aug 1998.
22. D. Wise. Matrix algorithms using quadtrees. Technical Report 357, Computer Science Department, Indiana University, Jun 1992.

Handling quantifiers in Relational Data Mining

David Lorenzo

Computer Science Dept., Univ. Coruña
15071 A Coruña (Spain)
lorenzo@dc.fi.udc.es

Abstract. Relational data mining systems are able to mine regularities from data that is spread over multiple tables in a database. In these systems, both input data and the discovered regularities are represented using the syntax of Prolog. An interesting issue is to investigate whether these methods can be extended to find more expressive regularities where variables can be freely quantified. For this task several issues have to be studied concerning the search space to explore, the evaluation methods and the interestingness measures. From this study we implemented a simple learning system based on the system Claudien. We report on a reduced set of experiments to show the potential of the approach.

1 Introduction

Data Mining is defined as the extraction of non trivial information previously unknown and potentially useful [2]. Typical Data Mining algorithms use an attribute-value representation where each example corresponds to a tuple in a relational database. In order to apply these algorithms to relational databases we need to search for regularities that involve multiple relations. Relational data mining (RDM) systems as Claudien [14], Tertius [6] and Warm [5], among others, apply data mining methods to data that is spread over multiple tables. In these systems, both input data and the discovered regularities are represented using Prolog queries and clauses which have proved to be enough for many applications, however, it would be interesting to open the possibility of mining more expressive patterns. In the following domain about graphs [11], an example is represented by the following facts:

```
connects(a,b). connects(b,a). connects(a,c). connects(c,a).
connects(a,d). connects(d,a). connects(a,e). connects(e,a).
```

It can be seen that for every pair of points such that the first one connects to the second one, the second one also connects the first one. This can be expressed as follows:

$$\forall x \forall y (connects(y,x) \rightarrow connects(x,y))$$

This kind of regularities can be obtained with the current RDM systems, however we can also observe the property that every point of the graph is connected to at least another point in the same graph:

$$\forall x (point(x) \rightarrow \exists y (point(y), connects(x,y)))$$

Current RDM systems cannot find regularities like this since they use the predefined quantification of Prolog clauses and queries. Similarly the possibility of having universally quantified embedded implications [8] allows to express things like "if for some vertex x all adjacents vertices have a property then the vertex x also has this property", etc... Our aim is then to consider the extension of previous RDM methods to discover logical formulas where variables can be freely quantified. This will require to adapt the methods of evaluation and search to the enhanced representation. To reduce the number of formulas to explore we have used an extended version of the declarative language DLAB [14].

This work is organized as follows. Section 2 briefly introduces data mining methods focusing especially on those working over relational data. Section 3 describes a simple learning system based on the system Claudien that allows the explicit use of quantifiers in the mined patterns. In section 4 we report on a reduced set of experiments on some datasets commonly used by RDM systems. Finally, section 5 comments on related work and sketches future work.

2 Relational Data Mining

A fundamental element of any Data Mining method is the definition of a formal language to establish the kind of regularities that we are interested in. Recently Agrawal, Imielinski and Swami [1] have defined a class of regularities called association rules. An association rule is an expression like $W \Rightarrow B$, where W and B are sets of attributes. A relation r satisfies $W \Rightarrow B$ with respect to σ (confidence threshold) and γ (frequency threshold) if at least a fraction γ of the tuples of r satisfy all the attributes of WB and at least a fraction σ of the tuples that satisfy W, satisfy also B. There are very efficient algorithms to find these association rules that minimize the number of readings on the database [10].

Most of these methods are applicable only to data stored in a single relation of a database. However many domains require systems that can find regularities on data spread over multiple tables which cannot reasonably be merged into one table. Most RDM systems work on relational data represented in predicate logic. For instance, consider the following domain of families:

```
person(a).   parent(a,b).   father(a,c).   male(d).
person(b).   parent(a,d).   father(a,d).   male(a).
person(c).   parent(b,c).   mother(b,c).   female(c).
person(d).   parent(b,d).   mother(b,d).   female(b).
                            son(d,a).
                            daughter(c,a).
```

The kind of regularities that these methods can find correspond to Prolog clauses:

```
daughter(X,Y) :- female(X),parent(Y,X).
mother(X,Y);father(X,Y) :- parent(X,Y).
parent(X,Y) :- father(X,Y).
:- father(X,Y),mother(X,Y).
:- male(X),female(X).
:- parent(X,X).
```

Claudien can also discover so-called denials that represent missing patterns in the database, e.g., that a person cannot be parent of himself, etc... Claudien relies on Prolog to evaluate each possible hypothesis. For instance, a clause c holds in the input data if body(c) is true in the data and the atom in the head is also true, hence the truth of a clause can be determined by executing a query `?- body(c), not head(c)` over the input data in a theorem prover like Prolog. If the query succeeds, the clause is false. This works only if the clause is range-restricted, meaning that all variables in the head should also occur in the body. For instance, the rule $daughter(X,Y) \leftarrow female(X), parent(Y,X)$ is true if the following query fails:

```
?- female(X), parent(Y,X), not(daughter(X,Y)).
```

Findings returned by a data mining system must be interesting enough according to some criteria. The interestingness of a formula can be measured through different estimators, the most general ones being frequency and confidence. In systems like Claudien, the frequency of a Prolog clause is defined as the frequency of the query that makes up the body. The confidence of a clause is defined as the frequency of the conclusion divided by the frequency of the body.

Claudien does not employ the well-known two-phased algorithm for induction of association rules used for instance in Apriori [2]. Instead, it employs an algorithm that directly enumerates all the relevant clauses starting from the most general hypothesis, e.g., $daughter(X,Y) \leftarrow$, that is refined or specialized successively by adding literals to the body or by variable substitutions (Fig. 1).

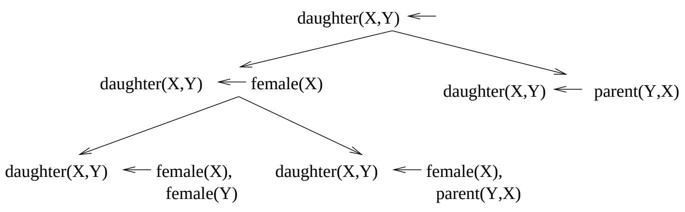

daughter(X,Y) ←
daughter(X,Y) ← female(X)
daughter(X,Y) ← parent(Y,X)
daughter(X,Y) ← female(X), female(Y)
daughter(X,Y) ← female(X), parent(Y,X)

Fig. 1. Search space for the family domain

In practice, however, there are complications when using refinement operators wrt. propositional systems, for instance, there might be infinite chains of refinements and some refinements of a clause do not affect the coverage of the examples [7]. These difficulties can be alleviated by using a declarative (language) bias like DLAB [14] so that the set of clauses to explore is finite. DLAB is a declarative language that allows a user to precisely describe the kind of regularities he is interested in, by using so-called cardinality lists. Cardinality lists are lists of elements enclosed between `[]` that have a cardinality Min-Max associated, so that the list represents every combination of at least Min and at most Max elements. These lists are used to force or avoid certain combinations of elements, to limit the number of literals, etc... For instance:

- All the subsets: Min = 0, Max = length(L)
- All the subsets non-empty: Min = 1, Max = length(L)
- Exclusive OR: Min = Max = 1

where L is a cardinality list. For instance, the template:

```
dlab_template('Q(X,Y) <-- 1-4:[P(X), P(Y), Q(X,Y), Q(Y,X)]').
dlab_variable('P', 1-1, [male, female]).
dlab_variable('Q', 1-1, [parent, father, mother]).
```

specifies that clauses should have one binary predicate in the head, and one to four predicates in the body. The variables P and Q are placeholders for real predicates, so that they can be substituted by the predicates in the corresponding dlab_variable declaration. Several occurrences of a DLAB variable can be substituted by different predicates.

Apart from Prolog clauses other types of patterns considered by current methods are Prolog queries, that are the first-order logic equivalent of an itemset [2], and so-called query extensions [5] or existentially quantified implications, that are the first-order logic equivalent of association rules.

3 Mining patterns in full first-order logic

In this section we consider an extension of the system Claudien to discover regularities that can be described by logical formulas where variables can be freely quantified. We first describe the structure of the input observations and then introduce quantifiers in the language DLAB from which we derive a refinement operator. Finally we revise the evaluation methods used by current RDM systems to work on these more expressive patterns.

3.1 Description of input data

We focus on the learning from interpretations [14] paradigm where input observations consist of a set of models (or databases) of a domain such that every model represents a possible scenario of the domain. For instance, the following domain (Fig. 2) represents a set of planes with geometrical figures related to each other, where each figure has features (or attributes). In this case, each model corresponds to a plane.

Models are sets of ground facts that can be seen as Herbrand interpretations over their associated Herbrand universe, such that those facts not specified, are assumed to be false by default. This logical definition seems appropriate provided

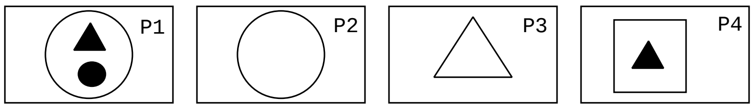


Fig. 2. Planes with geometrical figures

each model represents different worlds of a same universe and not different parts of the same world [6]. The intuition is that the learning system is to look for intra-example relations between objects, but not for inter-example relations, for instance, the pattern "there is a figure that contains other figures" holds in half of the planes. Every model is considered separately, such that it is not necessary to name the objects differently. This is a typical representation used in RDM systems, although some systems work over a single model by looking for properties that are frequent, where frequency is measured as the number of bindings of a particular variable of the query. Plane $P1$ of Fig. 2 is represented as follows:

```
shape(o1,triangle).    in(o1,o2).    color(o1,black).
shape(o2,circle).      in(o3,o2).    color(o3,black).
shape(o3,circle).                    color(o2,white).
```

Models are described extensionally, however, a set of intensional definitions describing general properties of the domain can be used that are applied to the set of models. For instance, the domain of figures is described by primitive predicates circle/1, square/1 and triangle/1, but we can provide definitions like `polygon(X) :- square(X) ; triangle(X)` that completes the description of each model based on the primitive predicates.

3.2 A declarative bias language

To delimit the search space we have used DLAB templates that allow the explicit use of quantifiers so that every variable must be associated to a quantifier. We consider a subset of formulas in first order logic so that a DLAB template is of the form $\forall x.(G(x) \rightarrow A(x))$ or $\exists x.G(x)$ where $G(x)$ is a conjunction of literals or other quantified formulas and $A(x)$ is a single literal or formula. For instance:

```
dlab_template('
ALL X: 1-2:[shape(X,1-1:[triangle,square,circle]),
            color(X,1-1:[black,white])]
        ->
            ANY Y: 2-2:[1-1:[in(Y,X),in(X,Y)],
                        0-2:[shape(Y,1-1:[triangle,square,circle]),
                             color(Y,1-1:[black,white])]]
').
```

We implemented a downward refinement operator ρ that expands a DLAB template by the (recursive) selection of all subsets of L with length within range

$Min \ldots Max$ from each DLAB atom $Min \ldots Max : L$. Elements of lists are considered from left to right to avoid generating the same formula from several paths [14]. The algorithm is basically the same as in Claudien:

```
H:=$\varnothing$
Q:={$\square$}
**while** $Q \neq \varnothing$ **do**
  delete c from Q
  **if** c is valid on O
  **then** add c to H
  **else**
    **for** all $c' \in \rho(c)$ **do**
      add c' to Q
    **endfor**
  **endif**
**endwhile**
```

The algorithm above starts with an empty hypothesis H and a queue Q containing only the most general element $\square$ in the template. It then applies a search process where each element c is deleted from the queue Q and tested for validity in the observations O. If c is valid, it is added to the hypothesis. If c is invalid, its refinements are generated and added to the queue. When the queue is empty, the algorithm is halted and outputs the current hypothesis. The function delete determines the search strategy. The function valid determines when a clause is accepted as part of a solution. Claudien applies diverse filters to prune candidates before refinement, like tautologies (the same literal occurs at both sides of the implication sign) those clauses logically entailed by the background knowledge (redundancy) or by the conjunction of the previously discovered solutions (compactness), among others.

3.3 Evaluation methods

In the learning from interpretations paradigm, a dataset is already split into entities to count[1], so that a dataset satisfies a formula with respect to σ (confidence threshold) and γ (frequency threshold) if at least a fraction γ of the models satisfy the formula and at least a fraction σ of the models where the formula is applicable, satisfy the formula. Formulas of the form $\exists x.G(x)$ are evaluated only by its frequency.

To evaluate the candidate formulas we build a Prolog query for each formula which is evaluated in the input models. Each model is previously asserted in the Prolog interpreter together with the background theory. To evaluate universally

[1] Claudien works also over a single model by looking for properties that are frequent in a sufficient number of objects included in it. Since domains may have multiple kinds of objects, for each query we must specify a key predicate to determine the objects that must be counted. The evaluation must return a number of instantiations of the key variable that satisfy a formula.

quantified embedded implications we use the Prolog built-in predicate $forall/2$ that uses default negation, `forall(P,Q):- not(P,not(Q))`, i.e. every solution to P is a solution to Q. We consider only range-restricted formulas so that the range of a variable is restricted to the elements defined by some other relations in the same formula. For this reason, a literal is implicitly added for each variable of the formula, which represents its type[2]. For instance, the formula

$$\forall X : shape(X, circle) \rightarrow (\exists Y : in(X, Y))$$

is represented by the query:

```
?- forall((figure(X),shape(X,circle)), (figure(Y),in(X,Y))).
```

Variables that appear in P are universally quantified, so that if there is any variable existentially quantified in the first argument of $forall/2$, for instance, the formula:

$$\forall X : (\exists Y : in(Y, X)) \rightarrow shape(X, circle)$$

which is represented by the query:

```
?- forall((figure(X),(figure(Y),in(Y,X))),shape(X,circle))
```

Prolog will try to prove $shape(X, circle)$ for all pairs (X, Y) such that $in(Y, X)$, which is inefficient. One possible way around this problem is to translate implications into queries of the form `?- forall(type,(not(P),!;Q))` so that a single variable appears in the first argument of $forall/2$. Thus, the formula above is evaluated with the following query:

```
?- forall(figure(X),(not(figure(Y),in(Y,X)),! ; shape(X, circle))).
```

It is not clear, however, if the query can be rewritten into an alternative form that improves efficiency.

Once a query has been obtained from each candidate formula, it must be executed in each input model. Let us consider Fig. 2 and the formula "white figures that contain only black figures, are circles":

$$\begin{array}{ll} \forall X : figure(X), & \\ & \forall Y : figure(Y), in(Y, X) \rightarrow color(Y, black) \\ \rightarrow & shape(X, circle) \end{array}$$

The formula holds in $P1$, fails in $P4$ and it is not applicable in those models where, for instance, figures do not contain other figures. However, the Prolog query associated to the formula succeeds in $P2$ and the formula is said to hold trivially in the model [14]. Similarly, the formula fails trivially in $P3$ since the square satisfies trivially the embedded implication. As a consequence, it is necessary to know when a formula holds trivially in a model.

[2] The set of values of the argument found in the data defines the domain of the associated type.

With Prolog clauses, the query `?- body(c)` is enough to determine if a clause c is applicable, however the explicit use of quantifiers makes this query more complex since it must check that there are no universally quantified embedded implications with empty domains. The formula above is applicable to a model provided there is a figure that contains only (and at least one) black figures. If the formula is applicable, a second query is executed to look for a counterexample in a model, e.g., a white rectangle that contains only black figures, since these are the responsible for the falsity of the formula. However, this approach may fail in some cases. Consider an additional plane $P5$ that includes both $P1$ and $P3$. In this case, the Prolog query corresponding to the formula fails in $P5$ and however the formula is applicable and holds non-trivially. As a consequence, the second query must also check non-triviality so that the query succeeds when the formula fails non-trivially.

Similarly, formulas $\exists x.G(x)$ must also hold non-trivially when there are universally quantified embedded implications.

3.4 Pruning methods

The implemented algorithm returns every formula with allowed values for frequency and confidence until the DLAB template has been totally expanded or until the user stops the search explicitly. A downward refinement operator that simply adds literals to the body of a Prolog clause is monotonic wrt. the frequency of the bodies of the clauses, so that refining a clause can only lead to a new body satisfied by fewer models. This allows to prune non-interesting candidates because if a pattern is not frequent then none of its specializations are frequent. However, the explicit use of quantifiers makes that a downward refinement operator does not behave monotonically. Indeed, by adding a literal to an embedded implication can make the formula applicable to more models than before so that the frequency can increase or decrease. Consider again the following formula:

$$\begin{array}{ll} \forall X : figure(X) & \\ & \forall Y : in(Y,X)[, shape(Y, square)] \to color(Y, black) \\ \to & shape(X, circle) \end{array}$$

For models where the formula before adding the literal $shape(Y, square)$ is not applicable (e.g., all circles contain a non-black figure), the resulting formula after adding the literal can be applicable (e.g., there is a circle where the non-black figures inside are not squares).

However, it is clear that the formula above (before adding $shape(Y, square)$) and all its refinements will never be applicable to those models without circles or where no circles contain other figures. As a consequence, we build a third query for each formula to determine if a non-applicable formula is to be refined. For instance, the formula above should not be refined if the following query fails:

$$\begin{array}{l} \exists X : figure(X), \\ \qquad \exists Y : in(Y,X), color(Y, black) \end{array}$$

Note that the query just checks that there are no universally quantified implications with empty domains. Since the frequency of these queries decrease monotonically with the specialization, they can be used to prune the search space. A lower bound on the percentage of models where a formula is applicable must be provided.

We have not considered so far models that are labeled or belong to particular classes. In this case, we want to search for formulas that represent differences between the models of different classes. We have considered only two classes (+ and -) so that we have a positive and a negative frequency. Thus, a formula is a solution if it holds frequently in the positive models and hardly ever in the negative ones (an upper bound on the negative frequency has to be provided).

4 Experimental results

We have implemented a basic system based on the Claudien and carried out some preliminary experiments on some datasets frequently used in the literature of relational data mining [6] to determine the utility of the enhanced language for data mining as well as the impact on performance. Though these domains are toy-problems, they are very similar to real-life problems in e.g. the field of molecular biology where essentially the same representational problems arise [7], i.e., data consists of a set of molecules composed of several atoms related to each others (bonds,...) with specific properties. For domains including positive and negative examples, the negative frequency is set to 0 since we are looking for classification rules. Since the enhanced representation may lead to long waiting times in some domains, the system can be interrupted at any time and returns the solutions found so far.

The number of results obtainted varies greatly depending on the frequency and the confidence specified. We used as default values $\gamma = 40\%$ and $\sigma = 60\%$. We present the results only for the most representative domains focusing mainly on those patterns that previous methods cannot discover. In general, the enhanced expressiveness allows discovering interesting patterns that could not be discovered by previous methods, however, it is critical to specify and use DLAB templates to reduce the size of the search space. Unfortunately, the use of a more expressive language makes that we obtained also many non-interesting patterns that represent knowledge that was previously known or tautologies, so that it may become laborious to extract the interesting patterns from the results. Claudien removes many of these results by using tests of redundancy and compactness.

Bongard-like problems We used Bongard-like problems like that in Fig. 3. In this domain there are three types of figures (circles, squares and triangles) so that some figures can be contained in others. For triangles the direction in which they point (up or down) may be relevant. The possible configurations are classified as positives or negatives. The interest lies in finding a description for positive and negative configurations.

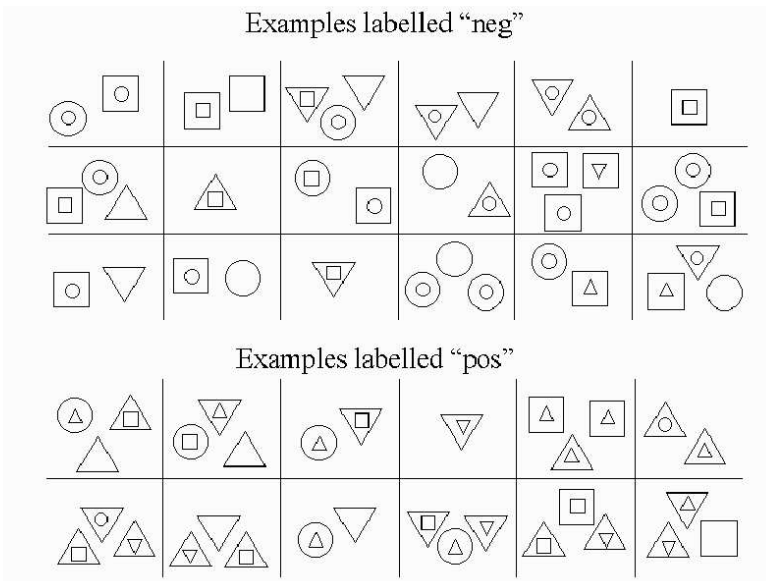


Fig. 3. A Bongard-like problem

We first tried with a reduced dataset [13] shown in Fig. 3. We wrote a DLAB template that includes concepts like "figures that contain no other figures", etc... that can generate up to 193500 candidate hypotheses. Execution terminated after exploring 38569 nodes in 516.06 seconds. No solution was found that holds in 100% of the positive or negative examples. The most frequent pattern is:

$$\begin{aligned} \exists X &: \mathit{shape}(X, \mathit{triangle}), \\ &\quad \exists Y : \mathit{in}(Y, X), \mathit{shape}(Y, \mathit{triangle}) \\ &\quad (f = 75\%) \end{aligned}$$

i.e., "there is a triangle that contains another triangle" which holds in 9 of 12 positive examples and in no negative examples. Other patterns found are much less frequent, for instance, "triangles that contain no figures are contained in other triangle" that holds only in 5 of the 12 positive examples. The rest of the discovered patterns represent either non-interesting regularities like "figures contained in other figure contain no figures" or tautologies.

The full dataset consists of 128 positive examples and 264 negative examples [7]. In this case, execution terminated after exploring 28415 nodes in 2812.97 seconds. Some of the results obtained are:

$$\begin{array}{llll} \forall X: & & \forall X: shape(X, triangle), & \\ \rightarrow & \nexists Y: shape(Y, circle), & & \exists Y: in(Y, X) \\ & \quad in(Y, X) & \rightarrow & \exists Z: in(Z, X), shape(Z, triangle) \\ & (f = 54.68\%, c = 75.26\%) & & (f = 46.875\%, c = 64.51\%) \end{array}$$

with the following meaning: "no figure contains a circle" and "every triangle that contains a figure, contains a triangle".

East West Challenge The second domain consists of distinguishing trains that are travelling east from trains travelling west 4. Each train consists of 2-4 cars where the cars have attributes like shape (rectangular, oval, u-shaped,...), length (long, short), number of wheels (2, 3), type of roof (none, peaked, jagged,...), shape of load (circle, triangle, rectangle,...), and number of loads (1-3). The dataset consists of 20 trains.

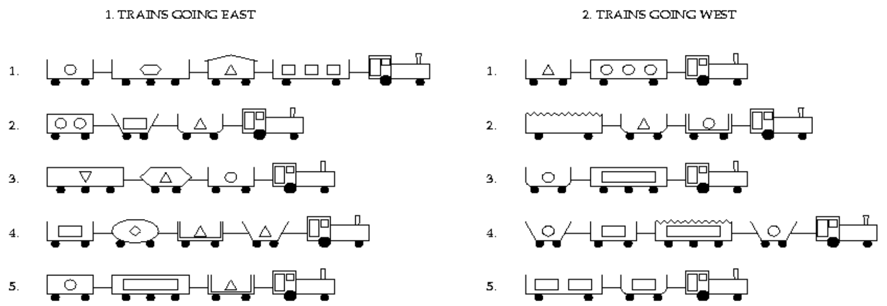


Fig. 4. East-West challenge

As part of the background knowledge, we included a definition for the predicates $geq/2$ and $leq/2$ that apply both to the number of wheels or objects. Even in this simple domain, it seems relevant to be able to mine patterns that refer to cars without objects, cars loading only triangles, etc... However, we found only results very similar to those obtained by other systems like Tertius, which correspond to standard Prolog queries, for instance:

$$\begin{array}{ll} \exists A: roof(A, flat), short(A) & \exists A: roof(A, flat), wheels(A, 2) \\ & \quad \exists B: load(A, B), number(B, 1) \\ \quad (f = 50\%) & (f = 50\%) \end{array}$$

that is, "a train is eastbound if it contains a short closed car" or "if it contains a closed car with two wheels and loading one object".

Problem solving This is an experimental framework that consists of six examples of 'blocks world' pairs representing the situations before and after the application of one of the operators stack, unstack, and transfer [6]. Each example is described by a set of ground facts representing the properties of each block before and after the operation, as well as the identification of the subject of the operation (i.e., the block to be moved) and its destination. The first example, for instance, is represented by the following facts:

```
clear(table,before).       clear(table,after).      subject(x).
clear(x,before).           clear(x,after).          destination(w).
on(x,table,before).        on(x,w,after).           table(table).
on(u,table,before).        on(u,table,after).
on(t,u,before).            on(t,u,after).
    ...                        ...
```

We obtained some interesting results that hold in all the examples, for instance:

$$\begin{array}{ll} \forall A : destination(A) & \forall A : \\ \rightarrow \quad \exists B : on(B, A, after) & \rightarrow \quad \forall B : \nexists C : on(C, B, A) \\ & \qquad \rightarrow \quad clear(B, A) \end{array}$$

The rule on the left gives a post-condition of the action performed, i.e., there is any block on the destination block after each action. The second rule represents a correct definition for the predicate *clear*/2. Note that the definition is valid both for the situation before and after executing an action due to the universal quantification for A.

Families In the domain of families, it is now possible to refer to subgroups "persons with no children" or "parents whose children are all female", and so on, however in the experiments performed on a dataset taken from [12] we found only simpler patterns that could be discovered by previous methods whereas more expressive patterns are not representative in the dataset.

5 Conclusions

An interesting issue is to investigate whether RDM methods can be extended to find regularities in full first-order logic. We have implemented a basic system based on the well-known system Claudien that can find more expressive regularities than the latter where variables can be freely quantified. To restrict the search space, we have used an extended version of the language DLAB to describe search templates. The evaluation of the formulas relies on a Prolog interpreter and negation by failure to allow arbitrary quantifiers in a query. The implemented algorithm performs a complete search over the formulas represented concisely by a DLAB template and evaluates each formula on a set of models that represent possible scenarios of a domain. The explicit use of quantifiers required also to revise the evaluation methods of previous methods.

The most relevant related work is that of Cheng et al. [11] where the authors laid a theoretical foundation for systems to learn prenex conjunctive normal forms (PCNF) with existential variables. They developed a refinement operator defining the substitutions that specialize a PCNF clause and implemented it on a system called PCL. Approaches that learn description logics programs [4] can be also an interesting option for Data Mining.

There are several directions for further research. Perhaps the most obvious is to adapt the two-phased algorithm of Apriori to full first-order logic, as made in the system Warm [5]) for query extensions. It is also needed to develop a more efficient evaluation since e.g. the current one does not exploit overlaps between queries [3]. We plan to apply the implemented system on domains that can benefit from the enhanced expressiveness with respect to Prolog rules, for instance, finding frequent substructures in chemical compounds [5], discovering spatial associations [9], etc..., where previous methods have been applied successfully. We need also to develop advanced interestingness measures similar to those used in the system Tertius [6], for instance, the unusualness of the dependency between the body and the conclusion, etc...

Acknowledgments We would like to acknowledge the anonymous referees for the detailed and constructive suggestions that helped me improve the earlier version of this paper.

References

1. R. Agrawal and T. Imielinski. Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD Conference*, pages 207–216, 1993.
2. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in knowledge discovery and data mining*, pages 307–328. AAAI Press, 1996.
3. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
4. W. W. Cohen and H. Hirsh. Learning the classic description logic: Theoretical and experimental results. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR94)*. Morgan Kaufmann, 1994.
5. L. Dehaspe. *Frequent pattern discovery in first order logic.* PhD thesis, K.U. Leuven, 1998.
6. P. A. Flach and N. Lachiche. Confirmation-guided discovery of first-order rules with Tertius. *Machine Learning*, 42(1/2):61–95, 2001.
7. W. Van Laer and L. De Raedt. How to upgrade propositional learners to first order logic: A case study. In Saso Dzeroski and Nada Lavrac, editors, *Relational Data Mining*. Springer-Verlag, 2001.
8. J.W. Lloyd and R.W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3(1):225–240, 1984.

9. D. Malerba and F.A. Lisi. Discovering associations between spatial objects: An ilp application. In C. Rouveirol and M. Sebag, editors, *Eleventh International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 156–163. Springer-Verlag, 2001.
10. H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192. AAAI Press, 1994.
11. S. Nienhuys-Cheng, W. Van Laer, J. Ramon, and L. De Raedt. Generalizing refinement operators to learn prenex conjunctive normal forms. In *Ninth International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 245–256. Springer-Verlag, 1999.
12. L. Raedt and N. Lavrac. Multiple predicate learning in two ILP settings. *Logic J. of the IGPL*, 4(2):227–254, 1996.
13. L. De Raedt, H. Blockeel, L. Dehaspe, and W. Van Laer. Three companions for data mining in first order logic. In Saso Dzeroski and Nada Lavrac, editors, *Relational Data Mining*, pages 105–139. Springer-Verlag, 2001.
14. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

A Web Oriented System for Equational Solving

Norio Kobayashi[1], Mircea Marin[2], and Tetsuo Ida[3]

[1] Institute of Biological Sciences,
University of Tsukuba, Tsukuba 305-8572, Japan
nori@nori.tv

[2] Johann Radon Institute for Computational and Applied Mathematics,
Austrian Academy of Sciences, A4040 Linz, Austria
mircea.marin@oeaw.ac.at

[3] Institute of Information Sciences and Electronics,
University of Tsukuba, Tsukuba 305-8573, Japan
ida@score.is.tsukuba.ac.jp

Abstract. We describe a collaborative constraint functional logic programming system for web environments. The system is called Web CFLP, and is designed to solve systems of equations in theories presented by higher-order pattern rewrite systems by collaboration among equation solving services located on the web. Web CFLP is an extension of the system Open CFLP described in our previous work in order to deploy the system on the realistic web environment. We will discuss the architecture of Web CFLP and its language.

1 Introduction

The development of the web has determined the need for standard languages and technologies such as Java, XML and CORBA. These tools enable the development of distributed applications which rely on the transparent access and collaboration between the various services deployed on the web. This web approach works well for solving scientific problems described as systems of equations. Usually, such problems can not be solved by a trivial application of a specialized constraint solving method. It is widely accepted that the design of a generic method to solve all possible equational problems is not a reasonable task. Instead, such problems can be effectively solved by a collaboration of solvers which implement specialized methods for equational solving.

Our collaborative equational system Open CFLP [1] relies on such a design approach. Open CFLP was designed to run in an open environment which provides transparent access to the constraint solving services deployed over the web. However, in order to make Open CFLP a true web application, we must address many other design issues, such as session management problems and security problems of communication. In this paper we discuss such an extension of Open CFLP towards a web application, called Web CFLP.

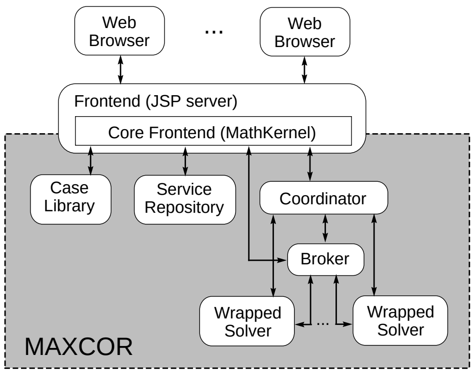


Fig. 1. The architecture of Open CFLP

2 System architecture

Figure 1 shows the architecture of our new system. In our previous paper, we introduced five kinds of CORBA compliant components which communicate via the framework based on CORBA and MathML called MAXCOR [2]:

Frontend which provides a user interface via a *Mathematica* [4] notebook,
Service Repository which looks up a solving service by attributes,
Broker which finds a solver which provides solving service,
Coordinator which manages the collaboration among solvers, and
Object Wrapper which realizes a CORBA compliant solver using existing applications.

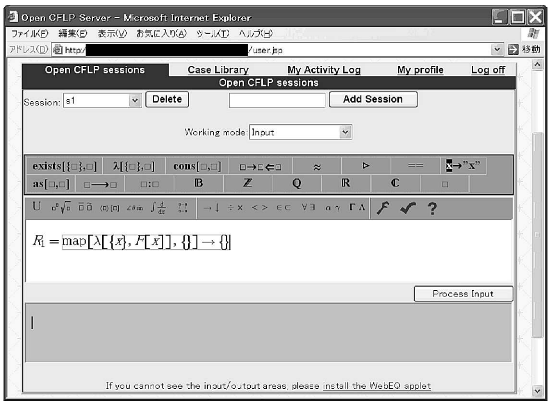
Web CFLP is an extension of Open CFLP with a security mechanism, a redesigned frontend, and a new component called *case library*.

In order to achieve the secure communication through firewalls on MAXCOR, we have introduced SSL/IIOP which is a standard secure protocol of CORBA and the CORBA firewall. As a result, we can deploy our system on web environments with firewalls, and refuse unjust access to our components.

The new frontend is implemented with a JSP (JavaServer Pages) server [5]. Our previous frontend, let's call it *core frontend*, is wrapped by the JSP server. The new frontend

1. provides web pages to browse the case library, the service repository and the broker,
2. provides web pages for input of goals, programs and solver collaboration, and displays the return of the core frontend, and
3. checks the syntax and type correctness of goals and functional logic programs given by the user.

As an example, we show in Figure 2 the web page for item 2. This example page consists of GUI components for input mathematical expressions using Java applet technology. The GUI provides a palette which makes input of mathematical



アドレス(D) http:/[redacted]/user.jsp

Session: s1

Working mode: Input

$R_1 = \mathrm{map}[\lambda[\{x\}, F[x]], \{\}] \to \{\}$

If you cannot see the input/output areas, please install the WebEQ applet

Fig. 2. A web page for input mathematical expressions

expressions easier and creating and displaying MathML documents for communication between the web browser and the JSP server via the HTTP protocol.

The case library is a repository of documents with instructive user sessions. The main objective of this component is sharing problem solving knowledge among users. These documents are written in a well-known format for the web such as XHTML and MathML, and can be created automatically from a history of user's session on a frontend. Further, each document is grouped into a category such as quantum mechanics and dynamics. The case library allows the user to search a document by giving key phrases or a name of category.

3 Web CFLP Sessions

A running Web CFLP environment relies on the existence of solving services registered with the broker and service repository. The developers and implementers of specialized constraint solvers can use Web CFLP to market their expertise as constraint solving services. A minimal set of constraint solving services is provided with the distribution of Web CFLP. It includes a solver for theories described by higher-order functional logic programs in typed λ-calculus, and solvers for linear, polynomial and ordinary differential equations over the domains of reals and complex functions [2]. The functional logic programming solver relies on lazy narrowing calculi developed by the SCORE group [6], whereas the other solvers are extensions to typed λ-calculus of some *Mathematica* solving capabilities.

A Web CFLP session starts with a login step. In this way, we guarantee the authentication of users and the allocation of distinct name spaces to distinct user sessions. Afterwards, the user interaction is a loop of 3 steps:

1. the user types an input expression in the canvas of an applet,
2. the input is submitted to the core frontend, and
3. the core frontend controls the evaluation of the input and displays to the user the result of evaluating it.

Valid inputs are (1) declarations of various identifiers, such as data constructors, user defined symbols, functional logic programs, goals, and collaboratives, and (2) invocations of user-defined constraint solving methods (collaboratives) to a given goal. The syntax is the same as for Open CFLP [1], except the type system. The type system has been extended with support for type classes and type constructors [3]. Type classes specify the range of type variables. Web CFLP provides built-in implementations for a number of useful type classes, e.g., the class of types with equality (`Eq`), the type class of integers (`Integer`), the type constructor `list` for list types, etc. In addition, users can declare their own type classes and type constructors.

The system avoids name clashes between the identifiers of users who use simultaneously the system from different sessions, by maintaining a distinct name space for each user session.

4 Conclusions and Further Work

We have described a web oriented system for constraint functional logic programming, especially our new results from our previous work. Our new system strengthens openness for deployment of the system on the web. In order to provide a general system interface to users we have developed a new frontend using JSP, and users can access our system via a web browser.

As further work, we will extend the applicability of our system by providing object wrappers to more languages and legacy applications with useful equational solving capabilities. The case library will be extended with illustrative examples of how to access these services in concrete situations.

Acknowledgements

Norio Kobayashi had been partially supported by Information-technology Promotion Agency of Japan,The Exploratory Software Project, 2002–2003.

References

1. N. Kobayashi, M. Marin, T. Ida, and Z. Che. Open CFLP: An Open System for Collaborative Constraint Functional Logic Programming, In Proc. of the 11th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP2002), pp. 229–232, Grado, Italy, 2002
2. N. Kobayashi, M. Marin, and T. Ida. "Collaborative Constraint Functional Logic Programming System in an Open Environment." IEICE Transactions on Information and Systems, Vol.E86-D, No.1, pp.63-70, January 2003.
3. T. Nipkow and C. Prehofer. "Type Reconstruction for Type Classes." Journal of Functional Programming, Vol.5, No.2, pp.201–224, 1995.
4. S. Wolfram. The Mathematica Book, 4th Edition, Wolfram Media Inc. Champaign, Illinois, USA, and Cambridge University Press, 1999.
5. JavaServer Pages Specification, Version 1.2, Sun Microsystems, 2001
6. `http://www.score.is.tsukuba.ac.jp`

Author Index