Notice that the object atoms have a metarepresentation as Σ-terms, and that the provability relation *agent* (used in place of the *demo* metapredicate) is defined over a context $\mathcal{C}$. Clause 1 explains the behaviour in presence of a termination request. Clauses 2 and 3 allow object-clonation and context enrichment, while clause 4 allows state transition by deduction step. As we can see, it is very natural in this way to express *LO* program properties. The various agents in a computation can be identified by the number of *agent* predicates present in the goal, while the constraints allow, among others, communication between objects [AP91a].

EXAMPLE 5 *Let us consider the program of ex. 1. The following is an interpretation for such program.*

$method(\{p, a\} \circ\!\!- r \,\&\, (q \parr a)).$
$method(\{q, a, b\} \circ\!\!- \top).$
$method(\{r, b\} \circ\!\!- \top).$

Such an interpretation allows programs to be used by metainterpreter $\mathcal{K}$in any process of inference involving objects such as the one of ex. 1. Notice that heads of clauses are re-interpreted as multisets, in order to allow that object selection via pattern-matching is performed by the constraint solver. □

3 Fixpoint Semantics

It is known [Kow90, HL89] that two kinds of meta-representations are possible in a meta-logical definition of an object-language: the ground representation and the non-ground representation. Ground representation is obtained by making a clear distinction between object-level and meta-level, by representing all object-terms as ground terms. This kind of formalization allows a correct amalgamation between object-language and meta-language and to obtain a correct and complete process of inference for the formulas described at the meta-level. On the other hand, the non-ground representation relaxes the conditions over term representation, by letting object-variables be represented by meta-variables. Although this formalization is in general incorrect, it has been shown [DM92] that for certain kinds of programs a correctness result can be obtained.

The correctenss result we are interested in, however, can be obtained by restricting the set of formulas to consider. The main problem in fact is that the meta-representation of object-variables via meta-variables enables to derive undesired information. Clearly, by pruning such information, we can reach an equivalence result. Practically, the provability relation *agent* we are interested in is considered only over *LO* formulas. The following theorem formalizes the statement.

Theorem 3.1 *Let $\mathcal{C}$ be a multiset of LO ground formulas. Then*

$$P \vdash \mathcal{C} \Leftrightarrow agent(\mathcal{C}) \in T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow \omega$$

Proof We can prove that $P \vdash \mathcal{C} \Leftrightarrow \exists k.agent(\mathcal{C}) \in T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow k$.

First we show that $\exists k.agent(\mathcal{C}) \in T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow k \Rightarrow P \vdash \mathcal{C}$ by induction on k.

The case $k = 1$ is trivial, because there is a valuation v such that $\mathcal{D} \models v(C \uplus \{\top\}) \doteq \mathcal{C}$ and so $\mathcal{C}$ contains the $\top$ const.

Consider the case $k > 1$. Consider the case of $\&$ (the others are quite similar).

- By definition of $T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}}$, there is a valuation v such that

$$\begin{array}{rll} \mathcal{D} \models & v(C \uplus \{x \,\&\, y\}) \doteq \mathcal{C} & \wedge \\ & v(C \uplus \{x\}) \doteq v(C), v(x) & \wedge \\ & v(C \uplus \{y\}) \doteq v(C), v(y) & \end{array}$$

and
$\{agent(v(C), v(x)), agent(v(C), v(y))\} \subseteq T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow (k-1)$.

Then $\mathcal{C}$ can be split in $v(C)$ and $\{v(x) \,\&\, v(y)\}$ and, by inductive hypotesis, there exist proofs for the sequents $P \vdash v(C), v(x)$ and $P \vdash v(C), v(y)$. So we can build a proof tree for $P \vdash \mathcal{C}$ by combining the two branches via the $[\,\&\,]$ rule.

By the converse, we show that $P \vdash \mathcal{C} \Rightarrow \exists k.agent(\mathcal{C}) \in T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow k$ by induction on the lenght n of the proof. Of course the case $n = 1$ is trivial.

Consider the case $n > 1$. We show the case $[\parr]$. The other cases are similar.

- Consider the following proof-structure:

$$\dfrac{\dfrac{\vdots}{P \vdash \mathcal{C}, A, B}}{P \vdash \mathcal{C}, A \parr B}\,[\parr]$$

Clearly, sequent $P \vdash \mathcal{C}, A, B$ has a proof of lenght $n-1$. By inductive hypotesis $agent(\mathcal{C}, A, B) \in T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow k$ for some k. So, by clause 3, $agent(\mathcal{C}, A \parr B) \in T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}} \uparrow (k+1)$.

□

EXAMPLE 6 *We show part of the computation of the fixpoint of $T_{\mathcal{V}_\mathcal{P} \cup \mathcal{K}}$ for the LO program of ex. 1. At level 0 we have the empty set. At level 1, among others, all literals of type $agent(\mathcal{C}, \top)$ are included, where $\mathcal{C}$ is a well-defined ground context. The literals of type $agent(\mathcal{C}, \{q, a, b\})$ and $agent(\mathcal{C}, \{r, b\})$ are included at level 2. Level 3 includes $agent(\mathcal{C}, \{b, q \parr a\})$, and level 4 includes $agent(\mathcal{C}, \{b, r \,\&\, (q \parr a)\})$, from which,at level 5, $agent(\mathcal{C}, \{p, a, b\})$ is derived.*
Finally, level 6 includes $agent(\mathcal{C}, \{p, a \parr b\})$. This shows the forward-chaining reconstruction of the proof-tree in ex. 2. □

4 Conclusions

In this paper we have attempted a structural reconstruction, from a metalogical point of view, of the *LO* semantics. We have introduced a metainterpreter based on a *CLP* instance which defines a constraint solver capable of handling multiset-matching. This is a crucial point for the implementation issues of *LO*. On the other hand, we have considered the re-interpretation, from a forward-chaining point of view, of the *LO* language features. Clearly, if we look at the concurrency semantics, we are not interested to a bottom-up reconstruction of the inference process. Notwithstanding, a declarative interpretation of *LO* formulas has to be defined in order to provide a formal framework in the standard logic programmning context.

The fixpoint semantics reconstruction in this paper is not, however, a simple declarative exercise. As a real semantics for the language, the metainterpreter can be used to analyze the language features. In such a context, for example, the implementation issues on *LO* can be easily identified. Since a concurrent agent is seen as an instance of the metapredicate *agent*, the language can be seen as a simple concurrent logic programming language based on and-parallelism [Sha89]. So, many techniques for analyzing concurrent logic programs can be adapted to be used in the *LO* framework. In such a context for example, variables play the same role as in the concurrent logic programming case, by supporting communication and synchronization. The better expression of inheritance in a multiset context is counterbalanced by the problem of defining a good constraint solver, whilst the communication and parallel-activity implementation can be solved by making use of the parallel implementation of logic languages. So, the better expression of inheritance is a further gap that raises when implementing Object Oriented Programmning within Concurrent Logic Programming.

In our investigation [MT95], we have identified two main ways of expressing object-dynamics in a logic programming framework: the concurrent approach, in which objects are described as processes and state-change is performed via evolution of the goal in the inference process, and the clausal approach, in which objects are logic theories assimilable to deductive databases and state-change is performed via evolution of the extensional database. *LO* fits naturally into the first frame, and the metainterpreter explained in this paper maps it naturally onto the family of concurrent logic programming languages.

References

[ACP92] J. M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*. 1992. To appear.

[ALPT93] J. M. Andreoli, L. Leth, R. Pareschi, and B. Thomsen. True Concurrency Semantics for a Linear Logic Programming Language with Broadcast Communication. In *Proceedings of TAPSOFT'93*, 1993.

[AP91a] J. M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. In *proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPLSA'91)*, pages 212–229, 1991.

[AP91b] J. M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, (3):359–483, 1991.

[AV82] K. R. Apt and M. H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[BG94] P. Bruscoli and A. Guglielmi. Expressiveness of the Abstract Logic Programming Forum in Planning and Concurrency. In Maria Alpuente, Roberto Barbuti, and Isidro Ramos, editors, *Proceedings of the GULP-PRODE'94 Joint Conference on Declarative Programming*, volume 2, pages 221–237, September 1994.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 1994.

[Bro93] A. Brogi. *Program Construction in Computational Logic*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993.

[CC94] Stefania Castellani and Paolo Ciancarini. Comparative Semantics of LO. Technical Report UBLCS-94-7, Laboratory for Computer Science, University of Bologna,Bologna-Italy, April 1994.

[DM92] D. De Schreye and D. Martens. A Sensible Least Herbrand Semantics for Untiped Vanilla Meta-Programming and its Extensions to a Limited Form of Amalgamation. In A. Pettorossi, editor, *Proceedings of the third Zorkshop on Meta-Programming in Logic*, pages 127–141, 1992.

[Gal92] J. Gallier. Constructive Logics. Part II: Linear Logic and Proof Nets. Technical report, Department of Computer and Information Science, University of Pennsylvania, 1992.

[Gir87] J. Y. Girard. Linear Logic. *Theoretical Computer Science*, 50, 1987.

[HL89] P. M. Hill and J. W. Lloyd. Analysis of Metaprograms. In H. D. Abramson and M. H. Rogers, editors, *Metaprogramming in Logic Programming*, pages 23–52. The MIT Press, 1989.

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19,20:69–90, 1994.

[Kow90] R. A. Kowalski. Problems and Promises in Computational Logic. In *Computational Logic,Symposium Proceedings*, pages 1–36. Springer-Verlag, Berlin, 1990.

[KY93] N. Kobayashi and A. Yonezawa. Reasoning on Actions and Change in a Linear Logic Programming. Technical report, Department of Information Science, University of Tokyo, 1993.

[MT95] G. Manco and F. Turini. On the Local State Problem in Object-Oriented Logic Programming. Draft, 1995.

[MTV93] M. Masseron, C. Tollu, and J. Vauzeilles. Generating Plans in Linear Logic I: Actions as Proofs. *Theoretical Computer Science*, 113(2):349–370, 1993.

[Sce94] A. Scedrov. Linear Logic and Computation: A Survey. In H. Schwichtenberg, editor, *Proof and Computation*. Springer Verlag, 1994. To appear.

[Sha89] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[VK76] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

THEORY AND FOUNDATIONS

Computational properties in context-sensitive rewriting *

Salvador Lucas

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, E-46071 Valencia, Spain.
e.mail: slucas@dsic.upv.es

Abstract

Recently we have introduced a novel class of rewrite relations we call *context-sensitive rewriting* that proves useful to mechanize a particular class of inference systems. In this paper, we give a more accurate characterization of some properties of context-sensitive rewriting. We show that several reduction strategies in λ-calculus, namely weak reduction and lazy reduction, can be viewed as an instance of context-sensitive rewriting. Finally, we investigate some basic criteria to ensure confluence and termination of the context-sensitive relation.

Keywords: term rewriting systems, regularity, confluence, termination.

1 Introduction

The theory of Term Rewriting Systems (TRSs) has proven useful to mechanize the execution of functional programs [5] by imposing directionality on the use of equations in proofs [3, 4]. Directed equations, called 'rewrite rules', are used to replace equals by equals, but only in the indicated direction. In this scheme, computation consists of rewriting terms to a normal form, i.e. an expression that cannot be rewritten any further [3]. The notion of sequence of rewrites traditionally defines an operational view of computation. The equational origin of term rewriting techniques leads to consider an expression $t \to s$ as 't *equals* s' rather than 't *becomes* s' [10]. As a consequence, the property of 'being closed under context application' (a particular form of the algebraic property of regularity) is essential in the definition of the rewriting relation.

In [9] we have introduced a novel class of rewrite relations, we call *context-sensitive* rewriting, whose definition is based on regularity. Roughly speaking, given a term t, the subterm of t at the occurrence u is a redex of the context-sensitive rewrite relation iff the symbols in t which label the occurrences above u satisfy a particular condition we call *replacement condition*.

In this paper, we focus on studying computational properties of context-sensitive rewriting. We introduce new properties for the replacement condition, which give a more

*This work has been partially supported by CICYT under grant TIC 92-0793-C02-02.

accurate description of the context-sensitive rewriting process. In particular, we show that the evaluation of the replacement condition can be made compositionally. We also show that context-sensitive rewriting satisfies some desirable properties, such as stability. A pair of interesting examples of application are two common reduction strategies used in λ-calculus [6, 8], namely weak reduction and lazy reduction, which can be viewed as an instance of the context-sensitive rewriting. Finally, we investigate on the confluence and termination of the context-sensitive rewrite relation by means of an appropiate notion of critical pair and we check the termination by defining a specially tailored notion of simplification ordering.

The paper is organized as follows. In Section 2, we briefly recall the technical concepts and results used in the remainder of the paper. In Section 3, we recall the definition and basic properties of the context-sensitive rewrite relation. We extend the notion of regularity of an equivalence relation wrt a given operation to the case of any relation R wrt the operations of an algebra. Then we recall the definition of replacement condition and we introduce some new computational properties. We also show the connections between inference systems and context-sensitive rewriting. To end this section, we illustrate the application of this technique to weak reduction and lazy λ-calculus. Section 4 provides a basic criteria to ensure confluence and termination in context-sensitive rewriting. Finally, Section 5 concludes. More details and missing profs can be found in [9].

2 Preliminaries

Let us first introduce the main notations used in the paper. For full definitions we refer to [4, 8, 14]. Throughout the paper, V denotes a countable infinite set of variables and Σ denotes a set of function symbols {f, g, ...}, each with a fixed arity given by a function $ar : \Sigma \to \mathcal{N}$. By $\mathcal{T}(\Sigma, V)$ and $\mathcal{T}(\Sigma)$ we denote the set of terms and ground terms respectively. Given a term t, $Var(t)$ is the set of variable symbols in t.

Terms are viewed as labelled trees in the usual way. Occurrences $u, v, \ldots$ are represented by chains of positive natural numbers used to address subterms of t, and they are ordered by the prefix ordering: $u \leq v$ if there exists a w such that $u.w = v$. By $u \parallel v$ we denote that the occurrences u, v are disjoint, i.e. $u \not\leq v$ and $v \not\leq u$. $O(t)$ denotes the set of occurrences of a term t. $O_x(t)$ denotes the set of occurrences of a single variable $x \in V$ in t. $t|_u$ is the subterm at occurrence u of t. $t[s]_u$ is the term t with the subterm at the occurrence u replaced with s. We refer to any term C that is the same as t everywhere except below u, i.e. there exists a term s such that $C[s]_u = t$, as the *context* within the replacement occurs. Roughly speaking, a context is a term C with a 'hole' (i.e., a lambda-bound variable) at a distinguished occurrence u.

Let us recall the main concepts and notations about the lattice of first order terms. A substitution is a mapping $\sigma : V \to \mathcal{T}(\Sigma, V)$. The quasi-ordering of subsumption $\leq$ in $\mathcal{T}(\Sigma, V)$, $t \leq t' \Leftrightarrow \exists \sigma\ t' = \sigma(t)$, induces an equivalence $\equiv$ on $\mathcal{T}(\Sigma, V)$: $t \equiv t' \Leftrightarrow t \leq t' \wedge t' \leq t$. The ordering $>$ can be defined as $t > t' \Leftrightarrow t' \leq t \wedge t \not\leq t'$. Let $\hat{\mathcal{T}}$ be the quotient set $\mathcal{T}(\Sigma, V)/\equiv$ completed with a *maximum* element $\top$. $\hat{\mathcal{T}}$ is a complete lattice. We denote the lub of two terms t, t' as $t \sqcup t'$. This term is unique modulo $\equiv$ and can be found (if exists) by an unification algorithm. If this lub exists, we write $t =^? t'$ and say that t and t' are unifiable.

A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\Sigma, V)$, $l \notin V$ and $Var(r) \subseteq Var(l)$. l is said the left-hand side (*lhs*) of the rule and r its right-hand side (*rhs*). A TRS is a pair $\mathcal{R} = (\Sigma, R)$ where R is a set of rewrite rules. For a given TRS $\mathcal{R} = (\Sigma, R)$, a term t rewrites to a term s (at the occurrence u), written $t \to_{\mathcal{R}} s$, if $t|_u = \sigma(l)$ and $s = t[\sigma(r)]_u$, for some rule $l \to r$ in R, occurrence u in $O(t)$ and substitution σ. $\to_{\mathcal{R}}$ is the one-step rewrite relation for $\mathcal{R}$. $\to^*_{\mathcal{R}}$ is the corresponding rewrite relation for $\mathcal{R}$.

$\mathcal{N}^+_k$ is an initial segment $\{1, 2, \ldots k\}$ of the set of positive natural numbers $\mathcal{N}^+$ where $\mathcal{N}^+_0 = \emptyset$ and $\mathcal{N}_k = \mathcal{N}^+_k \cup \{0\}$.

We let R denote a binary relation on A, i.e., $\mathrm{R} \subseteq A \times A$. We will often write x R y for $(x, y) \in$ R. We let R^* denote the reflexive-transitive closure of R.

A homogeneous algebra $\mathcal{A}$ is a pair (A, F) where A is a set, called the carrier, and F is a set of functions $\{f, g, \ldots\}$. We only consider *total functions* in the paper. The mapping $ar : F \to \mathcal{N}$ associates with each function f in F its arity, i.e. $ar(f) = k$ iff f is a mapping from $A^k \to A$. Given a signature Σ, and a set of variables V, we denote as $\mathsf{T}_\Sigma(V) = (\mathcal{T}(\Sigma, V), F_\Sigma)$ the Σ-algebra of terms.

By an Inference System $I = \langle K, \mathcal{V}, \mathcal{P}, \mathsf{A}, \mathsf{R} \rangle$ we mean a collection of the following items: (1) the signature K, (2) an alphabet $\mathcal{V}$ of symbols called I-variables, (3) an alphabet $\mathcal{P}$ of symbols called predicates, each with a fixed arity, (4) a set A of (schemes of) axioms which are well formed formulas (wffs), according to the definition given below, (5) a set R of distinguished relations, the set of wffs called (schemes of) rules of inference. In this way, axioms are obtained from the schemes by instantiating the variables in $\mathcal{V}$ with terms in $\mathcal{T}(K)$. Analogously, the inference rules can be obtained from the schemes of rules by instantiation. In abuse, by a term T of I we mean an element T of $\mathcal{T}(K, \mathcal{V})$. By a wff of I we mean an expression $P(T_1, \ldots, T_m)$ where P is an m-ary predicate symbol and $T_1, \ldots, T_m$ are terms of I. Where we need to distinguish terms of $\mathcal{T}(\Sigma, V)$ and terms of $\mathcal{T}(K, \mathcal{V})$, we call the elements of $\mathcal{T}(K, \mathcal{V})$ I-terms. A proof in an IS is a sequence $(X_1, \ldots, X_n)$ of wffs such that, for each $i \leq n$ either X_i is an axiom or there exist $i_1, \ldots, i_r \in \{1, \ldots, i-1\}$ such that $(X_{i_1}, \ldots, X_{i_r}, X_i)$ stands in one of the relations belonging to R. A wff X is called provable in I (or a theorem of I) iff there exists a proof of X in I, written $\vdash_I X$.

3 Context-sensitive rewriting

3.1 Regularity

First we summarize the main algebraic concepts that support the context-sensitive rewriting. A couple of related definitions and properties can be found in [9].

Definition 3.1 (Regularity) *Let* R *be a relation on* A. *Let* $f : A^k \to A$ *be a function. We say that* R *is* regular *wrt* f *with index* j, $1 \leq j \leq k$ *(we also say that* R *is* j-regular *wrt* f*) iff for all* $a, b, a_1, \ldots, a_k \in A$. a R $b \Rightarrow f(a_1, \ldots, a_{j-1}, a, \ldots, a_k)$ R $f(a_1, \ldots, a_{j-1}, b, \ldots, a_k)$.

The set $m_{\mathrm{R}}(f) \subseteq \mathcal{N}^+_k$ *of all indexes* j *such that* R *is* j*-regular wrt* f *is called the* regularity map *of* R *wrt* f.

The concept of regularity can be isolated of considering a particular relation by only taking the *possible* regular indexes allowed by each function in the underlying algebra. This leads

to the relation-independent concept of *algebraic regularity map*.

Definition 3.2 (Algebraic regularity map) *Let $\mathcal{A} = (A, F)$ be an algebra. A mapping $m : F \to 2^{\mathcal{N}}$ is an* algebraic regularity map *(r-map) for $\mathcal{A}$ iff for all $f \in F$. $m(f) \subseteq \mathcal{N}^{+}_{ar(f)}$. The set of all r-maps of $\mathcal{A}$ is denoted by $M_{\mathcal{A}}$.*

We use an algebraic regularity map to express in a compact mode the regularity property of a relation wrt each function in the considered algebra, according to definition 3.1.

Definition 3.3 (Regularity of a relation with respect to an algebra) *Let $\mathcal{A} = (A, F)$ be an algebra. Let* R *be a relation on A. The r-map m_{R} is called the* regularity map *of* R *wrt $\mathcal{A}$ (or simply the r-map of* R*) iff for all $f \in F$. $m_{\mathrm{R}}(f)$ is the regularity map of* R *wrt f. We also say that* R *is m_{R}-regular wrt $\mathcal{A}$.*

The following example illustrates how to build the regularity map of a given relation with respect to an algebra.

Example 3.4 *Let us consider the relation* R *of 'being multiple of' in the set $\mathcal{Q}^{+}$ of positive rational numbers. We define* R *as follows: $\forall\, a, b \in \mathcal{Q}^{+}.\ a\ \mathrm{R}\ b \Leftrightarrow \exists c \in \mathcal{N}.\ a = b \times c$. Then,*
1. *The regularity map of* R *wrt $(\mathcal{Q}^{+}, \{+\})$ verifies $m_{\mathrm{R}}(+) = \mathcal{O}$.*
2. *The regularity map of* R *wrt $(\mathcal{Q}^{+}, \{/\})$ verifies $m_{\mathrm{R}}(/) = \{1\}$.*
3. *The regularity map of* R *wrt $(\mathcal{Q}^{+}, \{\times\})$ verifies $m_{\mathrm{R}}(\times) = \{1, 2\}$.*

Now we introduce a partial order on r-maps.

Definition 3.5 (Partial order on regularity maps) *Let $\mathcal{A} = (A, F)$ be an algebra. Let $M_{\mathcal{A}}$ be the set of r-maps for $\mathcal{A}$ and let $m, m' \in M_{\mathcal{A}}$. We define a partial order $\sqsubseteq$ on $M_{\mathcal{A}}$ as follows: $m \sqsubseteq m' \Leftrightarrow \forall f \in F.\ m(f) \subseteq m'(f)$.*

It is immediate to see that $(M_{\mathcal{A}}, \sqsubseteq)$ is a complete lattice. The map $m_{\top}$ which assigns to each function $f \in F$ the whole segment $\mathcal{N}_{ar(f)}$ is the top element and the map $m_{\bot}$ which assigns to each function $f \in F$ the empty set $\emptyset$ is the bottom element.

3.2 Basic properties of context-sensitive rewriting

In this section, we briefly recall the basic definition of context-sensitive rewriting given in [9]. The idea behind the concept of context-sensitive rewriting is to impose a syntactic replacement condition which prevents one from performing some reduction steps.
In context-sensitive rewriting we have to restrict term replacements which can be performed by means of the rewrite rules. This is achieved by using a map on the signature we call replacement map.

Definition 3.6 (Replacement map) *Let Σ be a signature. A mapping $\mu : \Sigma \to 2^{\mathcal{N}}$ is a replacement map (or Σ-map) for the signature Σ iff for all $\mathrm{f} \in \Sigma$. $\mu(\mathrm{f}) \subseteq \mathcal{N}^{+}_{ar(\mathrm{f})}$.*

We use the replacement map to give a condition on the occurrences of a given term where the rewrite rules can be applied.

Definition 3.7 (Replacement condition) *Let Σ be a signature and μ be a Σ-map. Let $t \in \mathcal{T}(\Sigma, V)$ be a term. The replacement condition is a relation $\gamma_{\mu,t}$ defined on the set of occurrences $O(t)$ as follows:*

$\gamma_{\mu,t}(\varepsilon)$.

$\gamma_{\mu,\mathrm{f}(t_1,\ldots,t_i,\ldots,t_k)}(i.u) \Leftrightarrow (i \in \mu(\mathrm{f})) \wedge \gamma_{\mu,t_i}(u)$.

We say that the occurrence u of a term t satisfies the replacement condition $\gamma_{\mu,t}$ iff $\gamma_{\mu,t}(u)$. We also say that u is a replacing occurrence of t or that the subterm $t|_u$ is a replacing subterm.

In the following, we write $\gamma_t(u)$ when the replacement map is clear from the context.

Roughly speaking, the replacement condition for a term t wrt μ imposes a restriction based on the function symbols which label the path between the root of t and the subterm at occurrence u. To make clear the meaning of this restriction, we present some basic properties of the replacement condition we have defined.

Proposition 3.8 (Properties of the replacement condition) *Let Σ be a signature, $t, t' \in \mathcal{T}(\Sigma, V)$ and $u, v \in O(t)$. Let μ, μ' be Σ-maps verifying $\mu \sqsubseteq \mu'$. Let us denote $\gamma_{\mu,t}(u)$ as $\gamma_t(u)$ and $\gamma_{\mu',t}(u)$ as $\gamma'_t(u)$ for short. Let $\sigma : V \to \mathcal{T}(\Sigma, V)$ be any substitution. The following statements hold:*

RC1. $u = u_1.u_2 \Rightarrow (\gamma_t(u) \Leftrightarrow (\gamma_t(u_1) \wedge \gamma_{t|_{u_1}}(u_2)))$

RC2. $(u \leq v \wedge \gamma_t(v)) \Rightarrow \gamma_t(u)$

RC3. $(u \leq v \wedge \neg\gamma_t(u)) \Rightarrow \neg\gamma_t(v)$

RC4. $\gamma_t(u) \Rightarrow (u = u_1.u_2 \Rightarrow \gamma_{t|_{u_1}}(u_2))$

RC5. $\gamma_t(u) \Leftrightarrow \gamma_{\sigma(t)}(u)$

RC6. $\gamma_t(u) \Rightarrow \gamma'_t(u)$

RC7. $u \leq v \Rightarrow (\gamma_t(u) \Leftrightarrow \gamma_{t[t']_v}(u))$

RC8. $u \parallel v \Rightarrow (\gamma_t(u) \Leftrightarrow \gamma_{t[t']_v}(u))$

We give some new notation entailed by the concept of replacement condition. We let $O^{\mu}(t)$ denote the set of *replacing* occurrences of a term t i.e. $u \in O^{\mu}(t) \Leftrightarrow u \in O(t) \wedge \gamma_t(u)$. Also we denote by $O^{\mu}_{s}(t)$ the (possibly empty) set of replacing occurrences of a subterm s of t, i.e. $u \in O^{\mu}_{s}(t) \Leftrightarrow u \in O^{\mu}(t) \wedge t|_u = s$. In particular, we use this notation to design the set $O^{\mu}_{x}(t)$ of replacing occurrences of a variable x in t. Denote as $Var^{\mu}(t)$ the set of *replacing variables* of a term t, i.e. $x \in Var^{\mu}(t) \Leftrightarrow O^{\mu}_{x}(t) \neq \emptyset$. Now we introduce the context-sensitive rewriting by means of their one step reduction relation.

Definition 3.9 (One-step context-sensitive rewrite relation) *Let $\mathcal{R} = (\Sigma, R)$ be a TRS. Let μ be a Σ-map. A term t μ-rewrites to a term s, written $t \hookrightarrow_{\mathcal{R}(\mu)} s$, if $t \to_{\mathcal{R}} s$ at the* replacing *occurrence $u \in O^{\mu}(t)$. $\hookrightarrow_{\mathcal{R}(\mu)}$ is the one-step context-sensitive rewrite relation of $\mathcal{R}$ wrt μ. $\hookrightarrow^{*}_{\mathcal{R}(\mu)}$ is the context-sensitive rewrite relation of $\mathcal{R}$ wrt μ.*

In the sequel, we often assume that the TRS $\mathcal{R} = (\Sigma, R)$ and Σ-map μ are fixed, and we drop references to them and write $\hookrightarrow$ instead of $\hookrightarrow_{\mathcal{R}(\mu)}$ when we denote the one-step context-sensitive rewrite relation. Let us now consider a first example of context-sensitive rewriting.

Example 3.10 *Let $\alpha : \mathrm{f}(x) \to x$ be a rule and $t = \mathrm{g}(\mathrm{f}(a), \mathrm{f}(b))$ be a term. a) If $\mu(\mathrm{g}) = \emptyset$ then α cannot be applied to reduce t, because neither the occurrence $u = 1$ or $u = 2$ satisfy the replacement condition wrt μ. b) If $\mu(\mathrm{g}) = \{1\}$, then α can be applied to reduce $\mathrm{f}(a)$ but it cannot be applied to reduce $\mathrm{f}(b)$. c) If $\mu(\mathrm{g}) = \{2\}$, then α can be applied to reduce $\mathrm{f}(b)$ but it cannot be applied to reduce $\mathrm{f}(a)$. d) If $\mu(\mathrm{g}) = \{1, 2\}$, then α can be applied to reduce both $\mathrm{f}(a)$ and $\mathrm{f}(b)$.*

The following proposition illustrates the (one-step) context-sensitive version of the 'being closed under context application' property.

Proposition 3.11 (Restricted context replacement) *Let $C \in \mathcal{T}(\Sigma, V)$. Let $u \in O^{\mu}(C)$. If $t \hookrightarrow s$ then $C[t]_u \hookrightarrow C[s]_u$.*

The context-sensitive rewriting is 'closed under *replacing* context application'. A replacing context $C[\,]_u$ is such that $\gamma_C(u)$ holds. This is what next proposition establishes.

Proposition 3.12 *Let $C \in \mathcal{T}(\Sigma, V)$. Let $u \in O^{\mu}(C)$. If $t \hookrightarrow^n s$ with $n > 0$ then $C[t]_u \hookrightarrow^n C[s]_u$.*

The order $\sqsubseteq$ on algebraic regularity maps can easily be translated to compare *replacement* maps for a given signature Σ. For the sake of simplicity, we will use the same symbol $\sqsubseteq$ for both comparisons and, moreover, to compare r-maps for a given Σ-algebra $\mathcal{A} = (A, F)$ and the corresponding Σ-maps. Therefore there is no confusion in writing $\mu \sqsubseteq \mu'$, $\mu \sqsubseteq m$, $m \sqsubseteq \mu$ or $m \sqsubseteq m'$.

Remark 3.13 *Note that with the Σ-map $\mu_{\top}$ defined as: $\mu_{\top}(\mathrm{f}) = \mathcal{N}^{+}_{ar(\mathrm{f})}$ $\forall\, \mathrm{f} \in \Sigma$, the replacement condition is satisfied for every term t and occurrence u. This is to say that, given a TRS $\mathcal{R}$, the one-step context-sensitive rewrite relation for $\mu_{\top}$ coincides with the standard one-step rewrite relation, i.e. $\hookrightarrow_{\mathcal{R}(\mu_{\top})} = \to_{\mathcal{R}}$.*

The following proposition establishes that, given a TRS, the one-step context-sensitive rewrite relation induced by a replacement map includes the relation induced by a lesser replacement map.

Proposition 3.14 (Monotonicity of $\hookrightarrow$ with respect to $\sqsubseteq$) *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ, μ' be Σ-maps. Then $\mu \sqsubseteq \mu' \Rightarrow \hookrightarrow_{\mathcal{R}(\mu)} \subseteq \hookrightarrow_{\mathcal{R}(\mu')}$.*

Given a TRS and a replacement map, the context-sensitive rewrite relation induced by them is a relation on $\mathcal{T}(\Sigma, V)$ which regularity map wrt the Σ-algebra of terms $\mathsf{T}_{\Sigma}(V)$ is bounded by the replacement map. The following proposition formalizes this point.

Proposition 3.15 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. The r-map m of the relation $\hookrightarrow^{*}_{\mathcal{R}(\mu)}$ satisfies $\mu \sqsubseteq m$.*

Therefore, we think about the replacement map as the *regularity map* of a context-sensitive rewrite relation. Nevertheless, depending on the set of rewrite rules we have in the TRS, regularity can be greater than the expressed by the replacement map. For example, let us consider the TRS $\mathcal{R} = (\{a, b, \mathrm{f}\}, \{a \to b, \mathrm{f}(x, y) \to \mathrm{f}(y, x)\})$ and the Σ-map μ, where

$\mu(a) = \mu(b) = \emptyset$, $\mu(\mathrm{f}) = \{1\}$. It is obvious that, despite the replacement map does not allow rewriting in the second position of f, we get $t \hookrightarrow^{*} s \Rightarrow \mathrm{f}(t, t') \hookrightarrow^{*} \mathrm{f}(s, t')$ (since $1 \in \mu(\mathrm{f})$) and $t \hookrightarrow^{*} s \Rightarrow \mathrm{f}(t', t) \hookrightarrow^{*} \mathrm{f}(t', s)$ (since, by rule $\mathrm{f}(x, y) \to \mathrm{f}(y, x)$, we have $\mathrm{f}(t', t) \hookrightarrow \mathrm{f}(t, t') \hookrightarrow \mathrm{f}(s, t') \hookrightarrow \mathrm{f}(t', s)$) and then $m_{\hookrightarrow^{*}}(\mathrm{f}) = \{1, 2\}$.

Now we prove that context-sensitive rewriting is stable under substitution.

Proposition 3.16 (Stability of $\hookrightarrow$) *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. Let $\sigma : V \to \mathcal{T}(\Sigma, V)$ be a substitution. Then $t \hookrightarrow s \Rightarrow \sigma(t) \hookrightarrow \sigma(s)$.*

only rewriting at certain occurrences

3.3 Context-sensitive rewriting and Inference Systems

In a similar way as a TRS determines an inference system which transition relation is the same that the rewrite relation (taking axioms from instances of the rewrite rules and providing *contextual* inference rules for each k-ary symbol $\mathrm{f} \in \Sigma$ and each argument i, $1 \leq i \leq k$ [8]), we can attach an inference system to a TRS $\mathcal{R} = (\Sigma, R)$ and a Σ-map as follows.

Consider the IS IS$^{\diamond} = \langle K, \mathcal{V}, \mathcal{P}, \mathsf{A}, \mathsf{R} \rangle$ with $K = \Sigma \cup V$. Let $\mathcal{V} = \mathcal{V}_V \uplus \mathcal{V}_I$ be the set of IS$^{\diamond}$-variables, where the set $V = \{x_1, \ldots, x_n\}$ of variables of $\mathcal{R}$ is in a 'one to one' correspondence with the set $\mathcal{V}_V = \{X_1, \ldots, X_n\}$ of variables of IS$^{\diamond}$. Formally, $\nu : V \to \mathcal{V}_V$ and $X_i = \nu(x_i)$. The set of variables in IS$^{\diamond}$ not in $\mathcal{V}_V$ is denoted by $\mathcal{V}_I$. $\mathcal{P}$ contains a single binary predicate symbol $\diamond$ that we will use in as an infix functor. Then, the set Φ of wffs of IS$^{\diamond}$ is $\Phi = \{T \diamond S \mid T, S \in \mathcal{T}(K, \mathcal{V})\} = \{T \diamond S \mid T, S \in \mathcal{T}(\Sigma \cup V, \mathcal{V})\}$.

The set A of (schemes of) axioms of IS$^{\diamond}$ is denoted by $\{\mathsf{a}, \mathsf{b}, \ldots\} \subseteq \Phi$. We write $(\mathsf{a})\dfrac{}{T \diamond S}$ when $\mathsf{a} = T \diamond S \in \mathsf{A}$, as usual. A (scheme of) rule of inference of IS$^{\diamond}$ is a k-ary relation on Φ, with $k > 1$. We write $(\mathsf{r})\dfrac{T_1 \diamond S_1, \ldots, T_{k-1} \diamond S_{k-1}}{T \diamond S}$ when $\mathsf{r} = (T_1 \diamond S_1, \ldots, T_{k-1} \diamond S_{k-1}, T \diamond S)$ is a scheme of rule of inference of IS$^{\diamond}$.

According to the definition of axiom as the instance of an axiom scheme, a rule of inference as the instance of a scheme of rule, and the concept of proof (see Section 2), all theorems are ground wffs (wrt $\mathcal{V}$) $t \diamond s$ with $t, s \in \mathcal{T}(\Sigma, V)$.

The set of all theorems of IS$^{\diamond}$ can be thought of as a relation $\diamond$ on $\mathcal{T}(\Sigma, V)$. We define the (symbolic) relation $\diamond \subseteq \mathcal{T}(\Sigma, V) \times \mathcal{T}(\Sigma, V)$ induced by IS$^{\diamond}$ as follows: $t \diamond s \Leftrightarrow \vdash_{IS^{\diamond}} t \diamond s$.

In the following definition, we denote as T_t the IS$^{\diamond}$-term obtained from a term t by replacing all variables $x \in Var(t)$ in t for their corresponding variables $\nu(x) \in \mathcal{V}_V$.

Definition 3.17 (Basic IS associated to a TRS $\mathcal{R} = (\Sigma, R)$ and a Σ-map) *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. The basic inference system associated to $\mathcal{R}$ and μ, denoted by $I(\mathcal{R}, \mu) = \langle \Sigma \cup V, \mathcal{V}, \{\diamond\}, \mathsf{A}_R, \mathsf{R}_{\mu} \rangle$, is defined as follows:*

1. *$\alpha : l \to r \in R$ iff*
$$(\mathsf{a}_{\alpha})\frac{}{T_l \diamond T_r} \in \mathsf{A}_R$$

2. *For each $\mathrm{f} \in \Sigma$, $k = ar(\mathrm{f})$, $i \in \mu(\mathrm{f})$ and $X_1, \ldots, X_k, Y_i$ distinct variables in $\mathcal{V}$*
$$(\mathrm{r}[\mathrm{f}, i])\frac{X_i \diamond Y_i}{\mathrm{f}(X_1, \ldots, X_i, \ldots, X_k) \diamond \mathrm{f}(X_1, \ldots, Y_i, \ldots, X_k)} \in \mathsf{R}_{\mu}$$

The relation defined by $I(\mathcal{R}, \mu)$ on $\mathcal{T}(\Sigma, V)$ is denoted by $\to_{I(\mathcal{R}, \mu)}$.

We have proved in [9] that $-\!\!\rightarrow_{I(\mathcal{R},\mu)} = \hookrightarrow_{\mathcal{R}(\mu)}$. Note that the above definition formalizes the class of ISs that context-sensitive rewriting mechanizes. Roughly speaking, inference rules need to fit the patterns r[f, i] and axioms must satisfy the conditions $Var(T_r) \subseteq Var(T_l)$ and $T_l \notin \mathcal{V}_V$ which makes it feasible to reformulate them as rewrite rules.

In the next section, we illustrate this idea with some examples.

3.4 Examples and applications

Let us consider the λ-calculus. By analogy with Σ-terms [8], a λ-term $M ::= x \mid \lambda x.M \mid MN$ is represented as a tree by considering each λx as an unary symbol, application as a binary symbol @ and each lambda-variable (in V) as a symbol of arity zero. This can be expressed by means of the signature $\Sigma_\lambda = C_V \cup \Lambda_x \cup \{@\}$, being $\Lambda_x = \{\lambda_x \mid x \in V\}$ and C_V a set of arbitrary constants (one for each lambda-variable). $-\!\!\rightarrow_\beta$, the immediate β-reduction of λ-calculus, can be obtained from the following inference system [6, 8], which we refer to as $IS_\beta = \langle \Sigma_\lambda, \mathcal{V}, \{\triangleright\}, \{red\}, \{r\lambda, r1, r2\}\rangle$ (in spite of the introduction of Σ_λ, we express IS_β in the standard λ-notation):

$$(red)\ \frac{}{(\lambda x.P)Q \triangleright P[x := Q]}$$

We can see that any instance of the axiom scheme (red) can be taken as a rewrite rule $(\lambda x.P)Q \rightarrow P[x := Q]$. In the sequel, R_λ is the set of rules which are instances of the (red) axiom scheme.

$$(r\lambda)\ \frac{M \triangleright M'}{\lambda x.M \triangleright \lambda x.M'} \qquad (r1)\ \frac{M \triangleright M'}{MN \triangleright M'N} \qquad (r2)\ \frac{M \triangleright M'}{NM \triangleright NM'}$$

Note that the *context passing* rules $(r\lambda)$, $(r1)$ and $(r2)$ fit the pattern r[f,i] in Definition 3.17.

Example 3.18 (Weak β-reduction) *Let us consider the weak β-reduction, a restriction on the β-reduction of λ-calculus. For this reduction, an abstraction $\lambda x.M$ is irreducible, even if M contains a redex. This corresponds to drop the rule $(r\lambda)$ in IS_β, thus obtaining $IS_w = \langle \Sigma_\lambda, \mathcal{V}, \{\triangleright\}, \{red\}, \{r1, r2\}\rangle$. We can simulate this behavior with context-sensitive rewriting by considering the following TRS $\mathcal{R} = (\Sigma_\lambda, R_\lambda)$ and the Σ_λ-map μ_w defined as follows: $\mu_w(@) = \{1, 2\}$, $\mu_w(f) = \emptyset$ for all $f \in \Sigma_\lambda - \{@\}$.*

Example 3.19 (Lazy λ-calculus) *Now let us consider the lazy λ-calculus, another restriction on the β-reduction used by the majority of functional languages to avoid the 'eagerness' in evaluating functional applications. To delay the evaluation of an application MN we let only one context passing rule, namely (r1). Therefore we have $IS_z = \langle \Sigma_\lambda, \mathcal{V}, \{\triangleright\}, \{red\}, \{r1\}\rangle$. We can simulate this behavior with context-sensitive rewriting by considering the following TRS $\mathcal{R} = (\Sigma_\lambda, R_\lambda)$ and the Σ_λ-map μ_z defined as follows: $\mu_z(@) = \{1\}$, $\mu_z(f) = \emptyset$ for all $f \in \Sigma_\lambda - \{@\}$.*

The R. Milner's π-calculus [11, 12] is a computational scheme which models concurrency. In [9] we define a TRS and a replacement map whose context sensitive rewriting can be seen as a mechanization of the π-calculus process execution.

4 Confluence and termination

Let us recall that a binary relation R on any set A is *confluent* [4, 7, 8] if for every $b, c \in A$, if a R$^*b$ and a R$^*c$ then there exists $d \in A$ such that b R$^*d$ and c R$^*d$. In a similar way, R is said *locally confluent* if for every $b, c \in A$, if a R b and a R c then there exists $d \in A$ such that b R$^*d$ and c R$^*d$. A relation R is *noetherian* (or *terminating*) [4, 7] iff there is no infinite sequence a_1R $a_2 \cdots a_n$R $a_{n+1} \cdots$. An element $\bar{a}$ is said an R-normal form if there exists no b such that $\bar{a}$ R b. $\bar{a}$ is an R-normal form of a if $\bar{a}$ is an R-normal form and a R$^*\bar{a}$. In a noetherian relation, each element $a \in A$ has a normal form. In a confluent noetherian relation, this normal form is *unique*.

The *diamond lemma* (Newman, [13]) establishes that a locally-confluent terminating relation is confluent. In the sequel, we centre our discussion in analysing confluence (by means of studing the local confluence) and termination in context-sensitive rewriting.

4.1 Confluence

Given a replacement map, the next definition characterizes a class of TRS ensuring that the context-sensitive rewrite relation is locally confluent. This definition demands that any *replacing*-variable occurrence in the *lhs* of a rewrite rule forces any other occurrence of this variable in the *lhs* or *rhs* of this rewrite rule to be a replacing occurrence too.

Definition 4.1 (TRS with homogeneous replacing variables) *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. $\mathcal{R}$ is a TRS with homogeneous replacing variables if for all rule $l \rightarrow r \in R$, when $x \in Var^\mu(l)$ we have $O_x(l) = O^\mu_x(l)$ and $O_x(r) = O^\mu_x(r)$.*

For example, $\mathcal{R}$=({a,b,f,g,h},{g(x,b) $\rightarrow$ f(x,a), f(g(x,y),x) $\rightarrow$ g(x,y), h(g(x,y),x) $\rightarrow$ g(y,y)}) is a TRS with homogeneous replacing variables if μ(g) = {1} and μ(f) = μ(h) = {1, 2}. Now we define a slightly modified version of the notion of critical pair to consider the replacement map.

Definition 4.2 (μ-critical pair) *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. Let $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ be two rewrite rules. Let $u \in O^\mu(l_1)$, $t = l_1|_u \notin V$ and $t =^? l_2$. Let $t' \equiv t \sqcup l_2$, such that $Var(t') \cap Var(l_1) = \emptyset$. We say that the superposition of the two rules determines a μ-critical pair $\langle t_1, t_2\rangle$ defined by $t_1 = \sigma_1(l_1)[\sigma_2(r_2)]_u$ and $t_2 = \sigma_1(r_1)$, where σ_1 is the matching of t' and t ($t' = \sigma_1(t)$) and σ_2 is the matching of t' and l_2 ($t' = \sigma_1(l_2)$).*

Proposition 4.3 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. Let $\sigma.\sigma'$ be two substitutions such that $\sigma(x) \hookrightarrow \sigma'(x)$ and for all $y \neq x$. $\sigma(y) = \sigma'(y)$. Let $t \in \mathcal{T}(\Sigma \cup V)$ and $O_x(t) = \{u_i\}_{i=1}^n$. Let $s_0 = \sigma(t)$ and $s_i = s_{i-1}[\sigma(x)]_{u_i}$, $1 \leq i \leq n$. If $O_x(t) = O^\mu_x(t)$ then $s_i \hookrightarrow^{n-i} \sigma'(t)$, $0 \leq i \leq n$.*

Proposition 4.4 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. Let $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ be two rewrite rules in R, $u \in O^\mu(l_1)$, $t = l_1|_u \notin V$, and there exist $\sigma_1 : Var(t) \rightarrow \mathcal{T}(\Sigma, V)$ and $\sigma_2 : Var(l_2) \rightarrow \mathcal{T}(\Sigma, V)$ where $\sigma_1(t) = \sigma_2(l_2)$. Then there exists a μ-critical pair $\langle t_1, t_2\rangle$ of $\mathcal{R}$ and a substitution θ such that $\sigma_1(l_1)[\sigma_2(r_2)]_u = \theta(t_1)$ and $\sigma_1(r_1) = \theta(t_2)$.*

In the following proposition, we write $t \downarrow t'$ to denote the context-sensitive joinability, i.e. $t \downarrow t'$ if there exists s such that $t \hookrightarrow^* s$ and $t' \hookrightarrow^* s$.

Proposition 4.5 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with homogeneous replacing variables. Let μ be a Σ-map. $\hookrightarrow$ is locally confluent iff for every μ-critical pair $\langle t_1, t_2 \rangle$ we have $t_1 \downarrow t_2$.*

The following theorem allows us to devise an effective way to check confluence in a terminating TRS with homogeneous replacing variables.

Theorem 4.6 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with homogeneous replacing variables and μ be a Σ-map. If $\hookrightarrow$ is noetherian, let us denote by $\bar{t}$ an arbitrary $\hookrightarrow$-normal form of $t \in \mathcal{T}(\Sigma, V)$. Then $\hookrightarrow$ is confluent iff for every μ-critical pair $\langle t_1, t_2 \rangle$ of R we have $\bar{t_1} = \bar{t_2}$.*

Given a TRS with homogeneous replacing variables, this theorem gives us an effective way of testing the confluence of $\hookrightarrow$ provided that we have only a finite number of μ-critical pairs $\langle t_1, t_2 \rangle$. Since $\hookrightarrow$ is noetherian, we will get $\bar{t_1}$ and $\bar{t_2}$ in a finite number of rewritings and then we can check whether $\bar{t_1} = \bar{t_2}$ or not.

4.2 Termination

We can exploit the fact that the 'closed under context aplication' rewrite relation $\rightarrow^*$ is a particular case of context-sensitive rewriting whose replacement map is $\mu_\top$, to give an immediate result that allows us to apply any termination criteria of standard rewriting.

Proposition 4.7 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS. Let μ, μ' be two Σ-maps such that $\mu \sqsubseteq \mu'$. If $\hookrightarrow_{\mathcal{R}(\mu')}$ is noetherian then $\hookrightarrow_{\mathcal{R}(\mu)}$ is noetherian.*

Now we consider the Remark 3.13 to show that any criteria establishing the noetherianity of the rewrite relation for a TRS also establishes the noetherianity of any context-sensitive rewriting we can define from this TRS by giving a replacement map.

Proposition 4.8 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. If $\rightarrow_{\mathcal{R}}$ is noetherian, then $\hookrightarrow_{\mathcal{R}(\mu)}$ is noetherian.*

We are now ready to express a more general condition ensuring termination of a TRS. The following proposition modifies the Manna and Ness' theorem (see [1]), to deal with context-sensitive rewrite relations.

Proposition 4.9 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and μ be a Σ-map. The TRS $\mathcal{R}$ terminates if there exists a μ-regular noetherian ordering $>$ over $\mathcal{T}(\Sigma, V)$ such that $\sigma(l) > \sigma(r)$ for every rule $l \rightarrow r \in R$ and substitution $\sigma : V \rightarrow \mathcal{T}(\Sigma, V)$.*

The above proposition is very strict: it requires a strict order $>$ to be noetherian over the whole set of terms. In fact, it suffices if it is noetherian 'along the reductions' [8]. The existence of an infinitely decreasing sequence of terms has no importance if the sequence is not a reduction. It is much more difficult to prove that an order is noetherian along a chain of reductions. This proof normally depends on the rewriting system being studied. In standard rewriting, there is a class of orders which are necessarily noetherian along reduction chains. They are the simplification orders introduced by N. Dershowitz [1, 2].

From Proposition 4.8, we can directly use any simplification ordering $>$ (for example the recursive path ordering $>^*$ in [1, 2]) to check termination of the context-sensitive

rewriting induced by a TRS and a replacement map by just considering the rules in the TRS (without taking into account the replacement map). We use the simplification ordering to compare the *lhs* l and the *rhs* r of any rule $l \rightarrow r$ in the TRS. We only need to prove that $\sigma(l) > \sigma(r)$ for all rules in the TRS and substitutions σ. If this test holds, then Proposition 4.9 guarantees that any context-sensitive rewriting will terminate.

But we can consider many TRSs whose context-sensitive rewrite relation terminates for a given replacement map, despite the simplification-ordering test does not succeed.

Example 4.10 *Let us consider the TRS, $\mathcal{R} = (\{\mathrm{f}, \mathrm{g}\}, \{\mathrm{f}(\mathrm{g}(x), y) \rightarrow \mathrm{f}(x, \mathrm{f}(\mathrm{g}(x), y))\})$. It is easy to see that $\rightarrow_{\mathcal{R}}$ is not terminating. If we take μ such that $\mu(\mathrm{f}) = \{1\}$, then it is easy to verify that any μ-rewriting on $\mathcal{R}$ will eventually terminate.*

We need to consider the replacement map to take a more accurate decision about termination of context-sensitive rewriting. It is easy to slightly modify the requirements that a simplification ordering must satisfy to accomodate the replacement map. Since we only consider signatures with symbols of fixed arity, we give the following definition.

Definition 4.11 (μ-simplification ordering) *A transitive and irreflexive relation $>_\mu$ over $\mathcal{T}(\Sigma, V)$ is a μ-simplification ordering if it posseses the following two properties. For all $\mathrm{f} \in \Sigma$ and for all $i \in \mu(\mathrm{f})$*

1. $t >_\mu s \Rightarrow \mathrm{f}(t_1, \ldots, t_{i-1}, t, \ldots, t_k) >_\mu \mathrm{f}(t_1, \ldots, t_{i-1}, s, \ldots, t_k)$ *(μ-replacement)*
2. $\mathrm{f}(t_1, \ldots, t_{i-1}, t, \ldots, t_k) >_\mu t$ *($\mu - subterm$)*

The usefulness of this orderings is established in the following theorem.

Theorem 4.12 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS. Let μ be a Σ-map. The TRS $\mathcal{R}$ terminates if there exists a μ-simplification ordering $>_\mu$ over $\mathcal{T}(\Sigma, V)$ such that, for every rule $l \rightarrow r \in R$ and substitution $\sigma : V \rightarrow \mathcal{T}(\Sigma, V)$, we have $\sigma(l) >_\mu \sigma(r)$.*

From the example 4.10 we can see that, when comparing the *lhs* and *rhs* of the rewrite rule $l \rightarrow r$ in order to determine the termination of the TRS, we only need to take into account the *replacing* arguments (given by μ) occurring in l and r. Roughly speaking, since $\mu(\mathrm{f}) = \{1\}$ we would find $l >_\mu r$, i.e. $\mathrm{f}(\mathrm{g}(x), y) >_\mu \mathrm{f}(x, \mathrm{f}(\mathrm{g}(x), y))$ if we forget the second argument both in $\mathrm{f}(\mathrm{g}(x), y)$ and $\mathrm{f}(x, \mathrm{f}(\mathrm{g}(x), y))$. This is similar to check $\mathrm{f}(\mathrm{g}(x), \Box) > \mathrm{f}(x, \Box)$ and consider that $\Box$ is lesser than any other term according to some suitable simplification ordering (for example a recursive path ordering). By lack of space, we let a more accurate development of this idea to a forthcoming work.

5 Conclusions and further research

We have introduced a more accurate description of the context-sensitive rewriting process by extending the properties related to the replacement condition. This property determines the ability of μ-rewriting at an occurrence u given a replacement map μ. We have examined several computational properties of the context-sensitive relations and we have proven that they are stable under substitution and closed under replacing context application. We have connected the several context-sensitive rewrite relations we can define from a given TRS $\mathcal{R} = (\Sigma, R)$ and many Σ-maps which are related by the order $\sqsubseteq$ in the

lattice of Σ-maps. All of this allows us to give some basic conditions that a TRS must satisfy for the induced confluent context-sensitive rewrite relation to be confluent. We have also adapted some classical criteria to ensure termination of rewriting to context-sensitive rewriting.

In the future we plan to give a deeper treatment of confluence and termination, by defining syntactic criteria ensuring them. In particular, we are working in the definition of special recursive path orderings (rpo) that consider the replacement map when testing termination as we mentioned before.

6 Acknowledgements

I would like to thank María Alpuente and Germán Vidal for their encouragement and support. Also, I am grateful to Asunción Casanova, Javier Oliver and Javier Piris for their helpful comments and corrections.

References

[1] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science* 17:279-301, 1982.

[2] N. Dershowitz. Termination of rewriting. *Journal on Symbolic Computation*, 3:69-115, 1987.

[3] N. Dershowitz. A Taste of Rewrite Systems. In P.E. Lauer, editor, *Proc of Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 199-228, Berlin, 1993.

[4] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243-320. Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990.

[5] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification. volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[6] J.R. Hindley and J.P. Seldin. Introduction to Combinators and λ-Calculus. Cambridge University Press, 1986.

[7] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM* 27:797-821, 1980.

[8] R. Lalement. Computation as Logic. Masson-Prentice Hall International, 1993.

[9] S. Lucas and J. Oliver. Context-sensitive rewriting. Technical report DSIC-II/23/94.

[10] J. Meseguer. Conditional Rewriting Logic as an Unified Model of Concurrency. *Journal of Theoretical Computer Science*, 96:73-155, 1992.

[11] R. Milner. The polyadic π-calculus: A tutorial. In F.L. Brauer, W Bauer and H Schwintenberg, editors, *Logic and Algebra of Specifications*, Springer-Verlag, Berlin, 1993.

[12] R. Milner, J. Parrow and D. Walker. A Calculus for Mobile Processes, Part I and Part II. *Information and Computation*, 100:1-77, 1992.

[13] M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Ann. Math.*, 43:223-243, 1942.

[14] R. Smullyan. Theory of Formal Systems. Princeton University Press, N.J., 1971. Revised Edition.

Minimal Set Unification *

Puri Arenas-Sánchez
Dpto. Informática y Automática
Fac. C.C. Matemáticas, (U.C.M.)
Avda. Complutense s/n, 28040 MADRID.
e-mail: puri@dia.ucm.es

Agostino Dovier
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56100 PISA.
e-mail: dovier@di.unipi.it

Abstract

A unification algorithm is said to be minimal for a unification problem if it generates exactly the minimal complete set of unifiers, without instances, without repetitions. Aim of this paper is to describe a new set unification algorithm which is minimal for a significant set of sample problems that can be used as benchmarks for testing any set unification algorithm.
Keywords: Logic Programming with Sets, CLP, Unification.

1 Introduction

The drawing up of many papers concerning *Constraint Logic Programming with sets* (see e.g. [5, 6, 10]) has pointed out that the complexity of the (NP complete–see e.g. [3]) *set unification problem* is the real bottleneck of any attempt to extend Logic Programming with set entities.

The loss of the *uniqueness of the most general unifier* property forces any set unification algorithm to return a *complete* set of unifiers (i.e., as it will be explained in Sect. 2, a set of unifiers which covers all possible solutions for the problem at hand) for any satisfiable input. If the algorithm returns such unifiers all at once (as a disjunction), then the number of computed answers grows exponentially. The non-determinism lying inside any Logic Programming interpreter suggests the use of a non-deterministic unification algorithm which returns exactly one unifier for each non-deterministic branch: in this way any non-deterministic computation can run without undergoing the unpleasant effects of the non uniqueness of the most general unifier.

In [2] it has been shown that if a representation for sets comprising also a constant symbol for the set universe and the set-minus operator is adopted, the *unique*

*P. Arenas-Sánchez is partially supported by the Spanish National Project TIC92–0793–C02–01 "PDR"and the Esprit BRA Working Group Nr. 6028 "CCL". The work is partially supported by C.N.R. grant 94.00472.CT12, "Logic Programming with Sets".

unification theorem can be recovered. Nevertheless, nested sets are not allowed, and the answer to a unification problem contains a large amount of information, becoming scarcely readable.

In [10] any set unification problem is delayed until it is transformed into a simple ground 'test'. This improves efficiency, however, if the two terms do not become ground, obscure answers such as $\{X_1, f(X_1, X_3)\} = \{Y_1, f(\{Y_3\}, X_1), X_2\}$ are returned.

There are different ways to represent a finite set. Among them, the *union of singletons* representation which depicts $\{s_1, \ldots, s_m\}$ as $\{s_1\} \cup \cdots \cup \{s_m\}$, and the *list* representation which uses the term $\{s_1 \mid \{s_2 \mid \{\cdots \{s_m \mid \{\}\} \cdots\}\}\}$ for denoting the same set. The former representation (which is associated with an *ACI* equational theory–cf. [11]) is more expressive than the latter (which is associated to the equational theory described in Sect. 2). For instance, the problem $X_1 \cup \cdots \cup X_m = \{a_1\} \cup \cdots \cup \{a_n\}$, where X_i's are pairwise distinct variables and a_j's are pairwise distinct constant symbols, admits $(2^m - 1)^n$ independent solutions. Since the semantics of $\{t \mid s\}$ is $s \cup \{t\}$, if $m > 1$ such problem cannot be expressed by a list representation.

Since the minimum cardinality of a complete set of unifiers expressible with the list representation is itself conspicuous (cf. Sect. 3), we prefer to deal with such representation, avoiding further problems that the "union of singletons" approach would open. The same choice has been performed in [5, 10].

In this paper we present a new set unification algorithm, minimal for a significant set of sample problems. Such problems can be used for testing any set unification algorithm due to their simplicity (which reflects into a simplification of the analysis) and the fact that they maximize the number of solutions for unification problems of given size (the presence of distinct variables as elements of the sets to be unified guarantees the maximum number of solutions). A deep combinatorial study of such problems along with the recursive functions computing the cardinality of their minimal complete set of most general unifiers can be found in [1].

The aim of getting minimal algorithms for set unification has already been treated in literature. For instance in [12] three set unification algorithms are proposed. The most efficient seems to be the third one, however, comparing its results with our minimality study, it is possible to conclude that such algorithm is not minimal for problems (6), (7), and (8) described in Sect. 3. In [3], the presented naive set unification algorithm (based on [9]) has a minimal behavior only for the first of such benchmarks.

The paper is organized as follows: in Sect. 2 we comment briefly on some preliminary concepts needed in the rest of the paper. Section 3 presents eight sample unification problems along with the tables reporting their minimal number of solutions. In Sect. 4 the *new set unification algorithm* SUA is presented, proving its termination and minimality for all sample problems suggested; Some conclusions are finally drawn up in Sec. 5.

2 Preliminaries

We will make use of standard *CLP* (see e.g. [8]) and *unification theory* (see e.g. [11]) notations. Given a signature Σ (a set of functional symbols along with their arities) and a denumerable set of variables $\mathcal{V}$, $\tau(\Sigma \cup \mathcal{V})$ and $\tau(\Sigma)$ will denote the sets of terms and ground terms, respectively. We require the signature Σ contains (at least) the constant symbol $\{\}$, representing the *empty set*, and the binary symbol $\{\cdot \mid \cdot\}$, used as *set constructor symbol*; the intuitive semantics of $\{t \mid s\}$ is $\{t\} \cup s$. Similarly to lists in PROLOG, the term $\{a \mid \{b \mid \{c \mid \{\}\}\}\}$ will be denoted simply as $\{a, b, c\}$.

The following two equational axioms

$$(A_b) \quad \{X, X \mid Z\} = \{X \mid Z\} \qquad\qquad (C_R) \quad \{X, Y \mid Z\} = \{Y, X \mid Z\}$$

(A_b stands for *absorption* and C_R stands for *commutativity on the right*–see [11]) uniquely identify a finest congruence $=_{\mathcal{T}}$ on $\tau(\Sigma)$. Following [11], two terms s and t are said to be *$\mathcal{T}$-unifiable* iff there is a substitution σ such that $s\sigma =_{\mathcal{T}} t\sigma$; such a σ is called a *$\mathcal{T}$-unifier*. We write, for any set of variables $W \subseteq \mathcal{V}$, $\sigma =_{\mathcal{T}}^{W} \rho$ iff $\forall x \in W \; x\sigma =_{\mathcal{T}} x\rho$. In the same way, σ is *more general than ρ in $\mathcal{T}$ over W* ($\sigma \leq_{\mathcal{T}}^{W} \rho$) iff $\exists \eta \; \eta \circ \sigma =_{\mathcal{T}}^{W} \rho$. The corresponding equivalence relation on substitutions is denoted by $\equiv_{\mathcal{T}}^{W}$; i.e., $\sigma \equiv_{\mathcal{T}}^{W} \rho$ iff $\sigma \leq_{\mathcal{T}}^{W} \rho$ and $\rho \leq_{\mathcal{T}}^{W} \sigma$. The set of all $\mathcal{T}$-unifiers of two terms s and t is denoted by $\bigcup_{\mathcal{T}}(s, t)$. A set U of $\mathcal{T}$-unifiers is said to be *complete* w.r.t. s and t if $(\forall \sigma \in \bigcup_{\mathcal{T}}(s,t))(\exists \theta \in U)(\theta \leq_{\mathcal{T}}^{W} \sigma)$, where $W = Var(s) \cup Var(t)$. The *set of most general $\mathcal{T}$-unifiers* of s and t, denoted by $\mu\bigcup_{\mathcal{T}}(s, t)$, is a minimal complete set of $\mathcal{T}$-unifiers if $(\forall \sigma\tau \in \mu\bigcup_{\mathcal{T}}(s,t))(\sigma \leq_{\mathcal{T}}^{W} \tau \Rightarrow \sigma \equiv_{\mathcal{T}}^{W} \tau)$, where $W = Var(s) \cup Var(t)$. Note that since $\mu\bigcup_{\mathcal{T}}(s, t)$ exists, it is unique up to $\equiv_{\mathcal{T}}^{\mathcal{V}}$ [7]. Similar definitions can be given starting from a system of equations $\mathcal{E}$ instead of the unification problem $s = t$. When the context is clear, we will omit the prefix $\mathcal{T}$ before the word unifier.

For any satisfiable Herbrand system $\mathcal{E}$ involving terms from $\tau(\Sigma \cup \mathcal{V})$, a unification algorithm should be able to compute through non-determinism *each element* of a complete set of unifiers of $\mathcal{E}$. Notice that it is not required that it computes exactly $\mu\bigcup_{\mathcal{T}}(\mathcal{E})$. However, as we will see in detail in Sect. 3, the presented theory $\mathcal{T}$ is such that, even for simple unification problems $s = t$, $\mu\bigcup_{\mathcal{T}}(s, t)$ becomes larger and larger. This means that a valid criterion to compare two (set) unification algorithms is the analysis of the length of the *list* of solutions computed by them (the use of the word 'list' here is needed to reflect the fact that if a unification algorithm computes exactly $\mu\bigcup_{\mathcal{T}}(s, t)$, but some solution is returned more than once, it cannot be considered minimal).

If the input system $\mathcal{E}$ is $\mathcal{T}$-unsatisfiable, any unification algorithm should conclude its computation reporting a *failure* result.

3 Sample Problems

In this section we single out eight set unification problems that we propose as 'benchmarks' to test the minimality of any set unification algorithm.

Given a unification problem between two sets (for instance $\{\{X, f(g(Y)) \mid R\}, f(X)\} = \{A, \{B, f(C)\} \mid R\}$) it is possible, in principle, to determine the number of most general and independent unifiers for it (i.e. the cardinality of $\mu\bigcup_{\mathcal{T}}(s,t)$). Nevertheless, it is impossible to test the *optimality* (namely the capability of returning exactly $\mid \mu\bigcup_{\mathcal{T}}(s,t) \mid$ solutions) for all possible problems $s = t$ with s, t in $\tau(\Sigma \cup \mathcal{V})$. A criterion for selecting some sample problems must be chosen:

- problems with *nested* sets (i.e. sets containing sets, such as example above) give rise to confusion in the analysis. It is more important to concentrate the efforts in pointing out the new unification problems between elements of the two sets that must be generated;
- with the signature Σ, *closed* sets (i.e. of the form $\{t_1 \mid \{t_2 \mid \cdots \{t_n \mid \{\}\} \cdots\}\}$) and *open* sets (i.e. of the form $\{t_1 \mid \{t_2 \mid \cdots \{t_n \mid R\} \cdots\}\}$, R variable) can be described. Any unification problem between two sets can be of one of the following forms:
  - closed with closed (problems (1)–(4));
  - open with closed (or vice versa) (problems (5) and (6));
  - open with open (problems (7) and (8)).

  In each of such cases, if the elements of the sets are distinct variables, the number of possible solutions is maximized (for instance $\{X_1, X_2, X_3\} = \{Y_1, Y_2, Y_3\}$ admits 15 solutions, meanwhile $\{X_1, X_2, X_3\} = \{a_1, a_2, a_3\}$, $\{X_1, X_2, X_3\} = \{Y_1, X_2, X_3\}$ and $\{X_1, X_2, X_3\} = \{Y_1, Y_2, X_3\}$ admit 6, 3, and 6 solutions, respectively).

As particular cases, it is interesting to analyze the cleverness of an algorithm in solving problems in which the two sets share elements (problems (3) and (4)). Moreover, also a shrewd treatment of the *matching* problem (namely when one of the two sets is ground) is important (problems (1) and (5)).

The unification problem between open sets must be considered differently whether the 'rest' variables are identical or not. The announced sample problems are the following:

$$
\begin{array}{lrcl}
(1) & \{X_1, \ldots, X_m\} & = & \{a_1, \ldots, a_n\}, \\
(2) & \{X_1, \ldots, X_m\} & = & \{Y_1, \ldots, Y_n\}, \\
(3) & \{X_1, \ldots, X_m, a_1, \ldots, a_k\} & = & \{Y_1, \ldots, Y_n, a_1, \ldots, a_k\}, \\
(4) & \{X_1, \ldots, X_m, Z_1, \ldots, Z_k\} & = & \{Y_1, \ldots, Y_n, Z_1, \ldots, Z_k\}, \\
(5) & \{X_1, \ldots, X_m \mid Z\} & = & \{a_1, \ldots, a_n\}, \\
(6) & \{X_1, \ldots, X_m \mid Z\} & = & \{Y_1, \ldots, Y_n\}, \\
(7) & \{X_1, \ldots, X_m \mid Z\} & = & \{Y_1, \ldots, Y_n \mid Z\}, \\
(8) & \{X_1, \ldots, X_m \mid W\} & = & \{Y_1, \ldots, Y_n \mid Z\},
\end{array}
$$

where $X_1, \ldots, X_m, Y_1, \ldots, Y_n, Z_1, \ldots, Z_k, W, Z$ are pairwise distinct variables, and $a_1, \ldots, a_{\max\{n,k\}}$ are pairwise distinct constant symbols.

Due to space limits we will not enter into the details of the functions which compute the cardinality of $\mu\bigcup_{\mathcal{T}}$ for problems (1)–(8). A complete analysis of such functions can be found in [1]. We only give a taste of them by presenting the following tables which report some numerical values, where the number on the 'x' axis denotes the value for m (the first parameter in our unification problems).

(1)	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2		2	6	14	30	62	126
3			6	36	150	540	1806
4				24	240	1560	8400
5					120	1800	16800
6						720	15120
7							5040

(2)	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2		2	6	14	30	62	126
3			15	48	165	558	1827
4				184	680	2664	11032
5					2945	13080	59605
6						63756	320292
7							1748803

Problem (3) (numerically equal to problem (4)) would require a three-dimensional matrix to represent its values. Assume $k = 3$:

(3)	1	2	3	4	5	6	7
1	7	19	61	223	877	3559	14581
2		56	195	746	3093	13808	65391
3			705	2859	12681	60231	302829
4				12226	56891	284286	1510483
5					277091	1448325	8044117
6						7888698	45590823
7							273498973

(5)	1	2	3	4	5	6	7
1	2	2	2	2	2	2	2
2	4	12	28	60	124	252	508
3	6	30	126	462	1566	5070	15966
4	8	56	344	1880	9368	43736	195224
5	10	90	730	5370	36250	228090	1359130
6	12	132	1332	12372	106452	856212	6505812
7	14	182	2198	24710	259574	2562182	23928758

(6)	1	2	3	4	5	6	7
1	2	2	2	2	2	2	2
2	5	12	28	60	124	252	508
3	10	42	144	486	1596	5106	16008
4	19	126	584	2584	11208	48248	205864
5	36	360	2200	11930	63000	330450	1733000
6	69	1016	8118	52740	325812	1983084	12073836
7	134	2870	29876	231518	1641444	11310530	77511140

(7)	1	2	3	4	5	6	7
1	2	4	8	16	32	64	128
2		11	30	85	248	735	2194
3			94	308	1104	4210	16538
4				1041	3920	16981	80260
5					14006	59412	303428
6						221971	1048054
7							4063382

(8)	1	2	3	4	5	6	7
1	4	9	18	35	68	133	262
2		39	131	413	1185	3459	10071
3			652	2811	11402	44983	175224
4				15937	82499	409897	1997795
5					524056	3133773	18217350
6						21998671	148144723
7							1136372140

4 The Algorithm SUA

In order to make the description of the algorithm as clear as possible, some local notation will be defined.

The *set-operations* '$|\cdot|$' (cardinality), '$\subseteq$' (inclusion), '$\subset$' (strict inclusion), '$\cup$', (union) '$\cap$', (intersection) and '$-$' (set difference) will be used on terms denoting sets. The meaning of the set operators is purely syntactical; for instance $|\{X_1, X_2\}| = 2$: we do not need to distinguish the two cases $X_1 = X_2 \wedge |\{X_1, X_2\}| = 1$ and $X_1 \neq X_2 \wedge |\{X_1, X_2\}| = 2$. The function *Can* which removes duplicates in a term representing a set ensures that they can be used without semantical problems.

A set of equations $\mathcal{E}$ is said to be in *solved form* if it has the form $\{X_1 = t_1, \ldots, X_n = t_n\}$ where X_i's are distinct variables not occurring in t_j, for all $i, j \in \{1, \ldots, n\}$. A solved form system $\{X_1 = t_1, \ldots, X_n = t_n\}$ can be viewed as the substitution $\{X_1/t_1, \ldots, X_n/t_n\}$. An equation $X = t$ is said to be *solved* in $\mathcal{E}$ if X does not occur neither in t nor in $\mathcal{E} \setminus \{X = t\}$.

Actions 1–8 of the unification algorithm SUA are identical to the used ones in the unification algorithm presented in [3] (for further details the reader can consult that paper). SUA takes as input a system of equations $\mathcal{E}$ between terms and returns either *fail* –$\mathcal{E}$ is not unifiable– or, non-deterministically, a substitution θ. The set of all such θ's constitutes a complete set of T-unifiers. In the algorithm, X, W, W', Z and Z' denote generic variables, $t, t_1, t_2, \ldots, s_1, s_2, \ldots$ denote generic terms, N denotes a new variable introduced by SUA. k and k' will denote non-variable terms whose main functional symbol is distinct from $\{\cdot \mid \cdot\}$[1]. $Var(\ell)$ represents the set of variables occurring in the term ℓ, while $\theta|_S$ constraints the domain of θ to the variables contained in S. n.d. is a denotation for non-deterministically. The algorithm temporarily generates equations marked by '$*$'; they are called *active* equations and they are immediately removed by action 0. $\mathcal{E}^*$ denotes the set of active equations.

function SUA($\mathcal{E}$);
If $\mathcal{E}$ is in solved form then return $\mathcal{E}$
elseif $\mathcal{E}^*$ is not empty
then choose n.d. one *active* equation $s \stackrel{*}{=} t$ in $\mathcal{E}^*$; $\mathcal{E}' := \mathcal{E} - \{s \stackrel{*}{=} t\}$;
0. i. $s \equiv X$: if X occurs in t then *fail* else return SUA($\mathcal{E}'[X/t] \cup \{X = t\}$);
   ii. $s \equiv f(s_1, \ldots, s_m)$ and $t \equiv f'(t_1, \ldots, t_n)$: if $f \not\equiv f'$ then *fail* else
   (i.e. $f \equiv f'$ and $m = n$): return SUA($\{s_1 = t_1, \ldots, s_n = t_n\} \cup \mathcal{E}'$);

else choose n.d. one equation e (not in solved form) in $\mathcal{E}$; $\mathcal{E}' := \mathcal{E} - \{e\}$; case e of:
1. $X = X$: return SUA($\mathcal{E}'$);
2. $t = X$ and t is not a variable: return SUA($\{X = t\} \cup \mathcal{E}'$);
3. $X = t$, X is a variable occurring in $\mathcal{E}'$ but not in t: return SUA($\mathcal{E}'[X/t] \cup \{X = t\}$);
4. $X = \{t_1, \ldots, t_m \mid X\}$: return SUA($\mathcal{E}' \cup \{X \stackrel{*}{=} \{t_1, \ldots, t_m \mid N\}\}$);
5. $X = \{t_1, \ldots, t_m \mid t\}$, where t is a variable or $t \equiv f(t_1, \ldots, t_n)$, $f \not\equiv \{\cdot \mid \cdot\}$, and X occurs in t_1, or ..., or in t_m, or t is not a variable and X occurs in t: *fail*;
6. $X = t$, and $t \equiv f(t_1, \ldots, t_n)$, $f \not\equiv \{\cdot \mid \cdot\}$ and X is a variable occurring in t: *fail*;

[1] Such entities are named *kernels* in [3].

7. $f(s_1, \ldots, s_n) = g(t_1, \ldots t_m)$, where $f, g \in \Sigma$, $f \not\equiv g$: *fail*;
8. $f(s_1, \ldots, s_n) = f(t_1, \ldots t_n)$, $f \in \Sigma$, $f \not\equiv \{\cdot \mid \cdot\}$:
   return SUA($\mathcal{E}' \cup \{s_1 = t_1, \ldots, s_n = t_n\}$);
9. $\{t_1, \ldots, t_m \mid k\} = \{s_1, \ldots, s_n \mid k'\}$:
   return SUA(unify_set($\{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\}$) $\cup \{k \stackrel{*}{=} k'\} \cup \mathcal{E}'$);
10. $\{s_1, \ldots, s_n \mid k\} = \{t_1, \ldots, t_m \mid Z\}$:
    return SUA($\{\{t_1, \ldots, t_m \mid Z\} = \{s_1, \ldots, s_n \mid k\}\} \cup \mathcal{E}'$);
11. $\{t_1, \ldots, t_m \mid Z\} = \{s_1, \ldots, s_n \mid k\}$, choose n.d. $\{\} \neq S \subseteq Can(\{s_1, \ldots, s_n\})$: return
    SUA($\{Z \stackrel{*}{=} Can(\{s_1, \ldots, s_n\} - S \cup Z' \cup k)\} \cup$
    limit_1($\{t_1, \ldots, t_m\}, S, Z', \mathcal{E}')|_{Var([s_1,\ldots,s_n,t_1,\ldots,t_m,Z])}$);
12. $\{t_1, \ldots, t_m \mid Z\} = \{s_1, \ldots, s_n \mid Z\}$: select n.d. one of the following actions:
    i. choose n.d. $T \subseteq \{t_1, \ldots, t_m\}$ and $S \subseteq \{s_1, \ldots, s_n\}$ s.t. $T \cup S \neq \{t_1, \ldots, t_m, s_1, \ldots, s_n\}$: return SUA(unify_set($T, S$) $\cup \{Z \stackrel{*}{=} Can((\{t_1, \ldots, t_m\} - T) \cup (\{s_1, \ldots, s_n\} - S) \cup N)\} \cup \mathcal{E}'$);
    ii. return SUA(unify_set($\{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\}$) $\cup \mathcal{E}'$);
13. $\{t_1, \ldots, t_m \mid W\} = \{s_1, \ldots, s_n \mid Z\}$ where Z and W are different variables; choose n.d. $T \subseteq Can(\{t_1, \ldots, t_m\})$, $S \subseteq Can(\{s_1, \ldots, s_n\})$:
    return SUA($\{W \stackrel{*}{=} Can(\{s_1, \ldots, s_n\} - S \cup W'), Z \stackrel{*}{=} Can(\{t_1, \ldots, t_m\} - T \cup Z')\} \cup$
    limit_2($T, S, Z', W', \mathcal{E}'$)$|_{Var([s_1,\ldots,s_n,t_1,\ldots,t_m,Z,W])}$).

Active equations $s \stackrel{*}{=} t$ should be handled before the others in order to guarantee termination (see details in the proof of Theorem 1). They are temporarily introduced by actions 4, 9, 11, 12, and 13.

The function unify_set takes as input two terms S_1 and S_2 representing nonempty sets; it selects non-deterministically which equalities between elements of S_1 and S_2 should accompany the system $\mathcal{E}$ in a recursive call to SUA.

function unify_set(S_1, S_2);
Let $\{t_1, \ldots, t_m\} = Can(S_1)$; $\{s_1, \ldots, s_n\} = Can(S_2)$;
1. If $\{t_1, \ldots, t_m\}$ and $\{s_1, \ldots, s_n\}$ are syntactically equal then return SUA($\mathcal{E}$)
2. elseif $m = 1$ and $n > 1$ then return $\{s_i = t_1 : 1 \leq i \leq n\}$;
3. elseif $m \geq 1$ and $n = 1$ then return $\{t_i = s_1 : 1 \leq i \leq m\}$
4. else $Common_part := \{t_1, \ldots, t_m\} \cap \{s_1, \ldots, s_n\}$;
   $Disagr_1 := \{t_1, \ldots, t_m\} - Common_part$;
   $Disagr_2 := \{s_1, \ldots, s_n\} - Common_part$;
   (a) if $Common_part = \{\}$ then fix an $i \in \{1, \ldots, m\}$: select n.d. one of the following actions:
      i. return $\stackrel{1}{=} (t_i, \{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\})$;
      ii. return $\stackrel{2}{=} (t_i, \{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\})$;
      iii. return $\stackrel{3}{=} (t_i, \{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\})$;
   (b) if $Common_part \neq \{\}$ then choose n.d. $S_0, S_1 \subseteq Common_part$, $S_0 \cap S_1 = \{\}$; choose n.d. $T_0 \subseteq Disagr_1$ and $T_1 \subseteq Disagr_2$ such that $|T_0| \geq |S_0|$ and $|T_1| \geq |S_1|$: return unify_set($Disagr_1 - T_0, Disagr_2 - T_1$) $\cup$ unify_set2(T_1, S_1) $\cup$ unify_set2(T_0, S_0).

Some comments on unify_set are needed in order to relate it to the unification problems (1), (2), (3) and (4). Action 4.(a) is motivated by problems of type (1)

and (2). The basic idea is that when an answer is computed by SUA without using the function $\stackrel{2}{=}$, then such answer may be considered as a surjective function from the leftmost set to the rightmost one. Note that the cardinality of the minimal complete set of most general unifiers for problem (1) is exactly the number of surjective functions between $\{X_1, \ldots, X_m\}$ and $\{a_1, \ldots, a_n\}$. Since $\stackrel{2}{=}$ can not be applied to problem (1) (otherwise SUA would produce *fail*), then the algorithm computes surjective functions as solutions for such problem. On the other hand, for problem (2), $\stackrel{2}{=}$ must be considered in order to capture solutions containing $Y_{k_1} = X_i, \ldots, Y_{k_j} = X_i$ (named k_j-*forks*), for some $j > 1$, $1 \leq i \leq m$. The intended meaning of unify_set2 is to consider non deterministically either $\stackrel{1}{=}$ or $\stackrel{3}{=}$, but never $\stackrel{2}{=}$, since to consider *k-forks* produces redundant solutions for problem (4)[2]. The formal definition of unify_set2 is the following:

function unify_set2($\{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\}$);
1. if $m = 1$ and $n > 1$ then return $\{t_1 = s_i : i = 1, \ldots, n\}$;
2. if $m \geq 1$ and $n = 1$ then return $\{t_i = s_1 : i = 1, \ldots, m\}$;
3. if $m, n > 1$ then fix a value $i \in \{1, \ldots, m\}$; select n.d. one of the following actions:
   i. return $\stackrel{1}{=}(t_i, Can(\{t_1, \ldots, t_m\}), Can(\{s_1, \ldots, s_n\}))$;
   ii. return $\stackrel{3}{=}(t_i, Can(\{t_1, \ldots, t_m\}), Can(\{s_1, \ldots, s_n\}))$.

See now the definitions of functions $\stackrel{i}{=}$, $1 \leq i \leq 3$:
$\stackrel{1}{=}$ matches one element t_i of the first set with one element s_j of the second one, and combines the two sets deprived of the selected elements. $\stackrel{1}{=}$ has the following structure:

function $\stackrel{1}{=}(t_i, \{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\})$;
choose n.d. one $j \in \{1, \ldots, n\}$:
return $\{t_i = s_j\} \cup$ unify_set$(\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}, \{s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n\})$.

$\stackrel{2}{=}$ captures the concept of *k-fork* and is defined as follows:

function $\stackrel{2}{=}(t_i, \{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\})$;
choose n.d. $S \subset \{s_1, \ldots, s_n\}$ such that $|S| \geq 2$:
return $\{t_i = s :$ for all $s \in S\} \cup$ unify_set$(\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}, \{s_1, \ldots, s_n\} - S\})$.

$\stackrel{3}{=}$ try to unify $k > 1$ elements of the leftmost set to only one element of the rightmost one. In particular, for problem (2), $\stackrel{3}{=}$ produces bindings of the form $X_{k_1} = Y_j, \ldots, X_{k_i} = Y_j$, for some $1 \leq j \leq n$, $i > 1$, named k_i-*cones*.

function $\stackrel{3}{=}(t_i, \{t_1, \ldots, t_m\}, \{s_1, \ldots, s_n\})$;
choose n.d. $\{\} \neq T \subset \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}$:
return $\{t_i = t :$ for all $t \in T\} \cup \stackrel{1}{=}(t_i, \{t_1, \ldots, t_m\} - T, \{s_1, \ldots, s_n\})$.

A problem of type (6) can only be solved by using action 11 of SUA. The function limit_1 must simultaneously solve the unification problem $\{X_1, \ldots, X_m\} = S$, for some $S \subseteq \{Y_1, \ldots, Y_n\}$ (for such reason, the definitions of $\stackrel{1}{=}$, $\stackrel{2}{=}$, and $\stackrel{3}{=}$ are embedded in its definition) and bind Z to any subset of S not containing variables occurring in any *h-fork* corresponding to the solution being computed.

[2]For problem (3), the use of function $\stackrel{2}{=}$ is not possible.

function limit_1($S_1, S_2, Z, \mathcal{E}$);
Let $\{t_1, \ldots, t_m\} = Can(S_1); \{s_1, \ldots, s_n\} = Can(S_2)$;
1. If $\{t_1, \ldots, t_m\}$ and $\{s_1, \ldots, s_m\}$ are syntactically equal
   then return SUA($\{Z = \{\}\} \cup \mathcal{E}$);
2. elseif $m = 1$ and $n > 1$ then return SUA($\{s_i = t_1 : 1 \leq i \leq n\} \cup \{Z = \{\}\} \cup \mathcal{E}$)
3. elseif $m = 1$ and $n = 1$ or $m > 1$ and $n = 1$ then choose n.d. $T \subseteq \{s_1\}$ then
   return SUA($\{t_i = s_1 : 1 \leq i \leq m\} \cup \{Z = T\} \cup \mathcal{E}$)
4. else fix $i \in \{1, \ldots, m\}$ and choose n.d. one of the following actions:
   i. choose n.d. $j \in \{1, \ldots, n\}$ and $T \subseteq \{s_j\}$:
      return SUA($\{t_i = s_j\} \cup \{Z = T \cup Z'\} \cup$
      limit_1($\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}, \{s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n\}, Z', \mathcal{E}$));
   ii. choose n.d. $S \subset \{s_1, \ldots, s_n\}$ s.t. $|S| \geq 2$: return SUA($\{t_i = s :$ for all $s \in S\}$
      $\cup$ limit_1($\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}, \{s_1, \ldots, s_n\} - S, Z, \mathcal{E}$));
   iii. choose n.d. $T' \subset \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}$, $j \in \{1, \ldots, n\}$ and $T \subseteq \{s_j\}$:
      return SUA($\{t = s_j :$ for all $t \in T'\} \cup \{t_i = s_j\} \cup \{Z = T \cup Z'\} \cup$
      limit_1($\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\} - T', \{s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n\}, Z', \mathcal{E}$)).

The definition of limit_2 is similar to the one of limit_1. In particular, for problems of type (8), the values of W and Z are constrained in order to avoid the introduction of variables occurring in some *h-fork* or *h-cone* respectively, of the solution θ being computed. On the other hand, limit_2 must also control that those variables bounded in θ by a simple binding are not introduced in Z and W simultaneously.

function limit_2($S_1, S_2, Z, W, \mathcal{E}$);
Let $\{t_1, \ldots, t_m\} = Can(S_1); \{s_1, \ldots, s_n\} = Can(S_2)$;
1. If $\{t_1, \ldots, t_m\}$ and $\{s_1, \ldots, s_n\}$ are syntactically equal then
   return SUA($\{Z = \{\}\} \cup \{W = \{\}\} \cup \mathcal{E}$);
2. elseif $m = 1$ and $n > 1$ then choose n.d. $T \subseteq \{t_1\}$:
   return SUA($\{s_i = t_1 : 1 \leq i \leq n\} \cup \{Z = T\} \cup \{W = \{\}\} \cup \mathcal{E}$);
3. elseif $n = 1$ and $m > 1$ then choose n.d. $S \subseteq \{s_1\}$:
   return SUA($\{t_i = s_1 : 1 \leq i \leq m\} \cup \{Z = \{\}\} \cup \{W = S\} \cup \mathcal{E}$);
4. elseif $m = 1$ and $n = 1$ then choose n.d. $T \subseteq \{t_1\}$, $S \subseteq \{s_1\}$ s.t. $T \cup S \neq \{t_1, s_1\}$:
   return SUA($\{t_i = s_1 : 1 \leq i \leq m\} \cup \{Z = T\} \cup \{W = S\} \cup \mathcal{E}$);
5. else fix $i \in \{1, \ldots, m\}$ and choose n.d. one of the following actions:
   i. choose n.d. $j \in \{1, \ldots, n\}$ and $S \subseteq \{s_j\}$, $T \subseteq \{t_j\}$ such that $T \cup S \neq \{s_j, t_i\}$:
      return SUA($\{t_i = s_j\} \cup \{W = S \cup W'\} \cup \{Z = T \cup Z'\} \cup$
      limit_2($\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}, \{s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n\}, Z', W'\mathcal{E}$));
   ii. choose n.d. $S \subset \{s_1, \ldots, s_n\}$ s.t. $|S| \geq 2$, $T \subseteq \{t_i\}$:
      return SUA($\{t_i = s :$ for all $s \in S\} \cup \{Z = T \cup Z'\} \cup$
      limit_2($\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}, \{s_1, \ldots, s_n\} - S, Z', W, \mathcal{E}$));
   iii. choose n.d. $T \subset \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\}$, $j \in \{1, \ldots, n\}$ and $S \subseteq \{s_j\}$:
      return SUA($\{t = s_j :$ for all $t \in T\} \cup \{t_i = s_j\} \cup \{W = T \cup W'\} \cup$
      limit_2($\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\} - T, \{s_1, \ldots, s_{j-1}, s_{j+1}, \ldots, s_n\}, Z, W', \mathcal{E}$)).

4.1 Termination, Correctness, and Minimality of SUA

The following definition is helpful for the termination proof. Let $\mathcal{E}$ be a system of equations, and let p be the total number of occurrences of function symbols in it. Then a function $lev : Var(\mathcal{E}) \longrightarrow \omega$ extended to non-variable terms as follows

$$\begin{aligned} lev(f(t_0,\ldots,t_n)) &= \max\{1+lev(t_0),\ldots,1+lev(t_n)\} \quad \text{when } f \in \Sigma,\ f \not\equiv \{\cdot\,|\,\cdot\} \\ lev(\{t\,|\,s\}) &= \max\{1+lev(t), lev(s)\} \end{aligned}$$

and fulfilling condition *(i)* $lev(\ell), lev(r) \leq p$, for any equation $\ell = r$ in $\mathcal{E}$ always exists. If we require the further condition *(ii)* $lev(\ell) = lev(r)$ for any equation $\ell = r$ in $\mathcal{E}$ then such *lev* may not exist (e.g. when $\mathcal{E} = \{X = f(X)\}$).

Lemma 1 *Let $\mathcal{E}$ be a satisfiable equation system; then a function lev whose extension to terms fulfills both conditions (i) and (ii) always exists.*

Let $\{X = t\} \cup \mathcal{E}$ be an equation system and let p be the number of occurrences of functional symbols in it. Assume X does not occur in t and assume the function lev fulfills condition (i). Moreover, assume that lev is such that $lev(X) = lev(t)$. Then lev fulfills condition (i) also for the system $\mathcal{E}[X/t] \cup \{X = t\}$.

Algorithm SUA calls the functions unify_set, limit_1, and limit_2. To prove the termination of them is straightforward. In the proof of the following Theorem we will make use also of their semantics.

Theorem 1 (Termination) *For any input system $\mathcal{E}$,* SUA *always terminates, no matter what non-deterministic sequence of choices is made.*

Proof Assume that an infinite sequence of non-deterministic choices s.t. SUA($\mathcal{E}$) does not terminate, exists. Let $E^{(0)}, E^{(1)}, E^{(2)}, \ldots$ be the values of $\mathcal{E}$ at the $0^{th}, 1^{st}, 2^{nd}, \ldots$ iteration, respectively, and let p be the number of occurrences of functional symbols in $E^{(0)}$. A function $lev : Var(\bigcup_{j\geq 0} E^{(j)}) \longrightarrow \omega$ s.t.

- it fulfills condition (i) for all equation sets $E^{(j)}$, and
- any time a substitution $[X/t]$ has been applied (actions 0 and 3), then $lev(X) = lev(t)$,

must necessarily exist. If it did not exist, then a failure situation caused by occur check would rise, causing the computation be finite.

Picking such *lev*, we define a measure of complexity $\mathcal{L}_\mathcal{E}$ for the equation set $\mathcal{E}$: $\mathcal{L}_\mathcal{E} =_{Def} [\#(2p), \#(2p-1), \#(2p-2), \ldots, \#(1), \#(0)]$ where $\#(j)$ returns the number of equations *not in solved form* $\ell = r$ in $\mathcal{E}$ s.t. $lev(\ell) + lev(r) = j$. The ordering between two lists of this form is the well-founded lexicographical ordering.

It is easy to see that actions 0, 1, 3, and 8 cause $\mathcal{L}_\mathcal{E}$ to strictly decrease. Actions 4, 9, 11, 12, and 13 do not increase $\mathcal{L}_\mathcal{E}$; however they introduce active (i.e. marked by '$*$') equations which will fire immediately action 0. Actions 2 and 10 leave $\mathcal{L}_\mathcal{E}$

unchanged; nevertheless they at most can double the number of actions, hence we may forget them. Since the lexicographical ordering on constant-length lists of non-negative integers is a well-ordering, this is sufficient to prove termination. □

The above Theorem proves that the search tree built by SUA does not contain infinite branches. Since such tree is finitely branched, it is guaranteed that SUA computes a finite number of unifiers.

The following Theorem is a direct consequence of the corresponding one presented in [3] and therefore we omit the proof here.

Theorem 2 (Correctness and Completeness) *The function* SUA *is correct with respect to the equational theory presented in Sect. 2. Moreover, it is correct and complete with respect to the well-founded theory of hybrid sets presented in [3, 5].*

It remains to show that the unification algorithm SUA is optimal w.r.t. all chosen sample problems. We will deal only with set terms ended by the constant symbol $\{\}$. This simplifies the analysis of actions 9 and 10 of the algorithm (k and k' are both $\{\}$). The very long and technical proof is omitted due to space limits.

Theorem 3 (Minimality) SUA *is optimal w.r.t. all the eight sample problems described in Sec. 3, i.e.,* SUA *enumerates without repetitions a minimal complete set of unifiers, to any given unification problem belonging to one of the eight given kinds.*

5 Conclusions

In this paper we have presented a new set unification algorithm SUA, optimal for a significant set of sample problems, that is computing exactly the minimal complete set of most general unifiers for them. Tables reporting the optimal number of solutions for such sample problems have been also presented. These tables show that developing minimal set unification algorithms is an interesting line of research: since the minimal number of solutions is big by itself, it is very important to avoid repeated solutions and solutions being instances of other ones. Although it has been only proved that SUA is optimal for the eight problems presented in Sec. 3, due to their relevance –they can be used for testing any set unification algorithm– it is reasonable to expect a good behavior of SUA for any instance of the set unification problem. Finally, we deem that SUA can adventageously replace any other known set unification algorithms in implementations of set-based logic programming languages (for instance in the implementation of {log} presented in [4]).

Acknowledgements

We are very grateful to Giorgio Levi for giving us the possibility of working together in Pisa. We would like to thank Alberto Policriti, Enrico Pontelli, Mario Rodríguez-

Artalejo, Gianfranco Rossi, Alvaro Ruiz-Andino, and an anonymous referee for their wise advises. A special 'thank you' to Ana Gil-Luezas for her unconditional help.

References

[1] ARENAS-SÁNCHEZ, P., AND DOVIER, A. Minimal Set Unification. Technical Report: TR–6/95, Università di Pisa, dip. di Informatica, april 1995.

[2] BÜTTNER, W., AND SIMONIS, H. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation 4* (1987), 191–205.

[3] DOVIER, A., OMODEO, E., PONTELLI, E., AND ROSSI, G. Embedding Finite Sets in a Logic Programming Language. In *3rd Int.l Workshop on Extension of Logic Programming* (1993), E. Lamma and P. Mello, Eds., vol. 660 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 150–167.

[4] DOVIER, A., AND PONTELLI., E. A WAM based Implementation of a Logic Language with Sets. In *Proc. Fifth Int'l Symp. on Programming Language Implementation and Logic Programming* (1993), M. Bruynooghe and J. Penjam, Ed., vol. 714 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 275–290.

[5] DOVIER, A., AND ROSSI, G. Embedding Extensional Finite Sets in CLP. In *Proc. of International Logic Programming Symposium, ILPS'93* (1993), D. Miller, Ed., The MIT Press, pp. 540–556.

[6] GERVET, C. Conjunto: Constraint Logic Programming with Finite Set Domains. In *Proc. of International Logic Programming Symposium, ILPS'94* (1994), M. Bruynooghe, Ed., The MIT Press, pp. 339–358.

[7] HUET, G. Résolution d'équations dans des langages d'ordre 1,2,..., ω. Thèse d'État, Univ. de Paris, VII, 1976.

[8] JAFFAR, J., AND MAHER, M. J. Constraint Logic Programming: A Survey. *The Journal of Logic Programming 19–20* (1994), 503–581.

[9] JAYARAMAN, B. Implementation of Subset-Equational Programs. *Journal of Logic Programming 12*, 4 (1992), 299–324.

[10] LEGEARD, B., AND LEGROS, E. Short overview of the CLPS system. In *Proc. Third Int'l Symp. on Programming Language Implementation and Logic Programming* (august 1991), J. Maluszynsky and M. Wirsing, Eds., vol. 528 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 431–433.

[11] SIEKMANN, J. H. Unification theory. In *Unification*, C. Kirchner, Ed. Academic Press, 1990.

[12] STOLZENBURG, F. An Algorithm for General Set Unification and its Complexity. In *Proc. of Workshop on Logic Programming with Sets, in conjunction with ICLP'93* (1993), E. G. Omodeo and G. Rossi, Eds.

A Model Tree Computation of the Strong Well–Founded Semantics

Cristian Papp
Facultatea de informatică
Universitatea "Al. I. Cuza"
Iaşi 6600 România e-mail: papp@uaic.ro

Abstract. The model–tree data structure was introduced by Fernández and Minker in [FM91] as a tool for representing a set of minimal Herbrand models of a finite theory. Later, in [FM94] they found a characterization of the *perfect model semantics* in terms of minimal models. This approach permitted the design of an algorithm for computing the perfect model of a disjunctive deductive database, based on model–trees. In this paper, we extend the results of [FM94] to a more general semantics, the *modified strong well–founded semantics*, defined by Brass and Dix in [BD94]. Their semantics is a slight modification of Ross *strong well–founded semantics*, [Ross90], which was defined in a top–down, operational manner. Although Brass and Dix also presented a bottom–up fixpoint definition for their own version of strong well–founded semantics, their bottom–up definition is based on program transformation and it is inefficient to implement. Our fixpoint definition is based on a new data structure, called *enhanced model trees*. which are designed to represent pairs of sets of minimal interpretations with the property that interpretations in the second set are extensions of interpretations in the first set. Then we define operators which act on enhanced model–trees and give an alternating fixpoint computation of the strong well–founded semantics.

Keywords: deductive database, model tree, well–founded semantics

1 Introduction

A growing interest in the field of disjunctive logic programming and disjunctive deductive databases can be remarked in the last few years (cf. workshops at the *ICLP, ILPS* and *IFIP*'94 conferences). A great amount of research was done in the semantics of disjunctive logic programs. We cite only a few of the best known proposed semantics for disjunctive logic programs with negation: *stratification semantics* [MR89], which is a generalization of the strafication semantics for normal programs [ABW86], *disjunctive stable semantics* [Prz90a], which extends the stable semantics of [GL88] to the disjunctive case, and several proposals of extending the *well–founded semantics* [VGRS91]: Przymusinski's *stationary semantics* [Prz90b],

Ross's *strong and weak well-founded semantics* [Ross90], the *generalized disjunctive well-founded semantics* of [LMR92]. But there are a lot more!

We can remark a much less significant effort in implementing semantics for disjunctive deductive databases. We mention the works of Fernández and Minker [FM91], [FM94], D. Seipel [Sei94], [ST94] and M. Müller and J. Dix [MD93]. While D. Seipel uses as a basis for his implementation a data structure for representing disjunctive facts called *clause trees*, M. Müller and J. Dix approach is based on the use of *fringes* (introduced in [NNS91]).

In [FM91] Fernández and Minker propose a new data structure for representing the set of minimal Herbrand interpretations of a finite theory, called *model trees*. They use this data structure to implement a query evaluation procedure for *hierarchical disjunctive databases* (cf. [FM91]). Later, they extended their work for the class of *stratified disjunctive databases* (cf. [FM94]).

Our paper extends their work further on, by considering a more general semantics: the *strong well-founded semantics* of Ross [Ross90] in its modified form proposed by Brass and Dix [BD94] in order to correct a "faulty" behaviour of the original definition. This semantics applies not only to stratified databases but for *all* the databases.

The paper is structured as follows: Section 1 is the present introduction. Section 2 contains the basic notions needed in order to understand the rest of the paper along with Brass and Dix definition of the strong well-founded semantics. In Section 3 a new data strucure for representing pairs of sets of minimal interpretations is introduced. This data structure is an enhancement of the *model trees* proposed in [FM94]. Using this new data structure, yet another characterization of the strong well-founded semantics is obtained in Section 4. This characterization is closer to a practical implementation. The results in this section are a revised form of the author's previous result on reformulating Ross semantics (see [Papp94]). This new result is a generalization of Fernández and Minker characterization of *perfect model semantics*. (cf. [FM94]). Section 5 contains the conclusions and plans for future research.

2 Basic notions and definitions

In this section we present the basic notions and definitions which are needed in the rest of the paper. We assume that the reader already knows basic notions of mathematical logic and logic programming. For more details the interested reader may consult [Llo87].

A *disjunctive clause* is a rule of the form

$$A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n \wedge \neg C_1 \wedge \ldots \wedge \neg C_m$$

where A_i, B_i and C_i are atoms, $k \geq 1$, $m \geq 0$, $n \geq 0$. All variables in the clause are assumed to be universally quantified at the front of the clause. A *normal clause* is

a rule of the above form with the restriction that $k = 1$. An extended literal is a disjunction of atoms or the negation of a disjunction of atoms.

A disjunctive clause is said to be *range restricted* when all the variables that occur in the head or in a negative literal in the body of the clause also occur in some positive literal of its body. This condition ensure the possibility of representing a set of atoms by its complement in the Herbrand base.

A *disjunctive deductive database* DB is a finite set of range restricted disjunctive clauses which do not contain function symbols.

Definition 1 *If S is a set of extended literals, the* closure *of S, written $cl(S)$, is the least set S' of extended literals containing S that satisfies the following two conditions:*

- *If the disjunction d is in S' then every disjunction containing d is in S'.*
- *For all disjunctions d_1 and d_2, $\neg d_1 \in S'$ and $\neg d_2 \in S'$ iff $\neg(d_1 \vee d_2) \in S'$.*

We say that S is consistent if $cl(S)$ does not contain a pair of complementary extended literals.

Definition 2 *A (3-valued)* extended interpretation *for a disjunctive program is a consistent set of ground extended literals. If d is a disjunction then*

- *If $d \in I$ then d is* true *with respect to I.*
- *If $\neg d \in I$ then d is* false *with respect to I.*
- *If neither $d \in I$ nor $\neg d \in I$ then d is* undefined *with respect to I.*

The *Herbrand universe of* a disjunctive deductive database DB, denoted HU_{DB}, is the set of constants which appear in DB. The *Herbrand base* of P, denoted HB_{DB}, is the set of atomic formulas formed by predicate symbols in DB whose arguments are in the Herbrand universe of DB. The *extended Herbrand base* of DB, denoted EHB_{DB}, is the set of disjunctions of atoms from HB_{DB}. If DB is a disjunctive deductive database and r a disjunctive clause in DB. A *Herbrand instantiation* of r is a disjunctive clause obtained from r by substituting terms from HU_{DB} for all the variables in r. The *Herbrand instantiation* of DB, denoted DB_{inst}, is the set of all instantiated rules of DB.

We present one of the extensions of the well-founded semantics for normal programs to the disjunctive case. We shall present a version which is due to Brass and Dix [BD94].

Definition 3 (Strong Derivation Step) *Let Q be an extended interpretation. We say Q' is* strongly derived *from Q (written $Q \Leftarrow Q'$) if Q contains a disjunction $D \in Q$ and there is some instantiated rule r given by $A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n \wedge \neg C_1 \wedge \ldots \wedge \neg C_m \in DB_{inst}$ such that $D \cap \{A_1, \ldots, A_k\} \neq \emptyset$ and*

$$Q' = (Q - \{D\}) \cup \{B_1 \vee D, \ldots, B_m \vee D, \neg C_1, \ldots, \neg C_m, \neg(\{A_1, \ldots, A_k\} \setminus D)\}$$

Definition 4 (Strong Basis) *Let $Q_0 \Leftarrow Q_1 \Leftarrow \cdots \Leftarrow Q_n$ be a derivation sequence ending in a set $Q_n = \{\neg\mathcal{N}_1, \ldots, \neg\mathcal{N}_k\}$ of negated disjunctions. Then we call $\mathcal{N} = \mathcal{N}_1 \cup \ldots \cup \mathcal{N}_k$ a basis for Q_0. If $Q_0 = \{D\}$, where D is a disjunction. we shall also call $\mathcal{N}$ a basis for D.*

Definition 5 (Strong Well–Founded Model) *Disjunctions are assigned truth values by the following rules:*

- *The empty disjunction is assigned the truth value* false.
- *If all bases for D are already assigned the truth value* true *(e. g. because D has no basis), then D is assigned the truth value* false.
- *If one basis for D is already assigned the truth value* false. *then D gets the truth value* true.
- *A disjunction D gets the truth value* undefined, *if it is not assigned a truth value by a finite number of applications of the above rules.*

We shall denote by SWFS(DB) the strong well–founded semantics of a disjunctive deductive database.

This definition is top–down and so it is more operational than declarative. Bottom–up, fixpoint definitions are highly preferable especially in the database community, both from theoretical and practical considerations. Actually [BD94] contains a bottom–up definition which is based on transformations on the original databases. We shall present in this paper two new bottom–up definitions for SWFS which are not based on program transformations. Our definitions are more convenient for a practical implementation.

3 Enhanced Model Trees: A Data structure

In this section we present the notion of *set of minimal interpretations* as well as two order relations which we shall need in defining our fixpoint computation. Then we shall device a data structure for representing pairs of sets of minimal inteepretations. The following definition is from [FM94]:

Definition 6 (Set of Minimal Interpretations) *Let $\mathcal{I}$ be a set of interpretations over HB(DB). Then, $\mathcal{I}$ is called a set of minimal Herbrand interpretations iff for each $I \in \mathcal{I}$ there is no $I' \in \mathcal{I}$ such that $I' \subset I$.*

A partial order is defined between sets of minimal interpretations.

Definition 7 *Let $\mathcal{J}$ and $\mathcal{I}$ be sets of minimal interpretations. Then $\mathcal{J} \leq \mathcal{I}$ iff $\forall I \in \mathcal{I}, \exists J \in \mathcal{J}, J \subseteq I$.*

We shall compute the well-founded making use of an alternating fixpoint method. The partial information reached at the n–th step of the computation will be represented as a pair of sets of minimal interpretations $< \mathcal{I}_n, \mathcal{PD}_n >$. The first element, $\mathcal{I}_n$ will encode the posite disjunctions which are concluded to be true, while the second element, $\mathcal{PD}_n$ will encode the set of potentially not false atoms. At any step of the computation $\mathcal{I}_n \leq \mathcal{PD}_n$. The first component of the pair will increase during the computation, while the second will decrease. Thus, we need a new relation order between pairs of sets of minimal interpretations.

Definition 8 *Let $< \mathcal{I}_1, \mathcal{J}_1 >$ and $< \mathcal{I}_2, \mathcal{J}_2 >$ be two pairs of sets of minimal interpretations. Then $< \mathcal{I}_1, \mathcal{J}_1 > \sqsubseteq < \mathcal{I}_2, \mathcal{J}_2 >$ iff $calI_1 \leq \mathcal{I}_2$ and $\mathcal{J}_2 \leq \mathcal{J}_1$.*

We need an efficient data structure to represent the pair $< \mathcal{I}_n, \mathcal{PD}_n >$ obtained in the n-th step of the computation. This new data structure is a generalization of the *model trees*, defined by Fernández and Minker in [FM91]. Briefly, the model tree data structure is designed such that it can efficiently represent a set of minimal models. The representation maps a set of minimal models $\mathcal{I}$ into a model tree, denoted $\mathcal{MT}(\mathcal{I})$.

We could represent the pair $< \mathcal{I}_n, \mathcal{PD}_n >$ simply as a pair of model trees and perform the fixpoint computation on pairs of model trees. This solution is inefficient, since we have that $\mathcal{I}_n \leq \mathcal{PD}_n, \forall n \geq 0$. This means that for every minimal interpretation I in $\mathcal{I}$, there is a minimal interpretation J in $\mathcal{PD}$ such that $I \subseteq J$. If we would represent the pair $< \mathcal{I}_n, \mathcal{PD}_n >$ as a pair of model trees $< \mathcal{MT}(\mathcal{I}_n), \mathcal{MT}(\mathcal{PD}_n) >$, then the whole tree $\mathcal{MT}(\mathcal{I}_n)$ would be a subtree of $\mathcal{MT}(\mathcal{PD}_n)$.

The solution is to represent the pair $< \mathcal{I}_n, \mathcal{PD}_n >$ on a single tree. However, this representation must distinguish between a minimal model I from $\mathcal{I}_n$ and the minimal model J from $\mathcal{PD}_n$ which extends it (i.e. $I \subset J$). Thus, the tree will have two sorts of nodes: nodes which represent atoms from I and nodes which represent atoms from $J \setminus I$.

First, we shall present the model tree data structure from [FM91]. Then, based on this data structure, we shall device another data structure, designed to represent pairs of finite sets of minimal Herbrand interpretations $< \mathcal{I}, \mathcal{J} >$ which have the particularity that $\mathcal{I} \leq \mathcal{J}$. The following two definitions are borrowed from [FM91]:

Definition 9 *Let $\mathcal{I}$ be a finite set of Herbrand interpretations (not necessary minimal) over a first–order language L and let k be the cardinality of $\mathcal{I}$. A tree structure for $\mathcal{I}$ is a triple (V, E, F) where V is the set of nodes of the trees, E is the set of edges and F is a labeling function for the nodes of V into atoms of the Herbrand base of L, $HB_L \cup \{\varepsilon\}$, where ε is a special symbol labeling the root, and the set of atoms labeling each of the k branches of the trees forms a different interpretation in $\mathcal{I}$.*

Given a node v of a tree, we refer to $branch(v)$ as the set of branches of the tree where v occurs. Each branch of $branch(v)$ includes the path from the root to the node v and

from there to one of the leaf nodes descendants of the node v. Informally, $branch(v)$ denotes a subset of interpretations represented by the tree structure. When v is a leaf node, $branch(v)$ denotes one interpretation.

To represent a set of minimal interpretation, a device to maintain minimality when the interpretations are updated is needed. This is accomplished by adding a new component to the above representation. For more explanations, see [FM91].

Definition 10 (Model Tree) *Let $\mathcal{I}$ be a finite set of minimal Herbrand interpretations over a first-order language L. An interpretation (model) tree for $\mathcal{I}$, denoted $\mathcal{MT}(\mathcal{I})$, is a tuple (V, E, F, G) where (V, E, F) is a tree structure for $\mathcal{I}$ and G is a binary relation over V such that $G(x, y)$ iff y is the first node (from the root) in some branch such that the branch x does not contain $F(y)$.*

We are now in the position to present a formal definition of our new data structure.

Definition 11 (Enhanced Model Tree) *Let $< \mathcal{I}, \mathcal{J} >$ be a pair of finite sets of minimal Herbrand interpretations of a disjunctive deductive database DB, such that $\mathcal{I} \leq \mathcal{J}$. An enhanced model tree for $< \mathcal{I}, \mathcal{J} >$, denoted $\mathcal{EMT}(< \mathcal{I}, \mathcal{J} >)$ is a tuple $(V_{\mathcal{I}}, V_{\mathcal{J}\setminus\mathcal{I}}, E_{\mathcal{I}}, E_{\mathcal{J}\setminus\mathcal{I}}, F_{\mathcal{I}}, F_{\mathcal{J}\setminus\mathcal{I}}, G_{\mathcal{I}}, G_{\mathcal{J}\setminus\mathcal{I}})$ which satisfies the following conditions:*

- *$\mathcal{MT}_1 = (V_{\mathcal{I}}, E_{\mathcal{I}}, F_{\mathcal{I}}, G_{\mathcal{I}})$ is a model tree for $\mathcal{I}$.*
- *$\mathcal{MT}_2 = (V_{\mathcal{I}} \cup V_{\mathcal{J}\setminus\mathcal{I}}, E_{\mathcal{I}} \cup E_{\mathcal{J}\setminus\mathcal{I}}, F_{\mathcal{I}} \cup F_{\mathcal{J}\setminus\mathcal{I}}, G_{\mathcal{I}} \cup G_{\mathcal{J}\setminus\mathcal{I}})$ is a tree model for $\mathcal{J}$.*
- *$\mathcal{MT}_1$ is a subtree of $\mathcal{MT}_2$.*

We shall denote by V, E, F, G the unions of $V_{\mathcal{I}} \cup V_{\mathcal{J}\setminus\mathcal{I}}, E_{\mathcal{I}} \cup E_{\mathcal{J}\setminus\mathcal{I}}, F_{\mathcal{I}} \cup F_{\mathcal{J}\setminus\mathcal{I}}, G_{\mathcal{I}} \cup G_{\mathcal{J}\setminus\mathcal{I}}$, respectively.

We shall graphically represent the enhanced model trees in the following way: we shall use boxes to represent nodes from $V_{\mathcal{I}}$ and dashed boxes to represent nodes from $V_{\mathcal{J}\setminus\mathcal{I}}$. In the same way, we shall use arrows to represent edges from $E_{\mathcal{I}}$ and simple lines to represent edges from $E_{\mathcal{J}\setminus\mathcal{I}}$.

Example 1 *The following enhanced model tree represents the pair of sets of minimal models*
$< \mathcal{I}, \mathcal{PD} >=< \{\{a\}, \{b\}, \{c\}\}, \{\{a, q, r\}, \{b, r\}\{c, q, \}\} >$.

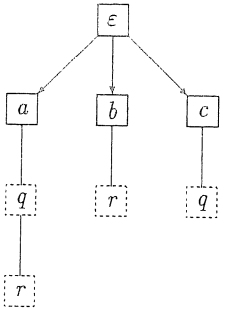

Figure 1: *An enhanced model tree*

We shall use the enhanced tree data strucure as the basis of an incremental evaluation algorithm for $SWFS$. During the computation process, the minimization step, which eliminates unnecessary models, is important. We present below an algorithm, taken from [FM91] which computes an interpretation tree for the set of of minimal interpretations of an arbitrary set represented by a tree structure.

Algorithm 1 Computes a model tree $\mathcal{MT} = (V, E', F, G)$ for $\mathcal{I}$ from a tree structure for $\mathcal{I}_t$, $t = (V, E, F)$ where $\mathcal{I}$ is a set of minimal interpretationbs in $\mathcal{I}_t$.

1. Copy t into $\mathcal{MT}$ removing any node v such that $\exists v'$ ancestor of v and $F(v') = F(v)$. If v is a leaf node then remove from $\mathcal{MT}$ all sibling nodes of v in t and their descendants.

2. For each leaf node l of $\mathcal{MT}$ inspect, bottom-up (from the leaf to the root), the ancestor nodes n of l.

   (a) Compute $S = \{F(x) : x \in path(n, l)\}$.

   (b) For each sibling v of n such that $F(v) \in S$, if there exists a branch l' in $branch(v)$ with all atoms in S, then branch l is not minimal and must be deleted. Otherwise in each branch l' in $branch(v)$ there exists a topmost node w such that $F(w) \notin S$. Add the pair (l, w) to the relation G.

4 Computing SWFS on Enhanced Model Trees

We shall present in this section an alternating fixpoint computation of $SWFS$ based on enhanced model trees. In this respect, we shall modify our alternating fixpoint definition of Ross strong well-founded semantics from [Papp94] in two repects. A first modification is needed to deal with $SWFS$ instead of W^S (Ross strong well-founded semantics), while the second is made in order to obtain a characterization obtained in terms of enhanced model trees.

What we need is to modify the definitions of the operators T_{DB}, Φ_{DB} and W_{DB} from [Papp94] so they can act on enhanced model trees. We shall obtain a fixpoint tree which can be taken to represent the $SWFS$.

First let us define two order relations on the set of enhanced trees.

Definition 12 *Let $\mathcal{EMT}_1 = < V_{\mathcal{I}_1}, V_{\mathcal{J}_1\setminus\mathcal{I}_1}, E_{\mathcal{I}_1}, E_{\mathcal{J}_1\setminus\mathcal{I}_1}, F_{\mathcal{I}_1}, F_{\mathcal{J}_1\setminus\mathcal{I}_1}, G_{\mathcal{I}_1}, G_{\mathcal{J}_1\setminus\mathcal{I}_1} >$ and $\mathcal{EMT}_2 = < V_{\mathcal{I}_2}, V_{\mathcal{J}_2\setminus\mathcal{I}_2}, E_{\mathcal{I}_2}, E_{\mathcal{J}_2\setminus\mathcal{I}_2}, F_{\mathcal{I}_2}, F_{\mathcal{J}_2\setminus\mathcal{I}_2}, G_{\mathcal{I}_2}, G_{\mathcal{J}_2\setminus\mathcal{I}_2} >$ be two enhanced model trees Then*

- *$\mathcal{EMT}_1 \leq \mathcal{EMT}_2$ iff $\mathcal{I}_1 \leq \mathcal{I}_2$.*
- *$\mathcal{EMT}_1 \sqsubseteq \mathcal{EMT}_2$ iff $< \mathcal{I}_1, \mathcal{J}_1 > \sqsubseteq < \mathcal{I}_2, \mathcal{J}_2 >$.*

It is straightforward to show that both the order $\leq$ and $\sqsubseteq$ define a complete lattice structure on the set of enhanced model trees. Now, we shall adapt the definitions of operators to work on enhanced model trees.

We shall consider from now an enhanced model tree to be specified by the set of its branches. Let us denote by $\mathcal{B}$ the set of branches of an enhanced model tree. and by $\mathcal{B}_{\mathcal{I}}$ the set of the branches of the subtree $(V_{\mathcal{I}}, E_{\mathcal{I}})$.

If $(T_i)_{i\in I}$ is an indexed set of transformations which act on enhanced model trees, and $\mathcal{EMT}$ is an enhanced model tree, we shall denote by $\bigotimes_{i\in I} T_i(\mathcal{EMT})$ the tree obtained from $\mathcal{EMT}$ by successively applying all the transformations from the indexed set (order is not important).

Let $\mathcal{EMT} = (\mathcal{B}, \mathcal{B}_{\mathcal{I}})$ be an enhanced model tree specified by the set of its branches. Let B be a branch in $\mathcal{B}_{\mathcal{I}}$ and $c = A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n \wedge \neg C_1 \wedge \ldots \wedge \neg C_m$ be a clause in DB_{inst}. We say that the branch B *fires* the clause c if the following two conditions are fullfilled:

1. $\forall i,\ \exists v \in B,\ F_{\mathcal{I}}(v) = B_i$.
2. $\forall i,\ \nexists v \in V_{\mathcal{I}} \cup V_{\mathcal{J}\setminus\mathcal{I}},\ F(v) = C_i$.

Let

$$\mathcal{EXP}_{c,B}(\mathcal{EMT}) = \begin{cases} \mathcal{MIN}\left(\mathcal{B}, \left((\mathcal{B}_{\mathcal{I}} \setminus \{B\}) \bigcup_{i=1}^{k} \{B \cup \{v_i\}\}\right)\right), & \text{if } B \text{ fires } c \\ \mathcal{EMT} \text{ otherwise} \end{cases} \quad (1)$$

where v_i denotes a new set of nodes such that $F(v_i) = A_i$ and $\mathcal{MIN}$ is the minimization transformation defined by Algorithm 1 of the previous section. Informally, the $\mathcal{EXP}_{c,B}$ transformation expands the terminal node of the branch B by adding to it k descendants, one for each atom in the head of the clause. provided that B triggers c. This expansion step is followed by a minimization step. If B does not trigger c then $\mathcal{EXP}_{c,B}$ leaves $\mathcal{EMT}$ unchanged. We are now ready to present the definition of the enhanced minimal tree version of the T_{DB} operator.

$$T_{DB}^{EMT}(\mathcal{EMT}) = \bigotimes_{B\in\mathcal{B}_{\mathcal{I}}} \left(\bigotimes_{c\in DB_{inst}} \mathcal{EXP}_{c,B}(\mathcal{EMT}) \right) \quad (2)$$

To define the analogue of the Φ_{DB} operator we need to modify the firing condition.

Let $\mathcal{EMT} = (\mathcal{B}, \mathcal{B}_{\mathcal{I}})$ be an enhanced model tree specified by the set of its branches. Let B a branch in $\mathcal{B}$ and $c = A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n \wedge \neg C_1 \wedge \ldots \wedge \neg C_m$ a clause in DB_{inst}. We say that the branch B *p-fires* c if the following conditions are satisfied:

1. $\forall i,\ \exists v \in B,\ F(v) = B_i$.
2. $\forall i,\ \nexists v \in B,\ F(v) = C_i$.

Let

$$\mathcal{PEXP}_{c,B}(\mathcal{EMT}) = \begin{cases} \mathcal{MIN}\left(\left((\mathcal{B} \setminus \{B\}) \bigcup_{i=1}^{k} \{B \cup \{v_i\}\}\right), \mathcal{B}_{\mathcal{I}}\right), & \text{if } B \text{ p-fires } c \\ \mathcal{EMT} \text{ otherwise} \end{cases} \quad (3)$$

$$\Phi_{DB}^{EMT}(\mathcal{EMT}) = \bigotimes_{B\in\mathcal{B}} \left(\bigotimes_{c\in DB_{inst}} \mathcal{PEXP}_{c,B}(\mathcal{EMT}) \right) \quad (4)$$

Now we shall define the iteration of the Φ_{DB}^{EMT} operator. Looking back to the definition of the iteration for the Φ_{DB} operator we note that we have used only the first component of the pair $< \mathcal{I}, \mathcal{J} >$. The same thing happens here. Given an enhanced model tree we "forget" the nodes from $V_{\mathcal{J}\setminus\mathcal{I}}$ and the edges from $E_{\mathcal{J}\setminus\mathcal{I}}$ when we begin to iterate Φ_{DB}^{EMT}. Let $\mathcal{EMT} = (\mathcal{B}, \mathcal{B}_{\mathcal{I}})$ be an enhanced model tree.

$$\Phi_{DB}^{EMT} \uparrow 0(\mathcal{EMT}) = (\mathcal{B}_{\mathcal{I}}, \mathcal{B}_{\mathcal{I}})$$

$$\Phi_{DB}^{EMT} \uparrow (n+1)(\mathcal{EMT}) = \Phi_{DB}^{EMT}(\Phi_{DB}^{EMT} \uparrow n(\mathcal{EMT})).$$

This sequence is monotone on the lattice of enhanced model trees with respect to the order $\leq$ and reaches a limit after a finite number of steps. Let us denote this limit by $\Phi_{DB}^{EMT} \uparrow \infty(\mathcal{EMT})$. Now we present the definition of the W_{DB}^{EMT} operator.

$$W_{DB}^{EMT}(\mathcal{EMT}) = \Phi_{DB}^{EMT} \uparrow \infty(T_{DB}^{EMT}(\mathcal{EMT}))$$

The operator W_{DB}^{EMT} is monotone on the lattice of enhanced model trees with respect to the order $\sqsubseteq$ and so it reaches a limit in a finite number of steps (since the set of enhanced model trees is finite).

Let

$$\mathcal{EMT}_0 = (\emptyset, \{B_{HB_{DB}}\})$$
$$\mathcal{EMT}_{n+1} = W_{DB}^{EMT}(\mathcal{EMT}_n)$$

$B_{HB_{DB}}$ denotes the branch formed by enumerating all the atoms in HB_{DB} (order is not important). Needless to say that we do not actually need to represent this branch in a practical implementation. The sequence $\mathcal{EMT}_n$ reaches a limit $\mathcal{EMT}_\infty$ after a finite number of steps. We shall take this limit to represent the semantics of the database. Let $\mathcal{EMT}_\infty = (\mathcal{B}^\infty, \mathcal{B}_{\mathcal{I}}^\infty)$.

Definition 13 *We shall call the enhanced-tree semantics of DB, ETS(DB), the semantics which assign the following truth values to a ground disjunction D:*

- *true if* $\forall B \in \mathcal{B}_{\mathcal{I}}^{\infty}$, $\exists v \in B$, $F_{\mathcal{I}^{\infty}}(v) = A_j$, *where* A_j *is a ground atom which occurs in* D.

- *false if* $\forall B \in \mathcal{B}^{\infty}$, $\exists v \in B$, $F(v) = A$ *and* A *is a ground atom which occurs in* D.

- *undefined otherwise.*

Theorem 1 (Tree Computation of SWFS) *Let DB be a disjunctive deductive database. Then* $ETS(DB) = SWFS(DB)$.

We shall omit the proof due to space limitations. It can be easy obtained by adapting the proof of Theorem 4.1 from [Papp94] to the terminology of sets of minimal interpretations.
The above theorem establishes an efficient way to compute SWFS bottom-up, by using the enhanced model tree data structure. We conclude with an example.

Example 2 *Let DB be the following disjunctive database:*

$$p \vee a \leftarrow q \wedge r$$
$$q \leftarrow \neg b$$
$$r \leftarrow \neg c$$
$$a \vee b \vee c$$

We present below an enhanced model tree computation

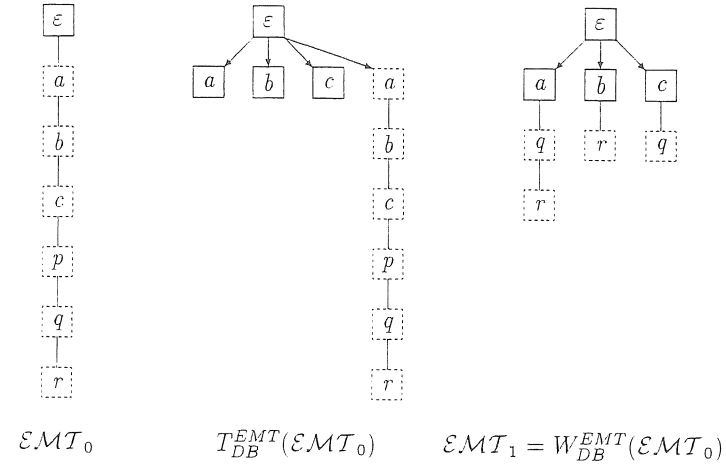

Figure 2: *An enhanced model tree computation*
$\mathcal{EMT}_1$ *is a fixpoint of* W_{DB}^{EMT} *and so it represents* $SWFS$. *Thus* $a \vee b \vee c$ *is true,* p *is false, while* q *and* r *are undefined. The same result would be obtained if we used the top-down definition of SWFS from Section 2.*

5 Conclusions and Future Work

A new data structure derived from Fernández and Minker FM91] *model tree* was defined. This data structure is well-suited for the computation of SWFS, a disjunctive extension of the well-founded semantics which treats disjunction exclusively. An alternating fixpoint computation of SWFS expressed directly in terms of the new data structure is then proposed.

Our future work will include two main directions: the first one is to adapt the enhanced tree computation to other disjunctive extensions of the well-founded semantics. We are currently investigating the adaption of our technique to *DWFS* of [Bar91] and *GDWFS* of [BLM90].

The other area of investigation is to find efficient techniques of implementation. Our idea is to use the RETE algorithm [For82].

References

[ABW88] K. R. Apt. H. Blair and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–104, Los Altos, Ca. 1988. Morgan Kaufmann.

[Bar91] C. Baral. *Issues in Knowledge Representation: Semantics and Knowledge Combination.* PhD. thesis, Dept of Computer Science, University of Maryland, USA, 1991.

[BLM90] C. Baral, J. Lobo and J. Minker. Generalized Disjunctive Well-founded Semantics: Declarative Semantics. In *Proceedings of the Fifth International Symposium on Methodologies for Intelligent Systems*, pages 465–473.

[BD94] S. Brass and J. Dix. Computing a Disjunctive Extension of WFS Based on Abstract Properties. Unpublished manuscript.

[FM91] J. A. Fernández, J. Minker. Bottom-up Evaluation of Hierarchical Disjunctive Databases. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 660–675.

[FM94] J. A. Fernández, J. Minker. Bottom-Up Computation of Perfect Models for Disjunctive Theories. Research Report, University of Maryland.

[For82] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19, pages 17–37.

[GL88] M. Gelfond and V. Lifschitz. The Stable Semantics of Normal Logic Programs. In K. Bowen and R. A. Kowalski editors, *Fifth International Conference on Logic Programming*, 1988, MIT Press.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming,* second edition, Springer-Verlag, 1987.

[LMR92] J. Lobo. J. Minker, A. Rajasekar. *Foundations of Disjunctive Logic Programming*, MIT Press. 1992.

[MR89] J. Minker. A. Rajasekar. Stratification Semantics For Normal Disjunctive Logic Programs. In E. Lusk. R. A. Overbeek, editors. *Proceedings of the North American Conference on Logic Programming*, 1989. MIT Press.

[MD93] M. Müller and J. Dix. On Implementing Semantics for Disjunctive Logic Programs. In L. M. Pereira and A. Nerode, editors. *Proc. of the Second Intl. Workshop on Logic Programming and Non-Monotonic Reasoning*, MIT Press. 1993. pages 43–59.

[NNS91] A. Nerode. R. Ng, V. Subrahmanian. Computing Circumscriptive Deductive Databases. Technical Report TR-91-66. University of Maryland. 1991.

[Prz90a] T. Przymusinski. Three Value Stable Semantics For Disjunctive Logic Programs. In D. H. D. Warren and P. Szeredi. editors. *Seventh International Conference on Logic Programming*, 1990. pages 459–477. MIT Press.

[Prz90b] T. Przymusinski. Stationary Semantics For Disjunctive Logic Programs. In S. Debray and M. Hermenegildo. editors. *Proceedings of the North American Conference on Logic Programming*. 1990. MIT Press.

[Papp94] C. Papp. A Bottom up Reconstruction of the Well-founded Semantics for Disjunctive Logic Programs. In F. Pfenning, editor. *Proceedings of the 5th International Conference On Logic Programming and Automated Deduction (LPAR'94)*, Kiev, July, 1994. Lecture Notes in Artificial Intelligence 822. pages 230–244. Springer–Verlag, 1994.

[Ross90] K. A. Ross. The Well Founded Semantics for Disjunctive Logic Programs. In W. Kim. J.–M. Nicolas and S. Nishio. editors. *Deductive and Object–Oriented Databases*. pages 385–402. Elsevier Science Publishers B. V. (North–Holland). 1990.

[Sei94] D. Seipel. An Efficient Computation of the Extended Generalized Closed World Assumption by Support–for–Negation Sets. In F. Pfenning, editor. *Proceedings of the 5th International Conference On Logic Programming and Automated Deduction (LPAR'94)*. Kiev, July, 1994. Lecture Notes in Artificial Intelligence 822, pages 245–259. Springer-Verlag, 1994.

[ST94] D. Seipel and H. Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In *Proc. Intl. Workshop on the Deductive Approach to Information System and Databases*, 1994.

[VGRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The Well–Founded Semantics for General Logic Programs, *JACM, 38, 3* (1991), pages 620–650.

Mónadas para la Comunicación de Objetos Funcionales

José E. Gallardo **Pablo Guerrero** **Blas C. Ruiz**

Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Pza. El Ejido s/n, 29013–Málaga, España

e–mail: {pepeg,pablito,blas}@lcc.uma.es fax: +34–5–2131397

Resumen

Exponemos un sistema basado en mónadas para la comunicación de procesos funcionales que permite describir sistemas reactivos usando el paradigma orientado a objetos. Los objetos se modelan como procesos (instancias *Haskell* de la mónada *IO* de Launchbury, Peyton Jones y Wadler) que son reducidos en evaluadores separados y pueden comunicarse a través de un manejador de buzones asíncronos vía evaluación no estricta de mensajes, que son a su vez descritos en forma monádica, resultando una notación cercana a la usual de la *POO* (Programacióm Orientada a Objetos); el asincronismo y la evaluación no estricta enriquecen el mecanismo de suspensión de procesos. El comportamiento que caracteriza una clase de objetos se describe por *comprensión de mónadas*, resultando un método flexible y expresivo (como muestran los ejemplos que se acompañan). Se dan algunas ideas sobre cómo ampliar la comunicación monádica para describir un nuevo concepto que denominamos *rendezvous perezoso.*

Palabras clave: comunicación de procesos, mónadas, programación orientada a objetos, *rendezvous perezoso.*

0 Introducción

El paralelismo explícito en lenguajes funcionales es una alternativa al paralelismo implícito o con anotaciones. Los procesos son expresiones que se reducen en evaluadores separados, comunicándose y sincronizando sus evaluaciones con canales *á là CSP* [2] o buzones. En lenguajes con evaluación no estricta, el orden

de los efectos laterales provocados por las operaciones de entrada salida se determina dinámicamente; el problema es modelar tales efectos con el compromiso de mantener la transparencia referencial; así, los principales modelos existentes (*continuaciones* y los basados en las mónadas de Moggi [6, 7]) usan una extensión de *Haskell* [3], pues su sistema de tipos (extensión del de Hindley–Milner) y clases de constructores [4] permiten una descripción de las mónadas por comprensión (aunque es posible utilizar directamente las funciones de la mónada, -tal como propone el comité de *Haskell 1.3* para la mónada *IO*- la notación resultante es más engorrosa). En nuestro trabajo anterior [1] se describió una ampliación del sistema de canales de *Haskell*, que usaba implícitamente continuaciones monádicas y permitía describir con más expresividad la comunicación, pero no es adecuado para sistemas reactivos en general. En el presente trabajo, la descripción de la comunicación usa el paradigma *POO*, donde esencialmente los objetos se caracterizan por una memoria privada y un comportamiento; no es difícil extender el principio de Moggi [6] para capturar la memoria de los objetos en la mónada *IO* de [5]; el sistema de tipos encapsula directamente sus comportamientos con dicha mónada[1].

1 La Mónada de Transformadores de Estados

Una *mónada* es un constructor de datos `M` con dos funciones asociadas (`result, bind`) verificando ciertas propiedades [7]; tales funciones son capturadas en dos clases de constructores `Functor` y `Monad` [4] con objeto de facilitar un uso sobrecargado de éstas, así como la notación por comprensión. Un *transformador de estados* es una función que a partir de un estado (de tipo `s`) proporciona un nuevo estado del mismo tipo y un resultado (de tipo `a`): `type ST s a = s -> (a,s)`. El tipo de dato `ST s a` podemos parcializarlo con respecto al tipo del resultado en la forma `ST s`; esta parcialización produce un constructor de tipos al cual se le asocia una mónada con las definiciones:

```
result x   = \s -> (x,s)
t `bind` f = \s -> f x s' where (x,s') = t s
```

Para representar las operaciones de entrada/salida [5] (necesarias para la comunicación entre objetos), se considera un tipo abstracto de datos `RealWorld` cuyos valores representan el mundo real, en el que se producen los efectos laterales de tales operaciones. Tales estados quedan encapsulados en el sistema, y se ofrece un dato *predefinido* a través del cual se implementan las operaciones de entrada/salida: `data IO a = ST RealWorld a`

[1] el sistema está disponible via *anonymous ftp* en *lcc.uma.es* (150.214.58.71)

2 Primitivas para la Comunicación

Siguiendo la notación de §1, un objeto será un valor del tipo `IO ()`; los objetos reciben mensajes y/o emiten respuestas a éstos, por lo que el comportamiento deberá contener expresiones para la descripción de la comunicación; éstas consisten en la evaluación de una serie de primitivas. Un objeto *servidor* atiende mensajes de varios objetos (puede que simultáneamente); la comunicación entre objetos está basada en buzones (*mailboxes* o nombres globales). Como los tipos de los argumentos y respuestas pueden diferir, la forma natural de representar la sobrecarga de las operaciones básicas es usando clases; así, consideraremos una clase para argumentos y otra para respuestas desde las cuales se crearán instancias para los tipos estándar usuales (`Int`, `Bool`, tuplas, listas, ...):

```
class Arg arg where
  sendMsg     :: ObjName -> MsgId -> Selector -> arg -> IO ()
  -- mete en el buzón ObjName la terna (MsgId,Selector,arg)
  receiveMsg  :: ObjName -> IO (MsgId,Selector,arg)
  -- lee la terna desde el buzón
class Resp resp where
  sendResp    :: MsgId -> resp -> IO ()
  receiveResp :: MsgId ->         IO resp
```

La generación de objetos puede obtenerse con ayuda de un esquema de la forma

```
new :: (ObjName -> a -> IO a) -> ObjName -> a -> IO ()
new comp objName st = [ () | nst <- comp objName st,
                             ()  <- new comp objName nst ]
```

donde el argumento `comp` captura el comportamiento de una *clase* de objetos: una abstracción del concepto de clase de la *POO*. Consideremos a modo de ejemplo una clase abstracta de objetos formada por puntos bidimensionales capaces de recibir (y responder) mensajes tales como: `punto."GetX"!()` (devuelve la coordenada `x`), `punto."Traslada"!(a,b)` (traslada el punto en la dirección `(a,b)`). El siguiente objeto dialoga con una instancia de la clase:

```
user = new comp "User" () where
 comp _ _ = [ () | y <-"P1"."GetY"!(),
                   ()<-if y > 0 then  "P1"."Traslada" ! (0,-2*y)
                       else  [()]]
```

A través de una primitiva `createProcess :: a -> IO ()`, la evaluación de una expresión tal como `createProcess p` generará un nuevo proceso que evalua la expresión `p` en un evaluador separado; una lista de objetos `objs` puede activarse evaluando `foldl bind [()] (map createProcess objs)`. Si la recepción de un mensaje necesita una respuesta inmediata podemos utilizar el generador `arg <- obj ? f`, en el cual el objeto `obj` recibe un mensaje de argumento `arg` y responde con `f arg`. La operación `(?)` (recepción de mensaje/envío de respuesta) puede obtenerse como composición en la forma

```
(?) :: (Resp resp, Arg arg) => ObjName -> (arg -> resp) -> IO arg
objName ? f = [ arg | (emisor, sel, arg) <- receiveMsg objName ,
                                        () <- sendResp emisor (f arg)]
```

Los tipos de los argumentos (y los de las respuestas) de los distintos mensajes que puede aceptar un objeto pueden ser diferentes. Esto, en un lenguaje sin tipos, no plantea problemas; por el contrario, en un lenguaje funcional con tipos, como *Haskell*, podemos[2] prever que es el selector del mensaje el que determina tanto el tipo del argumento del mensaje como el tipo de la respuesta; por ello incorporamos una nueva primitiva para determinar el selector del próximo mensaje a tratar, que podemos invocar a través del generador: `sel <- nextMsgSelector obj`. Mediante esta primitiva es fácil escribir funciones comportamiento como la siguiente, que corresponde a la *clase* de puntos bidimensionales:

```
c_punto punto (x,y) = nextMsgSelector punto `bind` \sel ->
   case sel of
      "GetX"     -> [ (x,y) | ()<- punto? \_ -> x ]
      "GetY"     -> [ (x,y) | ()<- punto? \_ -> y ]
      "Traslada" -> [ (x+a,y+b) | (a,b)<- punto? \_ -> () ]
```

Para la *emisión de un mensaje* utilizamos el generador: `res <- obj.selec! arg`, el cual provoca el envío al objeto `obj` (a través de cierto buzón) de un mensaje con selector `selec` y argumento `arg`. La acción `(!)` (emisión de un mensaje/recepción de la respuesta) queda descompuesta en la forma

```
(!) :: (Resp resp, Arg arg) => (ObjName,Selector) -> arg -> IO resp
(objName,selector) ! arg =
     [ resp | msgId <- newMsg,
              ()    <- sendMsg objName msgId selector arg,
              resp  <- receiveResp msgId ]
```

donde, ya que un objeto *emisor* puede enviar *varios mensajes* al mismo objeto receptor, necesitamos una nueva operación (`newMsg`) que captura en una mónada un *nuevo* identificador de mensaje que también determina el objeto emisor y será utilizado para responder correctamente al objeto.

Exponemos con las notaciones anteriores la solución al clásico *problema de los filósofos* [2] en la que aparece un objeto por cada filósofo, otro objeto por cada tenedor, y un objeto `vigilante` que controla el número de filósofos que pueden solicitar la operación de tomar tenedores para evitar la situación de *deadlock*. Todos los filósofos tienen el mismo comportamiento:

```
c_filo nf k = [ k | () <- putString ("\n-pensando "++ nf),
                    () <- "vigilante".sentarse!(),
                    () <- (ten (iz k)).tomar_tenedor!(),
```

[2]En principio no se considera la sobrecarga de selectores para un mismo objeto receptor


```
                    () <- (ten (de k)).tomar_tenedor!(),
                    () <- putString ("\n-comiendo "++show k),
                    () <- (ten (iz k)).dejar_tenedor!(),
                    () <- (ten (de k)).dejar_tenedor!(),
                    () <- "vigilante".levantarse!() ]
  where iz k  = (k-1) `mod` n_fil
        de k  = k
        ten k = "ten "++ show k
```

Los tenedores se comportan como semáforos binarios, mientras que el `vigilante` se comporta como un semáforo general, con un valor inicial igual a `n_fil-1`, siendo `n_fil` una constante del programa:

```
filO      = new c_filo "descartes" 0
...
vigilante = new c_sem "vigilante" (n_fil-1,[],n_fil-1)
tenO      = new c_sem "ten 0" (1,[],1) -- semaforo binario
...
main      = par [vigilante, tenO, ..., filO, ...]
tomar_tenedor = "wait"; dejar_tenedor = "signal"
sentarse      = "wait"; levantarse    = "signal"
```

Aquí, un semáforo general es un objeto cuyo estado es `(val, waitingList, n_max)`, donde `n_max` es el número máximo de procesos (objetos) que lo pueden tomar, `val` es su valor actual (`0 <= val < n_max`), y `waitingList` es una lista que permite implementar una política FIFO. Su comportamiento vendrá dado por

```
c_sem objName e = [ ne | msg <- receiveMsg objName,
                         ne  <- c_sem' msg e ]
c_sem' (emisor,"signal",()) (k,os@(o:ros),top)
  | k == top  = error "___________"
  | os == []  = [ (k+1,[],top) | () <- sendResp emisor ()]
  | otherwise = [ (k,ros,top)  | () <- sendResp o (),
                                 () <- sendResp emisor ()]
c_sem' (emisor,"wait",()) (k,os@(o:ros),top)
  | k  == 0   = [ (0,os++[emisor],top) ]
  | os == []  = [ (k-1,[],top) | ()<- sendResp emisor ()]
  | otherwise = [ (k-1, ros++[emisor],top ) | ()<- sendResp o ()]
```

3 Rendezvous Perezoso

Un problema con las primitivas para la comunicación es la característica estricta de las operaciones de envío de mensajes y respuestas; por ejemplo, si

```
p1 = [ () | res <- "obj"."""![1..],
```

```
                () <- putStrLn (show (take 3 res::[Int])) ]
p2 = [ () | xs <- "obj" ? (take 10 . (map (^2))),
                () <- putStrLn (show (take 8 xs::[Int])) ]
```

el objeto p1 no llega a leer la respuesta enviada por el objeto p2, ya que el envío del argumento [1..] no termina (tal como describimos la primitiva (!) en §2, la acción receiveResp no llega a realizarse pues la operación sendMsg es estricta); por el contrario, la recepción del argumento de un mensaje y la recepción de una respuesta son perezosas. Pero ni p1 ni p2 terminarían si hubiéramos escrito

```
p2 = [ () | _ <- "obj"? (\_ -> [1..]) ]
```

Si las operaciones de envío de mensajes y respuestas no fueran estrictas, podríamos hablar de una situación de *rendezvous perezoso*, ya que el proseguir con la siguiente acción antes de completar la respuesta a cierto mensaje puede provocar, durante la llamada recursiva a la función comportamiento, el poder atender a otro objeto sin necesidad de responder al anterior. Esto puede conseguirse esencialmente de dos formas: (i) manteniendo una estructura de datos para los objetos emisores (en la cual se encolan éstos para que se les pueda responder posteriormente), que es la técnica que se usó al describir en §2 el comportamiento c_sem de los semáforos; (ii) cambiando sustancialmente la implementación de algunas primitivas con objeto de poder suspender la evaluación de la respuesta hasta que ésta sea necesaria (i.e., por demanda), pues esto no se consigue directamente a través de la primitiva interleaveST de [5]; una posibilidad es mantener en la estructura de los buzones referencias a funciones *suspendidas*, las cuales se activarán por demanda del objeto emisor.

Referencias

[1] J.E. Gallardo, P. Guerrero y B.C. Ruiz. Mónadas y Procesos Funcionales Comunicantes. GULP–PRODE'94, Peñíscola, España, p. 266–280, Sep. 1994.

[2] CAR. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[3] P. Hudak, SL. Peyton Jones y P. Wadler (Eds.). Report on the Prog. Lang. Haskell, Version 1.2. *SIGPLAN 27,5*, Agosto 1992.

[4] Mark P. Jones. A System of Constructor Classes: Overloading and Implicit Higher–Order Polymorphism. *6th ACM Conference on FPCA*, Junio 1993.

[5] John Launchbury y Simon L. Peyton Jones. Lazy Functional State Threads. *ACM Conference on Programming Languages Design and Implementation*, Junio, 1994.

[6] E. Moggi. Notions of Computation and Monads. *Information and Computation.*, 93:55–92, 1991.

[7] P. Wadler. The Essence of Functional Programming. *19th Annual ACM Symposium on Principles of Prog. Lang., Albuquerque*, p. 1–14, Enero 1992.

La potencia expresiva de los catamorfismos

Carlos Gregorio, Manuel Núñez y Pedro Palao

Departamento de Informática y Automática

Universidad Complutense de Madrid

e-mail: cgregor@eucmax.sim.ucm.es, {manuelnu,ecceso}@dia.ucm.es

Abstract

Es conocido que los catamorfismos no son cerrados bajo composición y que no todo lo computable es un catamorfismo. Esto lleva a cuestionarse qué clase de funciones puede definirse con la composición de catamorfismos. En la primera parte de este artículo demostramos que sólo con la composición de dos catamorfismos se obtiene todo lo computable. La segunda parte trata de resolver ciertos problemas en la definición usual de catamorfismo. Para ello introducimos los *catamorfismos constructivos* y demostramos que dan lugar a las funciones recursivas primitivas.

Keywords: Programación funcional, tipos de datos algebraicos, catamorfismos, comunidad Squiggol, computabilidad, recursión primitiva.

1 Introducción

La recursión en los lenguajes funcionales es la fuente de su potencia expresiva y también de la tediosidad de los razonamientos formales. Como consecuencia, la comunidad *Squiggol* ha introducido ciertas formas limitadas de recursión, entre las que se encuentran los homomorfismos [1] [2], catamorfismos [5], paramorfismos [4], etc. La simplicidad de todos ellos permite deducir gran cantidad de propiedades que se convierten en herramientas muy útiles cuando intentamos razonar sobre funciones definidas con estos esquemas.

En este trabajo nos vamos a centrar en el estudio de los catamorfismos. Es natural que, a cambio de la facilidad de razonamiento, pierdan algo del poder expresivo de la recursión general. Por ejemplo, hay operaciones que no se pueden expresar como catamorfismos (vease [3]) y la composición de catamorfismos no tiene por qué ser un catamorfismo. Esta disminución de la potencia expresiva y su interacción con la composición son aspectos que no están estudiados de manera sistemática.

En este trabajo nos centramos en el estudio de la potencia expresiva del cálculo generado por la composición y los catamorfismos. La primera parte del artículo estudia estas dos operaciones dentro del marco de los catamorfismos para estructuras

de datos convencionales. Demostramos el siguiente resultado: todas las funciones computables se pueden escribir como la composición de dos catamorfismos.

Las demostraciones de estos resultados muestran la falta de *constructibilidad* de los catamorfismos, al no imponerse ninguna restricción sobre la forma que han de tener sus componentes. En la siguiente parte de este trabajo realizamos un estudio para una clase de catamorfismos extendidos pero *construibles*. El resultado principal de esta sección muestra que todas las funciones recursivas primitivas se pueden definir constructivamente con composición y catamorfismos extendidos.

2 Catamorfismos al modo clásico

Definición 1 *Sean $\mathcal{C}$ una clase de funciones sobre listas, $f_{:}\colon \alpha \times [\alpha] \to \beta$ una función de $\mathcal{C}$ y $f_{[]}$ un elemento de β. Se dice que $f\colon [\alpha] \to \beta$ es un catamorfismo sobre $\mathcal{C}$ definido en función de $f_{[]}$ y $f_{:}$ si*

$$f(x : xs) = f_{:}(x, f(xs)), \qquad f([]) = f_{[]}.$$

Denotamos a f por sus componentes como $(\!|f_{[]}, f_{:}|\!)$. □

Esta definición es la dada en [5] e implica que la clase de funciones computables sobre listas es cerrada bajo catamorfismos. Es sabido que no todas las funciones computables sobre listas son catamorfismos (vease [3] para un ejemplo). La siguiente proposición caracteriza hasta que punto llega esta diferencia.

Proposición 1 (Evaluación en cabeza) *Sea $\mathcal{C}$ la clase de funciones que contiene a las funciones computables sobre listas. Sea f una función de $\mathcal{C}$ con dominio las listas de un tipo α que contenga al menos dos elementos distintos a_1 y a_2. Entonces, existen dos catamorfismos $g, h \in \mathcal{C}$ tales que $f = h \circ g$.*

Demostración Definimos $g = (\!|g_{[]}, g_{:}|\!)$ y $h = (\!|h_{[]}, h_{:}|\!)$, en donde

$$\begin{array}{l}
g_{[]} = [a_1, a_2, a_2, a_2] \\
g_{:}(a_1, [a_1, a_2, a_1, a_2] : xs) = [a_1, a_2, a_2, a_2, a_1, a_1, a_2, a_1] : xs \\
g_{:}(a_1, [a_1, a_2, a_1, a_1] : xs) = [a_1, a_2, a_2, a_2, a_1, a_1, a_1, a_1] : xs \\
g_{:}(x, [a_1, a_2, a_1, y] : xs) = [a_1, a_2, a_2, a_2, x, y] : xs \\
g_{:}(x, [a_1, a_2, a_2, a_2] : xs) = [a_1, a_2, a_1, x] : xs \\
h_{[]} = [] \\
h_{:}(a_1, a_2 : ys) = (f \circ d)(a_1 : a_2 : ys), \qquad \text{esPar}(\text{longitud}(ys)) \\
h_{:}(x, ys) = x : ys
\end{array}$$

Las funciones auxiliares d y d' son

$$\begin{array}{ll}
d([a_1, a_2, a_1, x] : ys) = x : d'(ys) & d([a_1, a_2, a_2, a_2] : ys) = d'(ys) \\
d'([a_1, a_1, a_1, a_1] : ys) = a_1 : a_1 : d'(ys) & d'([a_1, a_1, a_2, a_1] : ys) = a_1 : a_2 : d'(ys) \\
d'([x, y] : ys) = x : y : d'(ys) & d'([]) = [].
\end{array}$$

Los valores donde estas funciones no están definidas son irrelevantes para demostrar que $f = h \circ g$. □

Con la demostración de esta proposición queda claro por qué el concepto de catamorfismo, como se define usualmente, es demasiado general para estudiar su potencia expresiva: no hay limitaciones a la forma de las funciones con las que se construye el catamorfismo.

La hipótesis sobre el dominio α de la lista es fundamental para demostrar la proposición. Las listas sobre un tipo con un sólo elemento son isomorfas a los números naturales. Es fácil ver que la composición de catamorfismos sobre números naturales da lugar a funciones inyectivas o cíclicas. No tenemos una caracterización completa de qué subconjunto de estas funciones es expresable como composición de catamorfismos.

Cuando el tipo tiene una estructura recursiva más rica, no existe ninguna limitación provocada por la cardinalidad del tipo base. Vamos a estudiar los árboles como un ejemplo canónico.

Definición 2 *Sean $\mathcal{C}$ una clase de funciones sobre árboles, $f_{/\backslash}\colon \text{Arbol } \alpha \times \alpha \times \text{Arbol } \alpha \to \beta$ una función de $\mathcal{C}$ y $f_{\triangle}$ un elemento de β. Se dice que $f\colon \text{Arbol } \alpha \to \beta$ es un catamorfismo sobre $\mathcal{C}$ definido en función de $f_{\triangle}$ y $f_{/\backslash}$ si*

$$f(xs/a \setminus ys) = f_{/\backslash}(f(xs), a, f(ys)), \qquad f(\triangle) = f_{\triangle}.$$

De nuevo denotamos a f por $(\!|f_{\triangle}, f_{/\backslash}|\!)$. □

Proposición 2 (Evaluación en cabeza) *Sea $\mathcal{C}$ la clase de funciones que contiene a las funciones computables sobre árboles. Para cualquier función $f\colon \text{Arbol } \alpha \to \beta$ de $\mathcal{C}$ existen dos catamorfismos $g, h \in \mathcal{C}$ tales que $f = h \circ g$.*

Demostración Sea a un elemento cualquiera de α —no es necesario si α es vacio. Definimos $g = (\!|g_{\triangle}, g_{/\backslash}|\!)$ y $h = (\!|h_{\triangle}, h_{/\backslash}|\!)$, donde

$$\begin{array}{ll}
g_{\triangle} = \triangle/a \setminus (\triangle/a \setminus \triangle) & g_{/\backslash}(A, y, B) = \triangle/a \setminus (\text{rot}(A)/y \setminus \text{rot}(B)) \\
h_{/\backslash}(\triangle, x, \triangle) = \triangle/x \setminus \triangle & h_{/\backslash}(\triangle, x, A) = (f \circ d)(\triangle/x \setminus A) \\
h_{/\backslash}(A, x, B) = A/x \setminus B & h_{\triangle} = \triangle \\
\text{rot}(\triangle) = \triangle & \text{rot}(A/x \setminus B) = B/x \setminus A \\
d(\triangle/x \setminus A) = d'(A) & d(A/x \setminus \triangle) = d'(A) \\
d'(\triangle/x \setminus \triangle) = \triangle & d'(A/x \setminus B) = d(A)/x \setminus d(B)
\end{array}$$

La conclusión de la proposición se demuestra por inducción estructural. Es necesario demostrar antes que $(d \circ \text{rot} \circ g)(A) = A$, haciendo uso de que $g(A) \neq \triangle$ para todo A, y que si $A \neq \triangle$ entonces $\text{rot}(A) \neq \triangle$. □

3 Catamorfismos constructivos

En esta sección vamos a limitar la forma en que se pueden construir los catamorfismos para evitar el problema de la falta de una restricción sobre las funciones que definen un catamorfismo.

Definición 3 *La función* $h\colon \overline{\mathbb{N}}^n \to \overline{\mathbb{N}}$ *es un* catamorfismo generalizado *(a partir de ahora simplemente catamorfismo) sobre la clase de funciones* $\mathcal{C}$ *si existen funciones* $g_i\colon \overline{\mathbb{N}}^{n-1} \to \overline{\mathbb{N}}$ $(i = 1..n)$ *y* $g\colon \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ *en* $\mathcal{C}$ *tales que*

$$\begin{cases} h(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) = g_i(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_n) \\ h(x_1 + 1, \ldots, x_n + 1) = g(h(x_1, \ldots, x_n)) \end{cases}$$

con la restricción sobre las g_i *de ser compatibles entre ellas: para todo natural* i, j *y para toda secuencia de argumentos* $\bar{x}, \bar{y}, \bar{z}$ *tales que* $|\bar{x}| = i - 1$, $|\bar{x}| + |\bar{y}| = j - 2$ *y* $|\bar{x}| + |\bar{y}| + |\bar{z}| = n - 2$, *se tiene que* $g_i(\bar{x}, 0, \bar{y}, 0, \bar{z}) = g_j(\bar{x}, 0, \bar{y}, 0, \bar{z})$. *Denotaremos a* h *como* $(\!|g_1, \ldots, g_n; g|\!)$. □

La característica principal de esta definición es la generalización de los catarmorfismos sobre naturales a funciones de más de un argumento.

Definición 4 *Definimos la clase de funciones* $\mathcal{C}^\omega$ *como el menor conjunto que contiene al cero (0), el sucesor* $(+1)$ *y es cerrada bajo composición y esquema de catamorfismos generalizado.* □

Obsérvese la parquedad de recursos en la definición de esta clase.

Proposición 3 *La clase de funciones* $\mathcal{C}^\omega$ *es justamente la clase de las funciones recursivas primitivas.*

Demostración Para demostrar la inclusión directa basta ver que toda función construida como catamorfismo se puede expresar con los esquemas de las funciones r.p. Necesitaremos la función mínimo de n argumentos (min), la igualdad con 0 $(0 \doteq)$, y el condicional **if** $\cdots$ **then** $\cdots$ **else**, fácilmente definibles como r.p.

Sea $F = (\!|G_1, \ldots, G_n; H|\!)$, en donde de cada una de sus componentes ya hemos demostrado que son funciones r.p. Definimos $f'(\bar{x}) = f(\min(\bar{x}), \bar{x} \dot{-} \min(\bar{x}))$, donde la resta se aplica sobre cada elemento de $\bar{x}$. Definimos f por recursión primitiva a partir del caso base

$$\begin{aligned} g(\bar{y}) = \ & \textbf{if } (0 \doteq)(y_1) \textbf{ then } G_1(y_2, \ldots, y_n) \\ & \textbf{else if } (0 \doteq)(y_2) \textbf{ then } G_2(y_1, y_3, \ldots, y_n) \\ & \qquad \cdots \\ & \textbf{else } (0 \doteq)(y_n) \textbf{ then } G_n(y_1, \ldots, y_{n-1}) \end{aligned}$$

y del caso recursivo $h(x, \bar{y}, z) = H(z)$. Se demuestra por inducción sobre el $\min(\bar{x})$ que $F(\bar{x}) = f'(\bar{x})$.

Para obtener el resultado inverso hay que construir las proyecciones y simular el esquema de recursión primitiva. Como el esquema de catamorfismos generalizado es bastante limitado necesitamos introducir todas las funciones de la figura 1.

Utilizando sólo las funciones constantes podemos definir las proyecciones

$$\pi_i^n(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) = \begin{cases} \pi_i^{n-1}(x_1, \ldots, x_{i-1}, x_{i-1}, \ldots, x_n), & \text{si } i < j \\ k_0^{n-1}(x_1, \ldots, x_{i-1}, x_{i-1}, \ldots, x_n), & \text{si } i = j \\ \pi_{i-1}^{n-1}(x_1, \ldots, x_{i-1}, x_{i-1}, \ldots, x_n), & \text{si } i > j \end{cases}$$

$$\pi_i^n(\bar{x} + 1) = (\pi_i^n(\bar{x})) + 1.$$

Constantes: $k = (k - 1) + 1$

Identidad: $\mathrm{id}(0) = 0$, $\mathrm{id}(x + 1) = \mathrm{id}(x) + 1$

Funciones constantes: $k_n(0) = n$, $k_n(x + 1) = \mathrm{id}(k_n(x))$

Funciones constantes generales: $k_n^m(\bar{x}, 0, \bar{y}) = k_n^{m-1}(\bar{x}, \bar{y})$, $k_n^m(\bar{x} + 1) = \mathrm{id}(k_n^m(\bar{x}))$

Suma de una constante: $(n+)(0) = n$, $(n+)(x + 1) = ((n+)(x)) + 1$

Suma: $0 + y = y$, $x + 0 = x$, $(x + 1) + (y + 1) = (2+)(x + y)$

Resta restringida: $0 \dot{-} y = 0$, $x \dot{-} 0 = x$, $(x + 1) \dot{-} (y + 1) = \mathrm{id}(x \dot{-} y)$

Producto por una constante: $(n*)(0) = 0$, $(n*)(x + 1) = (n+)\big((n*)(x)\big)$

Valores booleanos: $\mathrm{true} = 1$, $\mathrm{false} = 0$

Igual a 0: $(0 \doteq)(0) = \mathrm{true}$, $(0 \doteq)(x + 1) = k_{\mathrm{false}}((0 \doteq)(x))$

Negación: $\neg(x) = k_1(x) \dot{-} \mathrm{id}(x)$

Igualdad: $x \doteq 0 = (0 \doteq)(x)$, $0 \doteq y = (0 \doteq)(y)$, $(x + 1) \doteq (y + 1) = \mathrm{id}(x \doteq y)$

Mayor: $x > 0 = \neg(0 \doteq)(x)$, $0 > y = k_{\mathrm{false}}(y)$, $(x + 1) > (y + 1) = \mathrm{id}(x > y)$

Menor: $x < y = \pi_2^2(x, y) > \pi_1^2(x, y)$

Condicional: **if** c **then** x **else** $y \;=\; c \mathbin{\bar{*}} y + (\neg c) \mathbin{\bar{*}} x$
donde $0 \mathbin{\bar{*}} x = x$, $x \mathbin{\bar{*}} 0 = 0$, $(x + 1) \mathbin{\bar{*}} (y + 1) = k_0(x \mathbin{\bar{*}} y)$

Potenciación simultanea $(q_1^{x_1} \cdots q_n^{x_n})$: $q_1^{x_1} \cdots q_i^0 \cdots q_n^{x_n} = q_1^{x_1} \cdots q_{i-1}^{x_{i-1}} q_{i+1}^{x_{i+1}} \cdots q_n^{x_n}$,
$q_1^{x_1+1} \cdots q_n^{x_n+1} = q_1 * \cdots * q_n * (q_1^{x_1} \cdots q_n^{x_n})$

Codificación: $\langle x_1, \ldots, x_n \rangle = p_1^{x_1} \cdots p_n^{x_n}$

Módulo: $0 \bmod n = 0$, $(x + 1) \bmod n = h_n(x \bmod n)$
donde $h_n(x) = \textbf{if } x \doteq n - 1 \textbf{ then } 0 \textbf{ else } y + 1$

Divisibilidad: $n|x = x \bmod n \doteq 0$

Producto por un racional constante $\frac{m}{n}(x) = \lfloor \frac{mx}{n} \rfloor$ con $m > n$:
$\frac{m}{n}(0) = 0$, $\frac{m}{n}(x + 1) = h_{mn}(\frac{m}{n}(x))$
donde $h_{mn}(x) = \textbf{if } x \bmod m \doteq n - 1 \textbf{ then } x + (m - n + 1) \textbf{ else } x + 1$

Producto por un racional constante $m < n$: $\frac{m}{n}(x) = \frac{m+n}{n}(x) - \mathrm{id}(x)$

Quitar un primo p: $(x)//p = h_p(x, x + x)$ donde $h_p(0, x) = x$,
$h_p(x + 1, y + 1) = h'_p(h_p(x, y))$, $h'_p(x) = \textbf{if } p|x \textbf{ then } \frac{1}{p}(x) \textbf{ else } x$

Logaritmo: $\lfloor \log_p(x) \rfloor = \big((h(x)//p) \dot{-} 1\big)/p$
donde $h(0) = 0$, $h(x + 1) = h'(h(x))$
$h'(x) = \textbf{if } x \doteq 0 \textbf{ then } p^{p-1}$
$\textbf{else if } p|x \textbf{ then } x/p$
$\textbf{else } (ph''(x) + 1)p^{(p-1)p^{h''(x)}-1}$
$h''(x) = (x \dot{-} 1)/p + 1$

Componente de una tupla de longitud n:
$x|_i = \log_{p_i}\big((x)//p_1// \cdots //p_{i-1}//p_{i+1}// \cdots //p_n\big)$

Figura 1: Algunos catamorfismos generalizados

Queda simular el esquema de recursión primitiva que se puede conseguir con ayuda de la función de codificación de n números $a_1, \ldots, a_n$, que denotaremos por $\langle a_1, \ldots, a_n \rangle$, y de la operación $|_i$ para extraer la componente i-ésima. Supongamos f de $n+1$ argumentos definida por r.p. a partir de las funciones g y h, de las que ya sabemos que están en $\mathcal{C}^\omega$. Definimos $F'(x, \bar{y}) = F(x, \bar{y} + x)|_{n+2}$ y $F = (\!|G_1, \ldots, G_{n+1}; H|\!)$, en donde $G_1(\bar{y}) = \langle 0, \bar{y}, g(\bar{y}) \rangle$ y

$$H(x) = \langle x|_1 + 1, x|_2, \ldots, x|_n, h(x|_1 + 1, x|_2, \ldots, x|_{n+2}) \rangle$$

El resto de las G_i se definen de forma que sean compatibles con G_1. Para demostrar que $f(x, \bar{y}) = F'(x, \bar{y})$, hay que ver por inducción sobre x que $F(x, \bar{y} + x) = \langle x, \bar{y}, f(x, \bar{y}) \rangle$. □

4 Conclusiones

En este trabajo hemos expuesto los primeros resultados de un estudio sistemático sobre la interrelación entre dos esquemas de definición de funciones que son muy utilizadas en la comunidad *Squiggol:* la composición y los catamorfismos. En la primera parte del trabajo hemos demostrado que todo lo computable se puede expresar como composición de dos catamorfismos. El estudio de estas demostraciones resalta la falta de constructibilidad de los catamorfismos usuales. En la segunda parte del trabajo hemos introducido unos catamorfismos que solucionan este problema sin perder la potencia expresiva: demostramos que se pueden obtener todas las funciones recursivas primitivas.

Referencias

[1] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, volume F36. Spring Verlag. NATO ASI Series, 1987.

[2] R. S. Bird. Lectures on constructive functional programming. In *Constructive Methods in Computer Science*, volume F55. Spring Verlag. NATO ASI Series, 1989.

[3] Luis Antonio Galán, Manuel Núñez, Cristobal Pareja, and Ricardo Peña. Non homomorfic reductions of data structures. *GULP-PRODE*, 2:393–407, 1994.

[4] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[5] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, 1991.

TRANSFORMATION AND SYNTHESIS

A Tabulation Transformation Tactic Using Haskell Arrays*

Cristóbal Pareja-Flores†, Ricardo Peña†
and J. Ángel Velázquez-Iturbide‡

Abstract

In this paper we propose a transformation tactic that provides general tabulation of functional algorithms. The tactic can be considered a generalization of the *tupling* tactic proposed by other authors [Pet84, Chi90]. We can afford call dependency graphs in which variable size *cuts* can be defined. The tables are dynamically created and their sizes determined at execution time. For this purpose, we use the data type *array* present in the functional language Haskell [HPJW92]. We also show how to apply the tactic with languages lacking this feature. We remain all the time in the functional paradigm, and all the proposed transformations are carried out at the programmer level. Finally, some significative examples are developed using the tactic.

Keywords: functional programming, tabulation, memoization, tupling, program transformation, dynamic programming, Haskell.

1 Introduction

Tabulation is a well-known technique to avoid redundant computations of recursive functions. It frequently happens that a recursive function is invoked more than once to solve the same subproblem. This situation is typical when dealing with problems solved by the *dynamic programming* method. The tabulation technique consists of storing in a table the result and the arguments of an invocation when the function is called for the first time to solve a particular subproblem. Were the function invoked again with the same arguments, the result would be obtained from the table instead of doing the calculation again.

The subject has been studied by several authors. In [Bir80] the main elements of the technique are identified: first, the *dependency* (acyclic directed) *graph* of

*This work was partially supported by CICYT projects TIC91-0111 and TIC95-0967-C02.

†Departamento de Informática y Automática, Universidad Complutense de Madrid, 28040 Madrid, Spain. e-mails: {cpareja,ricardo}@dia.ucm.es

‡Departamento de Lenguajes y Sistemas Informáticos, Universidad Politécnica de Madrid, 28660 Boadilla del Monte, Madrid, Spain. e-mail: avelazquez@fi.upm.es

recursive calls developed during a function invocation is studied; then, an order to solve the subproblems is established in such a way that, when a subproblem is needed, it is already solved and tabulated; finally, an iterative program solves the subproblems following such an order. The main concern of that paper is to study the shape of the dependency graph to be able to decide in advance the exact amount of space that will be required by the iterative program. In some complex cases, the author recommends to use the original recursive function, together with a global table providing space enough to keep the results of *all* subproblems. This is called *exact tabulation* in that paper. We will refer to it here as *total tabulation.*

Cohen [Coh83] identifies four classes of redundancy by studying what he calls *descent conditions* and gives efficient imperative schemes for each class. In [Pet84] the main emphasis is put, not in the tabulation itself, but in developing a transformation technique to avoid redundant computations. The technique is called *tupling* and consists of defining a sequence of *cuts* in the dependency graph of the original function in such a way that every cut (a cut is a tuple of invocations) can be computed in terms of the following one. Then a *tupled* linear recursive function is defined to compute the cuts. The original function can be easily expressed in terms of the tupled one. For the strategy to be applied, it is required all the cuts to have the *same number* of nodes. The tuple representing the cut plays the role of a table tabulating the results of the previously solved subproblems. So, in the terminology we use in this paper, tupling is equivalent to *partial tabulation* using a fixed size table. Other relevant works on redundant computation removal, tabulation, and tupling are [Coh83, Hug85, Chi90, Kho90].

In this paper we propose a transformation tactic to produce a tabulated version of a recursive function with redundant calls, in which the sizes of the cuts can be determined at execution time. One extreme case of the technique is *total tabulation*, used when the dependency graph does not admit clear cuts. The other extreme is *tupling*, used when clear cuts with a fixed (and small) number of nodes can be defined in the dependency graph. In the general case, we have *partial tabulation* in which a variable size table is used to record the results of a cut. Other differences with respect to some of the previously mentioned approaches are: 1) we do not leave in any respect the functional paradigm, and 2) the proposed transformations are carried out by the programmer, not by the compiler.

For both purposes, we use the data structure *array* present in the functional language Haskell [HPJW92]. Arrays in Haskell may be created, in a monolithic way, by using the function `mkArray` that has as one of its parameters a *creation function* mapping indexes into values. In our tactic, the original function is transformed into the creation function of an array, and array entries are made dependent on another entries of a different (or the same) array, in a recursive way. Lazy evaluation guarantees that only the needed entries are computed. They are computed on demand starting at the first required value. Updating will be almost never required in this tabulation technique, so the compiler will only maintain a single copy of the current array and, consequently, the constant access time of imperative arrays is

retained by the approach.

The organization of the paper is as follows: after this introduction, in section 2 we prove a theorem dealing with the transformation of the recursive version of a function into a totally tabulated version of it. In section 3, the concept of *cut* is introduced and the original recursive function is transformed into a partial tabulated version. This tactic allows to reduce the amount of table space used, since the size of the table depends in general of the current cut being processed. Both tabulation tactics are based on the rules of the *fold/unfold* system [BD77]. In section 4, some significative examples are developed using the tactic. Finally, in section 5 it is shown how the tactic can be applied to languages in which the array facility is missing.

2 Total tabulation

In Haskell [HPJW92], the data type `Array` is defined as follows:

```
data (Ix a) => Array a b = MkArray (a,a) (a -> b)
```

The class `Ix` is that of array indexes. It is a subclass of the class `Ord` of types with an order relation, and a number of additional operations are required for them:

```
range   :: (a,a) -> [a]        --lists all indexes between bounds
index   :: (a,a) -> Int        --converts an index into a position
inRange :: (a,a) -> a -> Bool --checks an index to be in range
```

However, arrays are considered as abstract data types and the constructor `MkArray` is not exported. The programmer may create the array by using the predefined function `array::(Ix a) => (a,a) -> [Assoc a b] -> Array a b` which needs an *association list*, or by using a function

```
mkArray:: (Ix a) => (a,a) -> (a -> b) -> Array a b
```

defined as in [HF92], which takes as arguments the array bounds and a function from indexes to values that we will call the *creation function.* The data type `Assoc` defines tuples of the form `i := v`, where `i` is an index and `v` is the value we wish to associate, in the new array, to index `i`. In any of these ways, arrays are dynamically created. Once created, they can be consulted or updated by using the predefined operations:

```
(!)   :: (Ix a) => Array a b -> a -> b
(//)  :: (Ix a) => Array a b -> [Assoc a b] -> Array a b
```

A reasonable implementation of the array feature will, of course, map an array into consecutive memory locations so that operator `(!)` could execute in constant time. Also, compile time analysis techniques are expected to detect those situations in which arrays can be safely updated in place without generating fresh copies of them.

If a1, a2 are instances of the class Ix, so it is the tuple (a1,a2) formed by them. The range function for tuples is defined in such a way that the array elements are enumerated by rows. For example, range ((0,4),(1,5)) produces the list [(0,4),(0,5),(1,4),(1,5)].

Definition 1 If f p1 p2...pn = rhs defines a recursive function f, and t1 is the type of p1,..., tn is the type of pn, and (Ix t1,...,tn) holds, we call the *totally tabulated* version of f to the following set of definitions:

```
f p1...pn = table!(p1,...,pn)
          where table          = mkArray b f'
                f' (j1,...,jn) = rhs'
```

where b has the form ((l1,...,ln),(u1,...,un)) and, for all i, li, ui are respectively the lower and upper bounds of argument pi. They may depend, in general, on the parameters p1...pn, so the definition of table in a local clause where the arguments p1...pn are visible. The expression rhs' is obtained from rhs by substituting ji for pi, for all i, and the subexpression table!(e1,...,en) for any recursive call in rhs of the form f e1...en. We will denote this fact by rhs' $\stackrel{\text{def}}{=}$ rhs[j1,...,jn,table!(e1,...,en)/p1,...,pn,f e1...en]. □

Proposition 2 Given a recursive definition for a function f and the definition of its totally tabulated version, both compute the same function provided that, $\forall$i.li $\leq$ pi $\leq$ ui is an invariant of f. □

The term *invariant* is used here in the following sense: it is a property that can be shown holds for any internal call to f assuming it holds for f.

To prove this proposition we formalize equational reasoning by using the fold/-unfold system of [BD77] whose rules can be summarized as follows:

Definition rule A new equation lhs = rhs is introduced so that lhs is not an instance of a left hand side of any other equation.

Instantiaton rule An equation lhs = rhs is derived from a previous one, where lhs and rhs are respectively instances of its left and right hand sides.

Unfolding rule In the right hand side of an equation, a subexpression e which is an instance of the left hand side of another equation is replaced by the corresponding instantiated right hand side e'.

Folding rule In the right hand side of an equation, a subexpression e which is an instance of the right hand side of another equation is replaced by the corresponding instantiated left hand side e'.

Abstraction rule All the ocurrences of subexpressions e1,...,en in a subexpression e of the right hand side of an equation are abstracted in a where clause, resulting the new expression e[z1,...,zn/e1,...,en] where (z1,...,zn) = (e1,...,en), where z1,...,zn are fresh variables.

Law rule Like the folding or unfolding rules, but replacing the equation referenced there by an algebraic law e = e'.

We need to use the following obvious algebraic law for arrays:

```
mkArray b f!i = f i    provided    inRange b i    holds        (1)
```

Then, the following derivation proves the proposition:

```
    f p1...pn
≡ fTable!(p1,...,pn)   {definition rule}
≡ mkArray b f'!(p1,...,pn)   {unfolding rule}
≡ f' (p1,...,pn)   {law rule, using law 1}
≡ rhs'[p1,...,pn/j1,...,jn]   {unfolding rule}
≡ rhs   {folding rule on f assuming the invariant holds}
```

Example 3 The following recursive function defines the combinatory number $\binom{m}{n}$:

```
comb m 0          = 1
comb m n | m == n = 1
         | m > n  = comb (m-1) (n-1) + comb (m-1) n
```

Applying proposition 2, we get the following totally tabulated version of comb:

```
comb m n = combTable!(m,n)
         where combTable = mkArray ((0,0),(m,n)) comb'
               comb' (i,0)          = 1
               comb' (i,j) | i == j = 1
                           | i > j  = combTable!(i-1,j-1) +
                                      combTable!(i-1,j)
```

□

3 Partial tabulation

It can be easily seen that total tabulation is in most cases an uselessly expensive technique in memory space. By studying the dependency graph of the recursive function, usually a set of tabulated results smaller than the whole set can be identified as a set having enough results to solve the current subproblem. This set may vary in size and contents from one invocation to another.

The *dependency graph* [Bir80] for a concrete function application is an acyclic graph obtained from the tree of recursive calls by the identification of nodes with identical arguments. A *cut* is a set of nodes of the dependency graph such that, were these nodes —and their incident edges— deleted from the graph, the result would be two disjoint graphs. A sequence of cuts $c_n, c_{n-1}, \ldots, c_0$ is *progressive* if, for all i, $1 \leq i \leq n$, every node in cut c_i can be computed from nodes in c_i or in c_{i-1}. To be more precise, we reproduce the definition given by Pettorossi [Pet84]:

Definition 4 A sequence of cuts $c_n, c_{n-1}, \ldots, c_0$ is *progressive* if,

$$\forall i, 1 \leq i \leq n.(c_i \neq c_i \cap c_{i-1} \neq c_{i-1})$$
$$\wedge(\forall m \in c_i - c_{i-1}.\exists n \in c_{i-1}.m \rightarrow n) \wedge (\forall n \in c_{i-1} - c_i.\exists m \in c_i.m \rightarrow n)$$

where $m \rightarrow n$ stands for "there exists a directed edge from node m to node n". □

The partial tabulation technique

For the sake of clarity, let us assume that the recursive function is defined by the single equation:

$$\texttt{f p1...pn} = \texttt{rhs} \tag{2}$$

where `rhs` contains references to `f`. Assume also that `res` is the type of `rhs`.

The first step of the technique proposed here is, like in the tupling technique, to define a progressive sequence of cuts in the dependency graph. Now, however, we remove the limitation that all the cuts must have a fixed number of nodes.

The second step consists —by applying the definition rule (see section 2)— of the introduction of an *eureka* equation with the following aspect:

```
cutTable q1...qr = mkArray b f'
                   where f' i = f e1...en
```

where `cutTable q1...qr` is the table to store the results of a cut, `b::(a,a)` are its bounds which, in general, depend on `q1,...,qr`, `f'::a -> res` is its creation function, and `e1,...,en` are expressions depending on `q1,...,qr` and `i`. Since the cut table will store values computed by `f`, one can expects that `f'` will be initially defined in terms of `f` (even, in some examples, `f'` coincides with `f`).

Then, for any cut and using law 1, we have the property:

$$\texttt{cutTable q1...qr ! i} \equiv \texttt{f e1...en provided inRange b i} \tag{3}$$

In particular, for the top level cut and for some actual parameters `s1,...,sr` and index `s` such that `inRange btop s`, where `btop` depends on `s1,...,sr`, we have:

$$\texttt{cutTable s1...sr ! s} \equiv \texttt{f p1...pn} \tag{4}$$

The next steps consist of eliminating all references to `f` in the definition of `f'`. For this purpose, we may instantiate the definition of `f` inside `f'` trying to define `f'` for the non recursive cases; we may introduce an auxiliary cut table representing the following cut in the sequence; and we may use property 3 from right to left to replace references to `f` by references to a cut table.

Finally, if we have succeeded in the previous steps, we eliminate equation 2 and replace it by a right to left version of property 4:

```
f p1 ... pn = cutTable s1 ... sr ! s
```


Example 5 In example 3 the result for `comb m n` depends, for all `m`> 0, on two values belonging to the previous row of the Pascal triangle. So, a possible idea for partial tabulation is to maintain two tables corresponding to two consecutive rows of the Pascal triangle, one of them being filled by using the values stored in the other. In fact, only a portion of each row is needed. We define, then, a cut as the needed values (say, from a lower bound `j1` to an upper bound `j2`) of row `i`.

Then, we introduce the following *eureka* equation:

```
combTable i j1 j2 = mkArray (j1,j2) comb'
                    where comb' k = comb i k
```

and obtain the following two laws:

$$\texttt{combTable i j1 j2 ! k} \equiv \texttt{comb i k} \quad \forall \texttt{k} \in \{\texttt{j1...j2}\}$$
$$\texttt{combTable m n n ! n} \equiv \texttt{comb m n}$$

After several instantiation, unfolding, folding and abstraction steps we obtain the following tabulated version of `comb`:

```
comb m n = combTable m n n ! n
combTable 0 0  0  = mkArray (0,0) (const 1)
combTable i j1 j2 = mkArray (j1,j2) comb'
   where comb' k
           | k==0 || k==i = 1
           | otherwise    = upRow ! (k-1) + upRow ! k
         upRow = combTable (i-1) (max 0 (j1-1)) (min j2 (i-1))
```

The transformation is straightforward, with the exception of the limits of the cuts, which require additional reasoning. □

In the next section we give more significative examples. In the first one, the transformation steps are carried out in full detail.

4 Cases study

0/1-knapsack problem

This is the well-known problem of finding the maximum profit we can get by introducing a subset of a set of n objects in a knapsack with fixed capacity c. For simplicity, we are assuming the existence of two global arrays `p, b :: Array Int Int`, of length `n`, respectively containing the weights and benefits of the n objects. Its recursive version follows:

```
knapsack   0   c = 0
knapsack (n+1) c
      | c < p!(n+1) = knapsack n c
      | otherwise   = max (knapsack n c)
                          (knapsack n (c-p!(n+1)) + b!(n+1))
```

In the analysis phase, we observe that its dependency graph consists of a n×c matrix, whose items represent the values of `knapsack i j`, for $0 \leq i \leq$ `n` and $0 \leq$ `j` $\leq$ `c`. Therefore, we provide an array with these bounds for the total tabulation version. By applying proposition 2, we obtain:

```
knapsack n c = knapsackTable!(n,c) where
   knapsackTable = mkArray ((0,0),(n,c)) knapsack'
   knapsack'(0,j) = 0
   knapsack'(i+1,j)
      | j<p!(i+1) = knapsackTable!(i,j)
      | otherwise = max (knapsackTable!(i,j))
                        (knapsackTable!(i,j-p!(i+1)) + b!(i+1))
```

To do a partial tabulation we choose cuts corresponding to a row in the total table (i.e., we fix the number `i` of objects and let the capacity `c` to vary). The equation representing this idea is:

```
knapsackCut i c = mkArray (0,c) f' where f' j = knapsack i j
```

Instantiating laws 3 and 4, we have the laws:

```
knapsackCut i c!j ≡ knapsack i j    ∀j ∈ {0...c}      (5)
knapsackCut n c!c ≡ knapsack n c                      (6)
```

To transform `f'`, we instantiate equation 5 according to the definition of knapsack:

```
knapsackCut 0 c = mkArray (0,c) f' where
   f' j = knapsack 0 j ≡ 0     {by unfolding}
knapsackCut (i+1) c = mkArray (0,c) f' where
   f' j = knapsack (i+1) j
```

Now, the transformation of the `f'` local to the scope of `knapsackCut (i+1) c` results in:

```
f' j | j < ws!(i+1)   = knapsack i j
                      ≡knapsackCut i c!j   {by law 5}
                      = max (knapsack i j)
                            (knapsack i (j-ws!(i+1)) + ps!(i+1))
                      ≡max (knapsackCut i c!j)   {by law 5}
                            (knapsackCut i c!(j-ws!(i+1)) + ps!(i+1))
     | otherwise
```


Finally, we apply the abstraction rule and obtain:

```
knapsack n c = knapsackCut n c ! c
knapsackCut   0   c = mkArray (0,c) (const 0)
knapsackCut (i+1) c = mkArray (0,c) f' where
   f' j | j < p ! i = previousRow!j
        | otherwise = max (previousRow!j)
                          (previousRow!(j-p!(i+1)) + b!(i+1))
                    where previousRow = knapsackCut i c
```

Minimal decomposition into triangles

This problem is solved in [AHU83] by the dynamic programming method. It consists of the decomposition of a convex polygon with `n` vertexes into triangles so that the sum of the lengths of the chosen diagonals be minimal. The call `minTri 1 n` solves the problem. In general, `minTri i s` finds the minimal cost for a polygon starting at vertex `i` of the original polygon and consisting of `s` consecutive clockwise vertexes, including `i`.

```
minTri i s | s<=3 = 0
           | s >3 = minimum [dist!(i,i+k) + dist!(i+k,i+s-1) +
                             minTri i (k+1) + (minTri (i+k)(s-k))
                             | k <- [1..s-2]]
```

where we are assuming that `dist` is a global array with the lengths of the diagonals, and `dist!(i,j)` gives 0 for two consecutive vertexes. The total tabulated version follows:

```
minTri 1 n = table!(1,n) where
 table = mkArray ((1,2),(n-1,n)) minTri'
 minTri'(i,s) | s<=3 = 0
 minTri'(i,s) | s> 3 = minimum [dist!(i,i+k) + dist!(i+k,i+s-1)
                                + table!(i,k+1) + table!(i+k,s-k)
                                | k <- [1..s-2]]
```

For this problem, we have not found a progressive sequence of cuts, so we do not present a partial tabulated version.

Fibonacci sequence

We include this well-known example to show that tupling can be seen as a particular case of partial tabulation.

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

Here, we are only interested in the partially tabulated version. The obvious cut is a pair of consecutive Fibonacci numbers.

```
fibCut n = mkArray (n,n+1) fib' where
  fib' i = fib i
```

so, the following law holds:

$$\texttt{fibCut n ! i} \equiv \texttt{fib i} \tag{7}$$

for `i=n` and `i=n+1`. Now, we instantiate `fibCut n` for two patterns of `n`, and `fib'` for the two values of `i`, namely `n+1` and `n+2`, in the second equation of `fibCut`, resulting in:

```
fibCut   0    = mkArray (0,1) fib' where fib' i = 1
fibCut (n+1)  = mkArray (n+1,n+2) fib' where
                fib' i| i==n+1 =  fib (n+1)
                      ≡        fibCut n ! (n+1)        {by law 7}
                      | i==n+2 = fib (n+2)
                      ≡        fib n + fib(n+1)        {by unfolding}
                      ≡        fibCut n!n + fibCut n!(n+1) {by law 7}
```

We do the case analysis because the local equation `fib' i = ...` has two possible actual parameters, `n` and `n+1`. Finally, by the abstraction rule, we have the final version:

```
fib n = fibCut n ! n
fibCut 0 = mkArray (0,1) (const 1)
fibCut (n+1) = mkArray (n+1,n+2) fib' where
   fib' i | i==n+1 = prevCut ! (n+1)
          | i==n+2 = prevCut ! n + prevCut ! (n+1)
            where prevCut = fibCut n
```

In this way, `fibCut (n+1)` is defined in terms of `fibCut n`. This program can be straightforwardly transformed into the known tupled version:

```
fib n           = fst (tupledFib n)
tupledFib 0     = (1,1)
tupledFib (n+1) = (v,u+v) where (u,v) = tupledFib n
```

5 Application to other functional languages

The techniques just described are not restricted to languages with arrays, such as Haskell. They can be also used in any language providing lists. The resulting algorithms will probably be less efficient, but their polynomial times will still be better than the usually exponential time of the original functions. We will be rather

informal in this section, simply giving a flavour of the tabulation by re[illegible]ng one of the previous examples. For the sake of homogeneity, we will still use Ha[illegible]ell syntax.

Some distinction must be made between algorithms where the values of a cut only depend on values of the next cut, and algorithms which may also depend on values of the same cut. An example of the first case, is the 0/1 knapsack problem. One of the second case, is a version of the combinatory number problem in which cuts are sequences $\binom{n}{n}, \binom{n+1}{n}, \ldots, \binom{m}{n}$ parallel to the left side of the Pascal triangle.

In the knapsack problem, lists comprehensions can be used as cut tables. Assuming that weights and profits are implemented as global lists `ws` and `ps` and, reversing the order of the indexes, the definition of a cut table for the 0/1 knapsack problem is:

```
knapsackT i c = [ knapsack (i,j) | j <- [c,c-1..0] ]
```

Its associated law is similar to that of arrays:

$$\texttt{knapsack i w} = \texttt{knapsackT i c !! (c - w)} \text{ provided } \texttt{w} \in \{0..c\} \tag{8}$$

The transformation is also similar to the one given in section 4, resulting in the following algorithm:

```
knapsack n c      = knapsackT n c !! 0
knapsackT   0   c = [0 | j <- [c, c-1 .. 0]]
knapsackT (i+1) c =
            [if j < ws!!(i+1) then u!!(c-j)
            else max (u!!(c-j)) (u!!(c-j+ws!!(i+1)) + ps!!(i+1))
            | j <- [c, c-1 .. 0]]
            where u = knapsackT i c
```

List comprehensions could be automatically removed [BW88] if we were working with languages, such as Standard ML, not having them. The resulting program would be expressed in terms of the higher-order functions `map` and `filter`, that could also be removed by well known program transformation tactics [Chi90].

If further refinements are made, more efficiency can be gained. Let us notice that the removal of list comprehensions and of higher-order functions produces an algorithm where values in a cut are obtained by doing some search in the list representing the next cut. A careful transformation allows us to eliminate, or at least to reduce, this search. This goal would force us to use some more laws relating list comprehensions to the recursive construction of lists.

By doing these improvements, the resulting 0/1 knapsack algorithm is the following:

```
knapsack' n c      = knapsackT' n c !! 0
knapsackT'   0   c = [0 | j <- [c, c-1 .. 0]]
knapsackT' (i+1) c = k (i+1) c (knapsackT' i c)
```

```
k   _   0 (x:_)     = [x]
k (i+1) j (x:xs)    =
   if j<ws!!(i+1) then x : k (i+1) (j-1) xs
   else (max x (xs!!(ws!!(i+1)-1) + ps!!(i+1))) : k (i+1) (j-1) xs
```

This version could still be further improved by eliminating the accesses to lists by operator `!!`, but we are not reproducing here the resulting version because of its obscurity.

Acknowledgements: We wish to express our gratitude to Javier Deniz by his help with the Glasgow Haskell Compiler while preparing this paper.

References

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[BD77] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bir80] R. S. Bird. Tabulation Techniques for Recursive Programs. *Computing Surveys*, 12(4):403–417, December 1980.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[Chi90] Wei-Ngan Chin. Automatic methods for program transformation. Ph. D. Thesis, Imperial College, University of London, 1990.

[Coh83] N. H. Cohen. Eliminating Redundant Recursive Calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[HF92] P. Hudak and J. H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN NOTICES*, 27(5), May 1992.

[HPJW92] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, 27(5), 1992.

[Hug85] J. Hughes. Lazy memo-functions. In *Conf. on Functional Programming and Computer Architecture. Lecture Notes in Computer Science 201, Springer Verlag*, 1985.

[Kho90] H. Khoshnevisan. Efficient Memo-Table Management Strategies. *Acta Informatica*, 28:43–81, 1990.

[Pet84] A. Pettorossi. A Powerful Strategy for Deriving Efficient Programs by Transformation. In *ACM Symposium on Lisp and Functional Programming, Austin, Texas, 6-8 Aug.*, pages 273–281, 1984.

Síntesis de programas lógicos: Marco Constructivo

Francisco José Galán Morillo y Miguel Toro Bonilla
Departamento de Lenguajes y Sistemas Infomáticos
Facultad de Informática y Estadística Universidad de Sevilla
Av. Reina Mercedes s/n c.p.: 41012-SEVILLA
Tef: +34 5 4552773 Fax: +34 5 4557139
E-mail: fjgalan@obelix.cica.es

Abstract

El problema se centra en la derivación de programas lógicos desde especificaciones en lógica de primer orden utilizando el concepto de extracción de programas desde pruebas [ManWal80] y [ConBat85]. Sintetizar un programa desde su especificación consiste en establecer que la especificación es decidible. Usaremos la Lógica Intuicionista como filosofía de razonamiento y adoptaremos como sistema de prueba el Cálculo de Secuentes de Gentzen. Ampliaremos el sistema de pruebas con nuevas reglas que permitan modelar de modo más adecuado aspectos relevantes en la programación: esquema de programas, transformaciones, recursion, ...etc.

Palabras claves: programas lógicos, especificación, inducción matemática, síntesis de programas, cálculo de secuentes, validez, completitud, prueba de teorema, decibilidad.

1 Modelo de Especificación

El modelo de especificación tipado adoptado es el siguiente:

$$\Gamma \vdash E(X{:}\mathrm{t}, Y{:}\mathrm{t}) ::_{p(X:\mathrm{t},Y:\mathrm{t})}$$

- p es el programa lógico que deseamos construir (asociado a E).
- X e Y son parámetros del predicado p y E.
- t es el tipo de los parámetros.
- E es la especificación del problema.

- Γ es el conocimiento de fondo (axiomas propios de las teorías que enmarcan el problema).

2 Cálculo de Secuentes: Construyendo programas

Un secuente es una expresión $\Gamma \vdash \Delta$ donde Γ and Δ son conjuntos finitos de fórmulas.

Semántica:
$\{\alpha_1, ..., \alpha_n\} \vdash \{\beta_1, ..., \beta_m\}$ sii $(\alpha_1 \wedge ... \wedge \alpha_n) \rightarrow (\beta_1 \vee ... \vee \beta_m)$ es válida.
Formalmente, el proceso de deducción es una tarea hacia adelante, pero con secuentes es "más natural" usar razonamiento hacia detrás; es decir, partir del objetivo y descomponerlos en subobjetivos hasta finalizar o no en s-axiomas.

El constructivismo enlaza perfectamente con la tarea de construcción de programas, por lo tanto, los secuentes intuicionistas tendrán la siguiente forma:

$$\{\alpha_1, ..., \alpha_n\} \vdash \{\beta\}$$

Cada regla del sistema de prueba tiene asociada una regla de construcción de programas; por lo tanto, mientras se prueba la especificación se construye el programa. [Wig91]:

- s-axioma:

$$\frac{}{\Gamma, \alpha \vdash \alpha :: p}$$

- Antecedente

$$\frac{\Gamma \vdash \gamma :: p}{\Gamma, \alpha \vdash \gamma :: p} \qquad \wedge \frac{\Gamma, \alpha, \beta \vdash \gamma :: p}{\Gamma, \alpha \wedge \beta \vdash \gamma :: p}$$

$$\vee \frac{\Gamma, \alpha \vdash \gamma :: p_1 \quad \Gamma, \beta \vdash \gamma :: p_2}{\Gamma, \alpha \vee \beta \vdash \gamma :: p_1 \vee p_2} \qquad \rightarrow \frac{\Gamma \vdash \alpha :: p \quad \Gamma, \beta \vdash \gamma :: q}{\Gamma, \alpha \rightarrow \beta \vdash \gamma :: p \wedge q}$$

$$\neg \frac{\Gamma \vdash \alpha :: p}{\Gamma, \neg \alpha \vdash :: p} \qquad \forall \frac{\Gamma, \forall x \alpha(x), \alpha(t) \vdash \gamma :: p}{\Gamma, \forall x \alpha(x) \vdash \gamma :: p}$$

$$\exists \frac{\Gamma, \alpha(y) \vdash \gamma :: p}{\Gamma, \exists x \alpha(x) \vdash \gamma :: p}$$

- Consecuente

$$\frac{\Gamma \vdash :: false}{\Gamma \vdash \gamma :: false} \qquad \wedge \frac{\Gamma \vdash \alpha :: p \quad \Gamma \vdash \beta :: q}{\Gamma \vdash \alpha \wedge \beta :: p \wedge q}$$

$$* \vee \frac{\Gamma \vdash \alpha :: p}{\Gamma \vdash \alpha \vee \beta :: p} \qquad ** \vee \frac{\Gamma \vdash \beta :: p}{\Gamma \vdash \alpha \vee \beta :: p}$$

$$\neg \frac{\Gamma, \alpha \vdash :: false}{\Gamma \vdash \neg \alpha :: false} \qquad \rightarrow \frac{\Gamma, \alpha \vdash \beta :: p}{\Gamma \vdash \alpha \rightarrow \beta :: p}$$

$$\forall \frac{\Gamma \vdash \alpha(y) :: p}{\Gamma \vdash \forall x \alpha(x) :: p} \qquad \exists \frac{\Gamma \vdash \alpha(t) :: p}{\Gamma \vdash \exists x \alpha(x) :: p}$$

$$\vee \frac{\Gamma \vdash \alpha :: p \quad \Gamma \vdash \beta :: q}{\Gamma \vdash \alpha \vee \beta :: p \vee q}$$

Las reglas $\forall$ y $\exists$ se cumplen sólo bajo estas condiciones:

- y es una variable y t es un término ambos libres para x en $\alpha(x)$.
- $\alpha(y)$ y $\alpha(t)$ resultan de $\alpha(x)$ reemplazando toda ocurrencia libre de x por y y t respectivamente.
- En $\forall$ en el consecuente, y no ocurre libre en $\Gamma \vdash \forall x \alpha(x)$, y para y en $\exists$ en el antecedente, y no ocurre libre en $\Gamma \vdash \gamma$ ni en $\exists x \alpha(x)$.

El uso de * y ** sintetiza programas lógicos sólo parcialmente correctos.

Los s-axiomas serán las "semillas" iniciales de nuestro proceso constructivo; llamaremos a sus programas asociados **constructores de programas.**

Es importante destacar que los programas lógicos sintetizados serán Programas Lógicos Definidos [Llo87]. En nuestro sistema constructivo cada fórmula tiene un programa lógico asociado pero **no** la **negación** de éste. Esto no es una limitación computacional sino sólo expresiva.

3 Cálculo de Secuentes Extendido

Podemos encontrarnos con situaciones del estilo de:

$$\Gamma, Pl \vdash Ps$$

En este punto, la prueba está **bloqueada**, no es posible considerar este secuente como un s-axioma. Por lo tanto es necesario extender el sistema de prueba:

- 1) Incorporar la **unificación** en los secuentes.
- 2) Incorporar **razonamiento equacional** en los secuentes.
- 3) **Transformaciones** basadas en E y Γ
- 4) Resolución de **bloqueos** en situaciones **completas.**