Lazy Linear Combinatorial Unification *

Marco Bellia and M.Eugenia Occhiuto
Dipartimento di Informatica,
Università di Pisa
e-mail: occhiuto@di.unipi.it

September 3, 1998

Abstract

In this paper we define a two-rule reduction system for the lazy computation of unification on first-order terms. The computation consists of a sequence of reductions which is almost linear with the size of the input. Every reduction transforms the problem into an equivalent one which is a better approximation of the solution. The approach is based on a combinatorial system which provides abstractions for terms and term unification, and does not resort to any specific reduction control strategy. The abstraction on both data and control makes our system suitable in any applicative context where unification is required.

1 Introduction

Since 1965, when J. A. Robinson [Robinson 65, Robinson 92] defined it as the substitution rule for resolution, unification has been widely used in several areas of computer science, e.g. programming languages, automated reasoning and artificial intelligence [Siekmann 90]. Several algorithms have been proposed to improve the complexity and efficiency of Robinson's first algorithm which was exponential in time and in space, see [Knight 89] for a survey on the subject. At the end of the Seventies, two algorithms for ordinary first order terms [Paterson 78, Martelli 82] and one for rational terms [Huet 76] were proposed, which unify in linear or almost linear time. The definition of such algorithms above all proved that unification can be computed at a feasible computational cost. In fact, such algorithms are very seldom used in real applications [Corbin 83]. In this paper we will briefly analyze the reasons for such a

*This research was partially supported by MURST: Programmi di ricerca di rilevante interesse nazionale 1997, N°9701248444 - Sistemi formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software.

gap between theory and practice. Linear complexity would appear to be due to the choice of the data structures used to represent the terms and the solution (i.e. unification substitution) respectively, and of the control structures used for the term visiting strategy. The resulting algorithms lack simplicity, and moreover when the data structures do not match those used in the applicative context in which unification is to be implemented, they require an extra process of mapping from one representation to the other, or alternatively auxiliary data structures: In both cases, the usability of the algorithms is seriously undermined [Mukai 83, Jaffar 84, Ruzicka 89]. In practice, a more expensive algorithm, but simple and well embedded in the whole implemented system, is preferable especially when the applicative context is such that the worst case very seldom, or never, occurs. Evidence of this is the WAM, which uses a revised version of Robinson's algorithm, with structure sharing [Aitkaci 90] [Ciprietti 95], whose behavior is similar to the quadratic time, linear space, algorithm in [Corbin 83].

In this paper we present a new linear unification algorithm which is based on a combinatorial system, $\mathcal{EC}$-expressions, which provides abstractions for unification and building terms, computation and the presentation of unification results. Studying unification, it is easy to realize that the way to deal with repeated variables is crucial in terms of the algorithm's complexity [Bellia 98, Jaffar 84]. A multi-occurring variable, in fact, states an (equivalence) relation among subterms (unified with that variable). In our approach this relation is explicitly maintained in the combinatorial structure used for terms. In this way, any binding that the computation of unification states on an occurrence of a variable is propagated to all the other occurrences by the effect of such a relation. In particular neither variable substitution nor structure sharing are required to support the computation. The abstraction for terms thus obtained is independent of any specific supporting structures, in other words it provides an abstract representation for terms whose implementation can be chosen according to the context in which unification is to be used. Our approach also provides abstraction for the computation control by guaranteeing the linearity of performed reductions irrespective of the subterms visiting strategy.

Another benefit of the combinatorial approach, we have chosen, is that it keeps in the same domain both the terms and results of term unifications, since the former (terms) are used as normal forms of the latter. This leads to a formulation of unification slightly different from Robinson's formulation [Robinson 65], in which unification computes a term morphism, or Paterson's [Paterson 78], and likewise Huet's [Huet 76], in which unification computes a minimal valid equivalence relation on graph vertices, or finally Martelli's [Martelli 82], in which unification computes a solved form of an equation system. In our approach, unification of a finite set of terms computes the least upper bound of the terms in the set [Reynolds 70, Bellia 95]. This makes the values computed by unification fully compatible with the objects on which it applies, in fact both are terms. At the same time, the computed values have a well founded algebraic formulation, described in [Bellia 95]. Another property of our approach is that it provides a lazy computation of unification, computing a sequence of increasingly better approximations of the result. This allows the use of unification as a lazy operator inside a more general lazy calculus. The algorithm may deal with rational infinite terms as well as finite terms. The domain of finite terms is the domain on which unification was originally expressed by Herbrand [Herbrand 30]. On the other hand rational terms, which are possibly infinite terms but with a finite number of different subterms, were introduced in [Huet 76] to apply unification to type inference. Rational terms also eliminate occur-check overheads, as pointed out in [Colmerauer 82], thus making the unification algorithm as fast as a term matching algorithm, when unification reduces to term matching, which is often the case. In our approach, dealing with rational infinite terms is a consequence of the abstraction on control and is accomplished by superimposing different control strategies. A control strategy constrains rule applicability allowing, for instance, reductions to be stopped when rational infinite terms are involved.

The combinatorial system presented in this paper extends the system of $\mathcal{C}$-expressions discussed in one of our earlier works on a combinatorial calculus for inferential programming systems including resolution and narrowing, [Bellia 93]. The paper is organized as follows. Section 2 contains the preliminary definitions which will be used throughout the paper. Section 3 summarizes the definitions and properties of $\mathcal{C}$-expressions fully developed in [Bellia 93]. Section 4 introduces $\mathcal{EC}$-expressions as an extension of $\mathcal{C}$-expressions. Section 5 defines of a new reduction rule system and examines its main properties. Section 6 is on system complexity. Section 7 presents a plain implementation of the system. Section 8 discusses the role of the control strategies and data representation in getting different implementations of the rule system. Section 9 summarizes the results and discusses future work.

2 Preliminaries

In this section we give a few essential definitions which are used in the rest of the paper. A complete presentation of the problem of unification, which is beyond the scope of the paper, can be found in [Jouannaud 91, Lassez 88, Reynolds 70, Eder 85].

2.1 First Order Terms

Let $(\Sigma, \mathcal{V})$ be a free Σ-algebra, where $\Sigma = \cup \Sigma_i$ is a set of constructors indexed by the arity i, i.e. $F \in \Sigma_{arity(F)}$, and $\mathcal{V}$ is a set of variables. The set of (first order) terms, $\Sigma(\mathcal{V})$, includes $\mathcal{V}$ and is the closure of the application of constructors to terms. The application of the constructor $F \in \Sigma_n$ to terms $h_1, ..., h_n$ is denoted by $(F\ h_1 ... h_n)$, where conventionally constructors begin with an upper case letter, and variable symbols with a lower case letter. A *substitution* ρ is a finite mapping from variables into terms. The domain of substitutions $\Gamma = \mathcal{V} \rightarrow \Sigma(\mathcal{V})$ allows one to introduce the *instance* operator. The instance of h under $\rho = \{x_1 \leftarrow h_1, x_2 \leftarrow h_2, ..., x_n \leftarrow h_n\}$, denoted as $h\rho$ is obtained from h by simultaneously replacing all the occurrences of each x_i in h by the h_i. Substitutions allow one to define an instance preorder, or subsumption, $\leq$ [Reynolds 70, Eder 85] on terms in the following way: for any $\rho \in \Gamma$ and $h \in \Sigma(\mathcal{V})$, we have $h\rho \leq h$. The preorder becomes a partial order on $\Sigma(\mathcal{V})/\approx$, where $\approx$ is the variant relation. Finally $\Sigma(\mathcal{V})/\approx$, completed with a top, $\top = lub(\Sigma(\mathcal{V}))$[1], and a bottom element, $\perp = glb(\Sigma(\mathcal{V}))$, is a complete lattice [Reynolds 70].

Terms are often considered as finite ordered trees, labeled with constructors from Σ and variables from $\mathcal{V}$. In this context several operators can be defined on terms [Dershowitz 90], we use the following:

- $Vars(h)$ computes the set of variables occurring in h. For instance if $h = (F\ x\ (G\ A\ y))$, then $Vars(h) = \{x, y\}$.

- by $h|_p$, we denote the subterm of h rooted at *path* p. A path (represented in Dewey decimal notation) is a sequence of positive integers which describes the path from the outermost root symbol to the root of a subterm. We denote λ as the empty path to the root. For example, if $h = (F\ x\ (G\ A))$, then $h|_1 = x$, $h|_{2.1} = A$, $h|_2 = (G\ A)$. Moreover, when P is a set of paths and i is an index then $P.i$ is the set $\{p.i\ s.t.\ p \in P\}$ and $h|_P = \{h|_p\ s.t.\ p \in P\}$. For example $h|_{\{\lambda,2\}.1} = \{x, A\}$.

- $Paths(h) = \{p \mid \perp \neq h_p\}$ is the set of all paths which are defined in h, for instance if $h = (F\ x\ (G\ A))$, then $Paths(h) = \{\lambda, 1, 2, 2.1\}$.

- $occ(h, h') = \{p \mid h' = h|_p\}$ that is the set of paths in h at which h' occurs. For instance if $h = (F\ x\ (G\ (F\ x\ x)))$, then $occ(h, x) = \{1, 2.1.1, 2.1.2\}$.

- by $h[p \leftarrow h']$, we denote the term obtained by replacing in the term h the subterm occurring at path p with the term h'. For instance if $h = (F\ x\ (G\ A))$, then $h[2.1 \leftarrow (F\ y\ B)]$ results in the term $(F\ x\ (G\ (F\ y\ B)))$. Moreover, when P is a set of paths $h[P \leftarrow h'] = (h[p \leftarrow h'])[(P - \{p\}) \leftarrow h']$. Eventually, we also write $h[t \leftarrow h']$ for $h[occ(h,t) \leftarrow h']$.

Hereafter, $\Re$ will denote the set of all finite paths and $\Re_\omega$ the set of finite subsets of $\Re$. We will reserve the letter h, possibly indexed, for terms, and the letters P, Q and R, possibly indexed for sets of paths.

2.2 Most General Instance and Unification

We can extend to tuples the lattice structure $\Sigma(\mathcal{V})/\approx$ defined for terms. Tuples are ordered structures of the form $(h_1 ... h_n)$, where $h_i s$ are terms. Formally,

[1] lub is the least upper bound and glb is the greatest lower bound.

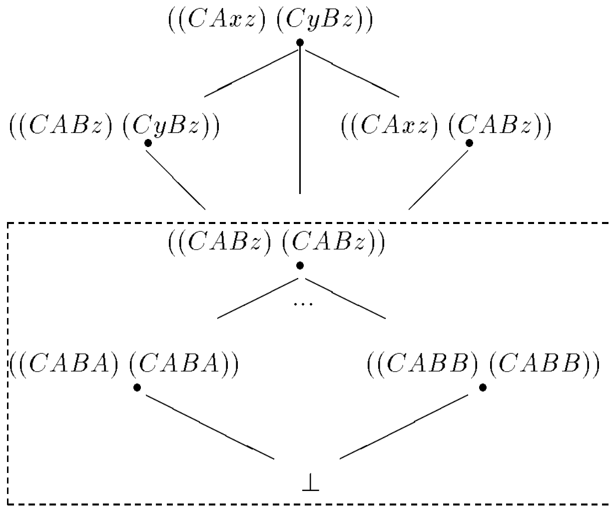


Figure 1: Computation of $mgi(\{1,2\},((C\ A\ x\ z),(C\ y\ B\ z)))$ in $\Sigma(\mathcal{V})^*/\approx$

$\Sigma(\mathcal{V})^*$ is the disjoint union, i.e. $\uplus\Sigma(\mathcal{V})^n$, of $\Sigma(\mathcal{V})^n$, where $\Sigma(\mathcal{V})^n$, for each natural n, is the set of tuples of length n. The exponent n is called the tuple order. The above defined operations on terms are easily extended to tuples as well. In particular, if n is the order of H, $H|_i$ for $i \in [1,n]$ is the i-the component of H. For example, let $H = ((A\ y\ (C\ z))\ (B\ y)\ (A\ 0\ z)\ (C\ z))$, then $H|_1 = ((A\ y\ (C\ z))$ and $H|_{2.1} = y$ and $H|_{1.2.1} = z$.

An operator mgi (most general instance) [Bellia 90] can be defined on tuples in the following way: given a tuple H of order n, for each $P \subseteq Paths(H)$, $mgi(P,H) = lub\{T \mid T < H \wedge \forall i,j \in P, T|_i = T|_j\}$. An mgi problem is any problem $mgi(P,H)$, for arbitrary sets of paths $P \subseteq Paths(H)$. Given any H of order n, we say that $mgi(\{i \mid 1 \leq i \leq n\}, H)$ is a unification problem, and solving it entails computing the $H' = lub\{T \mid T < H \wedge \forall i,j \in [1,n]\,, T|_i = T|_j\}$. We say that mgi problems generalize unification problems since if $H' \neq \perp$, then $\theta = mgu(\{i \mid 1 \leq i \leq n\}, H)$ can be easily computed from H' in the following way: $\theta = \{x \leftarrow t \mid x \in Vars(H) \wedge x = H|_p \wedge t = H'|_p\}$, that is, each variable occurring in H at a path p is bounded to the term occurring in H' at the same path p. The example in Figure 1 (where a directed edge from t to t' means $t' \leq t$) shows a few points in the domain $\Sigma(\mathcal{V})^*/\approx$ which describes the computation of $H' = mgi(\{1,2\},H)$ where $H = ((C\ A\ x\ z)\,(C\ y\ B\ z))$. In this case $H' = ((C\ A\ B\ z)\,(C\ A\ B\ z))$, and according to the subsumption relation $H' \leq H$. The elements of the set $S = \{T \mid T < H \wedge \forall i,j \in P, T|_i = T|_j\}$ are the tuples enclosed in the dashed box and $H' = lub(S)$.

When tuples are used as a presentation for first order formulas, mgi can

$\pi : \Sigma(\mathcal{V})^n \rightarrow \Sigma(\mathcal{V})^n$	$(identity)$
$(\forall H \in \Sigma(\mathcal{V})^n), \;\; \boldsymbol{\pi}\ H = H$	
$\uparrow \_\,\_ : \Sigma_a \rightarrow \aleph \rightarrow \Sigma(\mathcal{V})^n \rightarrow \Sigma(\mathcal{V})^{n-a+1}$	$(construction)$
$(\forall C_a \in \Sigma_a, j \in [1, n-a+1], H \in \Sigma(\mathcal{V})^n),$	
$\uparrow \mathbf{(C_a)\,(j)}\ \ H = (H\vert_1 ... H\vert_{j-1}\ (C_a\ H\vert_j ... H\vert_{j+a-1})\ H\vert_{j+a} ... H\vert_n)$	
$Eq[\_,\_] : \aleph \times \aleph \rightarrow \Sigma(\mathcal{V})^n \rightarrow \Sigma(\mathcal{V})^n$	$(equality)$
$(\forall i \leq j \in [1, n], H \in \Sigma(\mathcal{V})^n)$	
$\mathbf{Eq[i,j]}\ \ H = mgi(\{i, j\}, H)$	

Table 1: $\mathcal{C}$-combinators

act as a substitution rule of an inference system [Bellia 93]. For instance, let R be the cut rule of the resolution process (for clauses presented on $\Sigma(\mathcal{V})^*$). Let H be the presentation on $\Sigma(\mathcal{V})^*$ of a pair of clauses (C, G), and $H|_i, H|_j$ be a pair of complementary literals in C and G, respectively. Then mgi computes the most general instance of (C, G) for which R applies. For example, consider the pair of clauses $((A\ y\ y) \leftarrow (B\ y), \leftarrow (A\ 0\ z)\ (C\ z))$, it can be represented by the following seven element tuple $H \in \Sigma(\mathcal{V})^*$, $H = ((A\ y\ y) \leftarrow (B\ y) \& \leftarrow (A\ 0\ z)\ (C\ z))$. Hence $mgi(\{1, 6\}, H) = ((A\ 0\ 0) \leftarrow (B\ 0) \& \leftarrow (A\ 0\ 0)\ (C\ 0))$. The cut rule R is applied by reducing $((A\ 0\ 0) \leftarrow (B\ 0)\ \&\ \leftarrow (A\ 0\ 0)\ (C\ 0))$ to $(\leftarrow (B\ 0)\ (C\ 0))$.

3 $\mathcal{C}$-expressions and Tuples

As already mentioned, the combinatorial representation of unification problems is central to our approach and is based on our earlier work on equational logic programming. In particular, $\mathcal{EC}$-expressions will be introduced as an extension of $\mathcal{C}$-expressions. Hence, in this section we recall the main definition and properties of $\mathcal{C}$-expressions referring to [Bellia 93] for a complete presentation.

3.1 $\mathcal{C}$-expressions definition

$\mathcal{C}$-expressions are any compositions of the operators in Table 1. Such operators are called $\mathcal{C}$-combinators and are applied to tuples thus producing new tuples. Combinator π is identity. Construction $\uparrow$, applied to an a-arity constructor C_a and to an index j, replaces the terms $H|_j, ..., H|_{j+a-1}$ with the term having C_a as the main constructor and such terms as arguments. Equality Eq is the mgi operator previously mentioned. All the combinators compute $\perp$ when applied to $\perp$. In addition $Eq[i, j]H$ results in $\perp$ when $H|_i, H|_j$ are not unifiable in-

var :	$\boldsymbol{\eta}\ (V^n) = \pi$ if $occ(V, V\|_k) = \{k\}\ \forall 1 \leq k \leq n$
funct :	$\boldsymbol{\eta}\ (V^{m-1}(C_a\ H^a)\ H) = \uparrow (C_a, m)\boldsymbol{\eta}\ (V^{m-1}\ H^a\ H)$
mul :	$\boldsymbol{\eta}\ (V[j \leftarrow V\|_i]) = Eq[i,j]\boldsymbol{\eta}\ (V)$ if $i < j\ \wedge\ occ(V, V\|_k) = \{k\}\ \forall 1 \leq k \leq j \leq n$
where	V^n is any tuple of n variables, H^a is a tuple of a terms $(a \geq 0)$, H is any tuple.

Table 2: Definition of η

between. From an algebraic point of view, $\mathcal{C}$-expressions are a monoid having the functional composition as the binary operation and π as the identity [Bellia 93]. Hereafter, possibly indexed, T will denote $\mathcal{C}$-expressions.

3.2 Mapping terms into $\mathcal{C}$-expressions

Tuples can be mapped into $\mathcal{C}$-expressions through a mapping η, which associates constructions with constructors, Eq with multiple occurrences of variables and π with distinct variables. The formal definition of η is given in Table 2. For instance, if H is a tuple of four terms with constructors A_2, B_1, C_1, 0_0,

$$\begin{aligned} H &= ((A\ y\ y)\ (B\ y)\ (A\ 0\ z)\ (C\ z)),\ \text{then} \\ \eta(H) &= \uparrow(A,1)\uparrow(B,3)\uparrow(A,4)\uparrow(0,4)\uparrow(C,5)Eq[1,2]Eq[2,3]Eq[4,5]\pi. \end{aligned}$$

Hereafter, for notational convenience, we will write $C_a(j)$ instead of $\uparrow (C_a, j)$, $[i,j]$ instead of $Eq[i,j]$, and omit π. Hence the $\mathcal{C}$-expression, defined above, becomes.

$$\eta(H) \quad = \quad A(1)B(3)A(4)0(4)C(5)[1,2][2,3][4,5].$$

η behaves like a variable abstraction operator on tuples of terms (where tuples are universally quantified). In particular, if H is any tuple with n (not necessarily distinct) occurrences of variables, then for arbitrary tuples V^n of n distinct variables the application of $\eta(H)$ to V^n computes a variant of H, i.e. $\eta(H)V^n \approx H$. Moreover, η maps tuples into a proper subset of $\mathcal{C}$-expressions, called $\mathcal{C}E\Pi$. $\mathcal{C}E\Pi$-expressions, are such that all Eqs occur after any construction.

Compared with tuples, $\mathcal{C}$-expressions have a very compact algebraic structure and the two following appealing features. The variant relation $\approx$ on $(\Sigma, \mathcal{V})^*$

[Eder 85], becomes a congruence $\sim$ on $\mathcal{C}$-expressions, when we define $T_1 \sim T_2$ if and only if $T_1 H \approx T_2 H$ for each tuple H. The congruence $\sim$ is fundamental for the calculus on $\mathcal{C}$-expressions, described in [Bellia 93], since it is based on the replacement of $\mathcal{C}$-expressions with congruent and simpler $\mathcal{C}$-expressions. The second appealing feature is in the subsumption ordering $\leq$ on $\Sigma(\mathcal{V})^*$ which becomes composition ordering on $\mathcal{C}$-expressions, i.e. $T\ T' \leq T$ for each $\mathcal{C}$-expression T and T'. Hence $\mathcal{C}$-expressions were used as a variable free abstract syntax for tuples and we characterized unification on $\mathcal{C}$-expressions without resorting to the structure of substitutions [Bellia 93]. The semi-lattice $\mathcal{C}E\Pi/\sim$ is isomorphic to the semi-lattice $(\Sigma(\mathcal{V})^* - \{\perp\})/\approx$. This means that a not undefined tuple corresponds to any $\mathcal{C}E\Pi$-expression, and vice versa. Outside $\mathcal{C}E\Pi$, $\mathcal{C}$-expressions yield the computation of mgi and the solution of the corresponding unification/instantiation problems. Such a computation can result in a defined tuple or in $\perp$.

3.3 Unification as reduction of $\mathcal{C}$-expressions

Let $\eta(H)$ be any $\mathcal{C}E\Pi$-expression. $T = Eq[i,j]\eta(H)$ is thus an mgi problem based on $\mathcal{C}$-expressions. Solving it entails finding a $\mathcal{C}E\Pi$-expression $T' \sim T$ if and only if $\{H|_i, H|_j\}$ is unifiable. Otherwise, an equivalent $\mathcal{C}E\Pi$-expression cannot be found, thus denoting failure. [Bellia 93] provides $\mathcal{C}$-expressions with a reduction rule system which computes T' if any. The idea on which the system is based is to shift to the right (end) of the $\mathcal{C}$-expression the equality occurring at the beginning. Roughly, a reduction takes into account an equality and the adjacent constructor. If the constructor and the equality do not insist on the same indices, the indices are updated to allow the right shift. Conversely, if the two combinators do insist on the same indices a new construction and, possibly, new equalities are inserted to instantiate and propagate unification on corresponding descendants. The resulting system solves mgi problems with a complexity which is, in the worst case, exponential in time and space with the size of the input H (i.e. the number of indices in T), similarly to Robinson's first algorithm [Robinson 65]. Another limitation of this reduction system is that there is no provision for rational infinite terms.

In the next section we show how to overcome both problems by extending $\mathcal{C}$-expressions and reformulating the entire rule system.

4 Extended $\mathcal{C}$-expressions

4.1 Extended combinators

The extended combinators are defined in Table 3, as functions on tuples. At a syntactic level, they extend combinators of $\mathcal{C}$-expressions in two ways:

a) arguments are sets,

π :	$\Sigma(\mathcal{V})^n \rightarrow \Sigma(\mathcal{V})^n$	(*identity*)
$\uparrow$:	$\Sigma_a \rightarrow \Re_\omega \rightarrow \Sigma(\mathcal{V})^n \rightarrow \Sigma(\mathcal{V})^n$	(*construction*)
Eq :	$\Re_\omega \rightarrow \Sigma(\mathcal{V})^n \rightarrow \Sigma(\mathcal{V})^n$	(*equality*)

$$\boldsymbol{\pi}\ H = H$$

$$\uparrow (\mathbf{C_a})\,(\mathbf{P} \cup \{\mathbf{p}\})\ H = \begin{cases} \uparrow (\mathbf{C_a}(\mathbf{P})\ (H[x \leftarrow C_a(x_1 ... x_a)]) \\ \quad \text{if } H|_p = x \in Vars(H) \wedge x_1, ..., x_a \in \mathcal{V} - Vars(H) \\ \uparrow (\mathbf{C_a})\,(\mathbf{P})\ H \quad \text{if } H|_p = C_a(H|_{p.1} ... H|_{p.a}) \\ \bot \quad \text{if } H|_p = C'_{a'}(H|_{p.1} ... H|_{p.a}) \wedge C'_{a'} \neq C_a \end{cases}$$

$$\mathbf{Eq[P]}\ \ H = mgi(P, H)$$

Table 3: $\mathcal{EC}$-combinators

b) paths instead of indices occur in the arguments.

Point a) is syntactic sugar which allows grouping, into a single combinator, compositions of equalities which share one of the indices, as well as compositions of constructions with the same constructor. Point b) extends constructions with filtering features (see cases 2 and 3 of the definition in Table 3) which check the tuples for the occurrence of the constructor at the positions indicated in the paths. Hence, the $\mathcal{EC}$-combinators are such that:

π (*identity*), maps identically Σ-tuples into Σ-tuples.

$\uparrow (C_a)(P)$ (*construction*), usually denoted as $C_a(P)$, applies to a constructor C_a of arity a, to a set P of paths, and maps tuples into tuples (which are the most generals ones among the instances) whose subterms at paths in the set P have C_a as their main constructor.

$Eq(P)$ (*equality*), usually denoted as $[P]$, applies to a set of paths P, and maps tuples into tuples (which are the most general ones among the instances) whose subterms at paths in the set P are equal.

Hereafter, we will use, possibly indexed, $\mathcal{X}$ for the combinator Eq and for the combinator $\uparrow (C_a)$, for arbitrary construction C_a; and, possibly indexed, F will be used for $\mathcal{EC}$-expressions.

4.2 $\mathcal{EC}$-expressions and the congruence $\simeq$

$\mathcal{EC}$-expressions are any composition ($\circ$) of extended combinators. However, the use of sets as arguments of combinators and its interpretation as a composition makes composition on $\mathcal{EC}$-expressions a commutative operator. This is shown in Table 4 which contains the inference rules for the application of $\mathcal{EC}$-expressions to tuples. In particular, the rules *constr* and *eqns* show that combinators on a set can be replaced by a composition of combinators on subsets of it, according to point a) above. Rule *comm* says that any commutation of $\mathcal{EC}$-expressions computes identically i.e. the order in which combinators apply is no longer relevant, and $\mathcal{EC}$-expressions are a commutative monoid. Hereafter, we will use possibly indexed F for $\mathcal{EC}$-expressions, moreover, for notational convenience, symbol $\circ$ will be omitted. We can use the rules in Table 4 as reduction rules for the computation of the application of $\mathcal{EC}$-expressions to tuples. For example:

$$
\begin{array}{ll}
[1,2]S_1(1,3)A_0(2.1)\,(x\,y\,z) & \\
\quad\Downarrow & \text{by } comm,\ constr \text{ and definition of construction} \\
[1,2]A_0(2.1)\,((S\,x')\,y\,(S\,z')) & \\
\quad\Downarrow & \text{by } comm \text{ and definition of equality} \\
A_0(2.1)\,((S\,x')\,(S\,x')\,(S\,z')) & \\
\quad\Downarrow & \text{by definition of construction} \\
((S\,A)\,(S\,A)\,(S\,z')) &
\end{array}
$$

Let F be any $\mathcal{EC}$-expression, and H any tuple. Then the application of F to H, i.e. $F\,H$, computes a tuple H' in Σ^*, if any, which is the most general instance of H which satisfies all equalities in F and has all constructors indicated by the constructions in F. As a matter of fact, not for all H in Σ^* is each F such that $F\,H$ computes a tuple in Σ^*. However, when $F\,H$ does compute a tuple, i.e. $FH \in \Sigma^*$, we have the following congruence relation:

$$\forall H \text{ s.t. } F\,H \in \Sigma^*,\ F \simeq F' \text{ iff } FH \approx F'H$$

Note that since the congruence is symmetric, $F \simeq F'$ only if $\{H \mid F\ H \in \Sigma^*\} = \{H \mid F'\ H \in \Sigma^*\}$. Moreover we require $FH \approx F'H$ instead of $FH = F'H$ because we are working on variants rather than on tuples. Hence $\simeq$ will be used to model, on $\mathcal{EC}$-expressions, the variant relation $\approx$ on tuples. If necessary, the inference rules in Table 4 can be easily reformulated into rules for the congruence, even though, in such a case, they would not define a complete inference system.

4.3 Mapping $\mathcal{C}$-expressions into $\mathcal{EC}$-expressions

A mapping $\mathcal{E}$ from $\mathcal{C}$-expressions into $\mathcal{EC}$-expressions is given in Table 5. For this purpose the following two operators have been used:

comm	$$\frac{\mathcal{X}_1(P)H \Rightarrow H_1 \qquad \mathcal{X}_2(Q)H_1 \Rightarrow H_2}{(\mathcal{X}_1(P) \circ \mathcal{X}_2(Q))H \Rightarrow H_2}$$	
	$$\frac{\mathcal{X}_1(P)H_1 \Rightarrow H_2 \qquad \mathcal{X}_2(Q)H \Rightarrow H_1}{(\mathcal{X}_1(P) \circ \mathcal{X}_2(Q))H \Rightarrow H_2}$$	
constr	$$\frac{C_a(P)H \Rightarrow H_1 \qquad C_a(Q)H_1 \Rightarrow H_2}{C_a(P \cup Q)H \Rightarrow H_2}$$	
eqns	$$\frac{[P]H \Rightarrow H_1 \qquad [Q]H_1 \Rightarrow H_2 \qquad P \cap Q \neq \{\}}{[P \cup Q]H \Rightarrow H_2}$$	

Table 4: Inference rules for the application

Compactification [2] $\mathcal{X} \oplus F$ computes the $\mathcal{EC}$-expressions obtained by $F' = \mathcal{X} \circ F$ replacing all constructions $\mathcal{X}_1(P_1'), ..., \mathcal{X}_k(P_k)$ occurring in F' with $\mathcal{X}(P_1 \cup ... \cup P_k)$ if $\mathcal{X}_i = \mathcal{X}_j = \mathcal{X}$ for $1 \leq i, j \leq k$, and all equalities $\mathcal{X}(P_1), ..., \mathcal{X}(P_k)$ occurring in F' with $\mathcal{X}(P_1 \cup ... \cup P_k)$ if $P_i \cap P_j \neq \{\}$ for $1 \leq j \leq k$.

Replacement $F\{p \leftarrow p'\}$ computes the $\mathcal{EC}$-expression obtained by F replacing, for each pair of paths p, p', each occurrence of p in F with p'.

A step by step example of the computation of the mapping is given below for the expression $C_2(1)[1,2]S_1(1)$:

$$\begin{aligned}\mathcal{E}(C_2(1)[1,2]S_1(1)) &= C_2(1) \oplus (\mathcal{E}([1,2]S_1(1))\{1 \leftarrow 1.1, 2 \leftarrow 1.2\}) = \\ &= C_2(1)[1.1, 1.2]S_1(1.1).\end{aligned}$$

The transformation $\mathcal{E}$ maps $\mathcal{C}$-constructions into $\mathcal{EC}$-constructions, and $\mathcal{C}$-equalities into $\mathcal{EC}$-equalities. In this way, each construction $C_b(j)$, in T, at an index j, corresponds to a construction C_b in $\mathcal{E}(T)$ at path q. The path q depends on the index j and on the structure of the composition in which $C_b(j)$ is nested. When T occurs as a right hand expression of a composition

[2] This operator is similar to the one defined in [Martelli 82] for sets of multi-equations. One difference is that our compactification involves constructions in addition to equalities. Moreover, the algebraic framework is different and in this respect our operator behaves like a generalization of composition and can be expressed in terms of it:
$$\mathcal{X}(P) \oplus \mathcal{X}'(Q) \circ F \quad = \quad \begin{cases} \mathcal{X}(P \cup Q) \oplus F & \text{if } P \cap Q \neq \{\} \wedge \mathcal{X} = \mathcal{X}' \\ \mathcal{X}(Q) \circ (\mathcal{X}(P) \oplus F) & \text{otherwise} \end{cases}$$

$$\begin{array}{ll}\mathcal{E}(C_a(i)T) & = C_a(i) \oplus (\mathcal{E}(T)\{(i+j-1).p \leftarrow q\}) \\ \mathcal{E}([i,j]T) & = [i,j] \oplus \mathcal{E}(T) \\ & \\ \text{where} \quad q = & \left\{ \begin{array}{ll} i.j.p & \text{for } 1 \leq j \leq a \\ (i+j-a).p & \text{for } j > a \end{array}\right. \end{array}$$

Table 5: Definition of $\mathcal{E}$

$C_a(i)T$, the replacement, involved in the definition of $\mathcal{E}$, computes the path q of each construction, $C_b(j)$, (and equality) in $\mathcal{E}(T)$. In this case, in fact, the $\mathcal{C}$-combinator $C_a(i)$ would map, given any H, all terms $(TH)|_i ... (TH)|_{i+a-1}$ into subterms of $((C_a(i)T)H)|_i$. So that all constructions (and equalities) in $\mathcal{E}(T)$ which apply to a path $i, ..., i+a-1$ must have in $\mathcal{E}(C_a(i)T)$ such paths replaced by $i.1, ..., i.a$. Hence we have the following lemma.

Lemma 1 *Let T be any $\mathcal{C}$-expression and C_a be any constructor. Then, T contains a $\mathcal{C}$-construction C_a at an index i, if and only if $\mathcal{E}(T)$ contains $\mathcal{EC}$-construction C_a at a path p and p is such that, for all H on which T is ranging over, $(TH)|_p$ is a term with C_a as its main constructor.*

The following theorem derives from the definition of $\mathcal{E}$ and from the properties of the congruences $\sim$ and $\simeq$ on $\mathcal{C}$-expressions and $\mathcal{EC}$-expressions, respectively.

Theorem 1 *Let T_1 and T_2 be two $\mathcal{C}E\Pi$-expressions. Then*

$$T_1 \sim T_2 \textit{ iff } \mathcal{E}(T_1) \simeq \mathcal{E}(T_2)$$

Proof Since T_1 and T_2 are $\mathcal{C}E\Pi$-expressions, their form is $C_1 \circ E_1$, $C_2 \circ E_2$ respectively, where C_1, C_2 are compositions of constructions alone and E_1, E_2 are compositions of equalities alone. Then by definition of $\mathcal{E}$ and since $\sim$ is a congruence, the proof can be reduced to proving separately:

$$C_1 \sim C_2 \text{ iff } \mathcal{E}(C_1) \simeq \mathcal{E}(C_2) \tag{1}$$

$$E_1 \sim E_2 \text{ iff } \mathcal{E}(E_1) \simeq \mathcal{E}(E_2) \tag{2}$$

Consider sentence (1). If C_1 is a single construction then C_2 must be exactly the same and the sentence holds. Otherwise, let $C_1 = C_a(i) \circ C_1^r$, $C_2 = C_b(j) \circ C_2^r$ for $i \neq j$. Since $C_1 \sim C_2$, the $\mathcal{C}$-expression C_1^r must contain a construction $C_b(j')$ and analogously, C_2^r must contain $C_a(i')$ [Bellia 93]. Moreover, since $C_a(i)$ and $C_b(j)$ are left hand expressions in the composition of C_1 and C_2, respectively, then, for any H, C_a and C_b are the main constructors of the terms $(C_1H)|_i \approx (C_2H)|_i$ and $(C_1H)|_j \approx (C_2H)|_j$. By Lemma 1, both $\mathcal{E}(C_1)$ and

$\mathcal{E}(C_2)$ contain the $\mathcal{EC}$-construction C_a at path i and the $\mathcal{EC}$-construction C_b at path j. Hence $\mathcal{E}(C_1)$ and $\mathcal{E}(C_2)$ contain only constructions and these are all the same. Finally, by definition of $\mathcal{EC}$-construction, we note that for two $\mathcal{EC}$-expressions, $\mathcal{E}(C_1)$ and $\mathcal{E}(C_2)$, containing only constructions, $\mathcal{E}(C_1)\simeq\mathcal{E}(C_2)$ if and only if $\mathcal{E}(C_1)$ and $\mathcal{E}(C_2)$ differ only in terms of a permutation of the constructions.

The proof of sentence (2) is similar and quite immediate since E and $\mathcal{E}(E)$ differ only in terms of compactification.

4.4 $\mathcal{EC}E\Pi$ normal form

A mapping η_L from tuples to $\mathcal{EC}$-expressions can now be easily defined as the composition of η and $\mathcal{E}$, i.e. $\eta_L = \mathcal{E}\eta$. Given any H, η_L has as many equalities as there are different variables occurring more than once in H. In addition, η_L contains as many constructions as there are different constructions symbols in H. For instance,

$$
\begin{array}{lcl}
H & = & ((A\ y\ y)\ (B\ y)\ (A\ 0\ z)\ (C\ z)) \\
\eta(H) & = & A(1)B(3)A(4)0(4)C(5)[1,2][2,3][4,5] \\
\eta_L(H) & = & [\{1.1, 1.2, 2.1\}][\{3.2, 4.1\}]A(\{1,3\})B(\{2\})0(\{3.1\})C(\{4\}
\end{array}
$$

Analogously to η, η_L maps tuples into a proper subset of $\mathcal{EC}$-expressions, called $\mathcal{EC}E\Pi$. We say that F is in $\mathcal{EC}E\Pi$ normal form if and only if all the following conditions hold:

a) each pair of constructions $C_a(P)$ and $C'_a(Q)$ applies to different constructors, i.e. $C_a \neq C'_a$ and $P \cap Q = \{\}$

b) each pair of equalities $[P]$, $[Q]$, is such that $P \cap Q = \{\}$.

c) for each equality $[P]$ there is at most one $C_a(Q)$ such that $P \cap Q \neq \{\}$. In this case $P \cap Q$ is a singleton set.

Let H be any tuple and $F = \eta_L(H)$. Condition a) states that the $\mathcal{EC}E\Pi$-expression F contains only one occurrence of each C_a appearing in H. Of course, different constructions must insist on different paths ($P \cap Q = \{\}$). Condition b) states that equalities define a partition on paths, so that all paths in the same class are collected into a single equality. Condition c) states that for each equality there is at most one construction which shares paths with the equality. Moreover condition c) requires that only one path that appears in the equality also appears in the construction ($P \cap Q$ is a singleton set).

4.5 Rational terms and $\mathcal{EC}$-expressions

The calculus of $\mathcal{EC}$-expressions is not a merely syntactic extension of $\mathcal{C}$-expressions. As a matter of fact, $\mathcal{EC}E\Pi$ contains a presentation for rational terms as well, i.e.

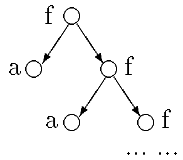
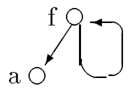


Figure 2: $t = (f \; a \; t)$



[1,1.2]f(1)a(1.1)

Figure 3: $t = (f \; a \; t)$

terms which may have infinitely many symbols but only a finite set of subterms. As an example, see term $t = (f \; a \; t)$ in Figure 2 which has only two distinct subterms: itself and the term a. Rational terms can be finitely represented by rational trees which are directed graphs that merge all the roots of the same subtrees. In $\mathcal{EC}$-expressions they are expressed by equalities which collect all the paths to the roots of such subtrees. For example, the graph and the $\mathcal{EC}$-expression in Figure 3 represent the term t defined above. The extension of $\Sigma(\mathcal{V})^*$ to rational terms and of unification on terms to unification on rational terms, mgu_ω, is discussed in [Huet 76, Colmerauer 82]. According to this extension, we can, correspondingly, extend mgi to mgi_ω defining $mgi_\omega(P, H) = H\theta$ for $\theta = mgu_\omega(P, H)$ and P any subset of paths in H.

5 A linear reduction system for mgi

An mgi problem can now be formulated in $\mathcal{EC}$-expressions. Let H be any tuple, F be an $\mathcal{ECE}\Pi$-expression such that $F \simeq \eta_L(H)$, and $P \subseteq \text{Paths}(H)$ be any subset of paths in H. Then $[P]F$ is an mgi problem based on $\mathcal{EC}$-expressions. Solving $[P]F$ entails finding an $\mathcal{ECE}\Pi$-expression F' such that $F' \simeq \eta_L(mgi(P, H)) = [P]\eta_L(H)$. In this section we provide $\mathcal{EC}$-expressions with a reduction system for mgi problems which is linear with the symbols in H. The system consists of the two rules shown in Table 6. $F \circ \mathcal{X}(P)$ can be read as an $\mathcal{EC}$-expression containing F and the combinator $\mathcal{X}(P)$.

A parallel with the rules for *Tree Syntactic Unification* (TSU) [Jouannaud 91] can be useful to understand the extent of the rule. *Merge* incorporates the rules Coalesce and Eliminate of TSU, which both deal with the substitution in a tuple H of a variable x with a subterm t of H, i.e. $H[x \leftarrow t]$. In $\mathcal{EC}$-expressions this substitution is expressed by issuing an equality on the set of occurrences of x in

$[Q] \circ F \circ [P] \ \ \rightarrow_{R_L} F \circ [Q \cup P]$ if $P \cap Q \neq \{\}$		*merge*
$[Q] \circ F \circ C_a(R) \ \ \rightarrow_{R_L} [Q] \circ F \circ C_a(R - P') \circ [P.1] \circ ... \circ [P.a]$ if $R \cap Q = P' \cup \{i\} = P \ \wedge \ P' \neq \{\}$		*reduce*

Table 6: The reduction system R_L for unification

H, i.e. $P = \{ps.t.H|_p = x\}$ and the occurrence of the subterm t, i.e. q such that $H|_q = t$, i.e. $[P \cup \{q\}]$. When q already occurred in an equality $[Q]$, i.e. $q \in Q$, *merge* involves the entire set Q, so we have $[P \cup Q]$. Our second rule, *reduce*, incorporates Decompose of TSU, which propagates unification from terms to corresponding subterms. Note that $P \cap R$ needs to be split into $P' \cup \{i\}$ because at least one path needs to be left as an argument of C_a. The remaining rules of TSU are implicitly obtained by the reduction process into the normal form, which may stop with an irreducible $\mathcal{EC}$-expression. In particular, rule Conflict of TSU corresponds to an irreducible $\mathcal{EC}$-expression having two constructions $C_a(P)$ and $C_b(Q)$ such that $P \cap Q \neq \{\}$ (see Section 5.3 for an example of functor mismatch).

A *reduction* of an $\mathcal{EC}$-expression F produces a new $\mathcal{EC}$-expression F' obtained by applying one of the two rules, *merge* or *reduce* to F. A reduction step is denoted $F \rightarrow_{R_L} F'$. Let $F_0 = [P]F$ be an *mgi* problem, the system is applied and always produces a finite sequence $F_0 \rightarrow_{R_L} F_1 \rightarrow_{R_L} F_2 \rightarrow_{R_L} ... \rightarrow_{R_L} F_n$, of reductions. $F_n (n \geq 0)$, also denoted $R^*_L([P]F)$, is (R_L-)irreducible, that is no rule applies to it, and we will see that an $\mathcal{EC}$-expression is irreducible in R_L if and only if it is either an $\mathcal{ECE}\Pi$-expression or it corresponds to a failure.

We now show an example of R_L-reduction, choosing the same example in [Paterson 78], where $(A\ (B\ v)\ (C\ u\ v))$ is unified with $(A\ (B\ w)\ (C\ w\ (D\ x\ y)))$. This unification entails computing

$$mgi(\{1,2\}((A\ (B\ v)\ (C\ u\ v))\ (A\ (B\ w)\ (C\ w\ (D\ x\ y)))))$$

For readability, in the following reductions, redexes are underlined and the notation is simplified omitting separator dots in the paths.

$F = [1,2]\eta_L((A\ (B\ v)\ (C\ u\ v))\ (A\ (B\ w)\ (C\ w\ (D\ x\ y)))) =$
$[1,2]A(1,2)B(11,21)C(12,22)D(222)[111,122][211,221]$
$[1,2]A(1)[11,21][12,22]B(11,21)C(12,22)D(222)[111,122][211,221]$
$[1,2]A(1)[11,21][12,22]B(11)[111,211]C(12,22)D(222)[111,122][211,221]$
$[1,2]A(1)[11,21][12,22]B(11)C(12,22)D(222)[111,122,211,221]$
$[1,2]A(1)[11,21][12,22]B(11)C(12)[121,221][122,222]D(222)[111,122,211,221]$
$F' = [1,2]A(1)[11,21][12,22]B(11)C(12)D(222)[111,121,122,211,221,222]$

F' is irreducible, because *merge* obviously cannot apply, and *reduce* does not since all constructors apply to singleton sets. Moreover, let $\theta = mgu\{H|_1, H|_2\}$, then

$$\begin{aligned} H' &= mgi(\{1,2\}, H) \\ &= (A\ (B\ (D\ x\ y))\ (C\ (D\ x\ y)\ (D\ x\ y))) \\ &\ (A\ (B\ (D\ x\ y))\ (C\ (D\ x\ y)\ (D\ x\ y))) \\ &\approx H\theta \end{aligned}$$

and for the correctness of R_L, we have $F' \simeq \eta_L(H')$.

5.1 Soundness

The soundness of R_L straightforwardly comes from the $\simeq$ congruence of the left and right sides of $\rightarrow_{R_L}$ in each rule. In particular, the first rule merges a pair of equalities sharing paths into a single equality. The second rule replaces $C_a(R)$ by $C_a(R-P')$, adding the equalities $[\{j.1 | j \in P' \cup \{i\}\}] \circ \cdots \circ [\{j.a | j \in P' \cup \{i\}\}]$ to guarantee the equivalence. Each equality thus shares at most one path with a construction.

5.2 Termination

Termination comes from the fact that *merge* and *reduce* apply finitely many times. In fact, R_L never creates new constructions. *Reduce* is applied by removing at least one path from a construction, and paths in a construction are finitely many. *Merge* is applied by removing at least one equality, and the equalities in each F, which may result from *reduce*, are finitely many.

Finally, when *merge* does not apply, then condition b) of $\mathcal{ECE}\Pi$ is satisfied. Further, if *reduce* does not apply either, then either F is an $\mathcal{ECE}\Pi$-expression or it contains at least one equality which violates condition c). In this case a mismatch is detected.

Theorem 2 *For each H of order n, and each set P of paths in H,*

$$R_L^*([P]\eta_L(H) \simeq \eta_L(mgi(P,H))$$

if $\{H|_i\ s.t. i \in P\}$ is unifiable. Otherwise $R_L^*([P]\eta_L(H))$ is not an $\mathcal{ECE}\Pi$-expression that points out the reasons for the failure.

5.3 Functor mismatches

Note that when $mgu\{H|_i \mid i \in P\}$ fails due to a functor mismatch, then R_L stops with an irreducible $\mathcal{EC}$-expression containing an equality which shares paths with two different constructions. Such an $\mathcal{EC}$-expression is not in $\mathcal{ECE}\Pi$ form and an $\mathcal{ECE}\Pi$-expression equivalent to it does not exist. This denotes a failure. For instance, if we attempt the unification of $(R\ y)$ with $(S\ x)$ we have $\eta_L((R\ y)\ (S\ x)) = R(1)S(2)$ and $R_L^*([1,2]R(1)S(2)) = [1,2]R(1)S(2)$ which is not reducible any further.

5.4 Rational terms

Finally, when unification involves rational terms, R_L finds a rational $\mathcal{ECE}\Pi$-expression, i.e. an $\mathcal{EC}$-expression in $\mathcal{ECE}\Pi$ form that has at least one equality containing both a path and a suffix of it. For instance, if we attempt the unification of x with $(S\ x)$ we have $\eta_L(x\ (S\ x)) = [1, 2.1]S(2)$ and $R_L^*([1,2][1,2.1]S(2)) = [1,2,2.1]S(2)$ by *merge*.

5.5 Laziness and partially evaluated expressions

The notions of data streams and non-strict evaluation can simplify writing a program (see lazy functional languages and languages which integrate functional and logic paradigms). Non-normal $\mathcal{EC}$-expressions may be used to denote terms which contain data streams as well as values computed by possibly non terminating computations [Hansson 82]. These expressions cannot be reduced to normal forms, but we can transform them into non-normal forms containing a greater normal part which can be used in the rest of the computation. For example, consider the computation of the $mgi(\{1.1, 1.2\}(P\ (C\ Y_f)(C\ Y_g)))$ where C and P are constructors and Y_f, Y_g are variables which are bound to the values computed by two, possibly divergent, functions $f(t_1)$ and $g(t_2)$ on terms. This can be denoted by the $\mathcal{EC}$-expression $[1.1, 1.2]P(1)C(1.1, 1.2)$. The computation of mgi can then be driven by the approximation we need, producing $[1.1, 1.2]P(1)C(1.1)[1.1.1, 1.2.1]$ which is a still a denotation for terms. Ultimately, data streams and non-strict computations extend the domain of terms with terms such as $(C\ A\ \bot)$. $\mathcal{EC}$-expressions provide a denotation for such terms and R_L is adequate for a demand driven computation which selects only the rule we need for the required approximation.

6 Complexity of R_L

Let F be an $\mathcal{ECE}\Pi$-expression and N be the number of paths occurring in F. We show that any mgi problem $[P]F$ is solved in $O(N)$ reduction steps by applying R_L rules.

Let R be the set of paths in F, and R^* be the minimum set which includes R and the set $\{j.i \mid j \in Q \wedge i \in [1,a]\}$ for each $C_a(Q)$ occurring in F. Let H be a tuple such that $\eta_L(H) \simeq F$, and M be the number of symbols in H. M is thus the number of paths in R^* and $M \geq N$. The values of M and N are bounds for the number of times *reduce* and *merge* can be applied.

Reduce is applied by removing at least one path from a construction. Hence it is applied no more than the number C of paths in the constructions, $C \leq N$.

Merge is applied by removing at least one equality. Hence, it is applied no more than the number of equalities occurring in F plus those generated by *reduce*. The latter ones are bound to the number of paths in the constructions, hence at most M.

Lemma 2 *Let H be any tuple containing M symbols, and $[P]\eta_L(H)$ be any mgi problem on H. Thus $[P]\eta_L(H)$ is reduced in $O(M)$ applications of R_L rules.*

Lemma 3 *Let H be any tuple containing M symbols, each unification problem on H can be solved in O(M) steps using R_L.*

The bound $O(M)$ was obtained without any constraints on the order in which the rules can be applied. This bound can be reduced to $O(N)$ if we apply a strategy in which *merge* is always applied first. In this strategy *merge* never applies to a pair of equalities both obtained by *reduce*.

The complexity of R_L is $O(N)$ if the application of each rule has $O(1)$ cost. In fact, this cost depends on the implementation of the operations of intersection and union on sets of paths. In our straightforward implementation, briefly discussed in the next section, such sets are union-find trees [Aho 74, Tarjan 75], and the complexity obtained with our implementation is $\alpha(N) * O(N)$.

7 Implementation

To give a more concrete view of how the system works, we now show an implementation, based on graph rewriting, of the two rules of system R_L. We limit the implementation of combinators to what is strictly necessary. In fact, a full implementation requires the introduction of reduction machines like the ones in [Bellia 93], and also involves rules for function application and composition.

An $\mathcal{ECE}\Pi$-expression F is represented by a collection of union-find trees[3],

[3] For the reader's convenience we recall that collections of union-find trees [Tarjan 75] represent disjoint sets or equivalence classes (classes of a partition). Each class is a tree where the root is the representative (element of the class). Basic operations on union-find trees are *find* and *union*. *Find* applies to an element and computes the representative (root) of the class which the element belongs to. *Union* applies to two classes and merges them by choosing, from the two roots, the new root of the resulting class (by appropriately setting pointers in the tree).

	constr	u-f	sons	weight
p				

Figure 4: Structure of a vertex representing path p

where each tree represents an equality and each vertex represents a path occurring in F. For each vertex, Figure 4, the following fields are available:

constr name of the constructor which applies to such vertex ($\neq NIL$ only for roots of union-find trees);

u-f pointer to the father in the union-find tree ($\neq NIL$ only for *non*-roots of union-find trees);

sons list of pointers to direct suffixes [4];

weight depth of the union-find tree (necessary to maintain balanced trees).

Let H be the tuple $((S\ x)\ (S\ 0))$ and $F = \eta_L(H) = S(1,2)0(2.1)[1.1,3]$. Suppose we want to compute $mgi(\{1,2\},F) = [1,2]T$. Figure 5 shows the collection of union-find trees representing $\eta_L(H)$. The equality originated by the mgi operation, hereafter called *unsolved*, is represented as a double box whose elements are the list of paths in the example {1,2}. Note that the unsolved equalities are implemented in a different way from the equalities, called *solved*. In fact, solved equalities are represented by the union-find trees: Pointers *u-f* in each vertex maintain the linked structure of such trees. The reduction process transforms unsolved equalities into solved ones, possibly generating new unsolved equalities. It consists in an iteration of the following steps:

1. Select an unsolved equality, according to the control strategy.
2. Try *merge*. This is accomplished by performing a *find* operation for each element of the unsolved equality. Depending on the result the following cases occur:
   - the *find* operations result in different vertices but at most one has a constructor. Then *union* is performed on such vertices. Merge is completed, hence we return to step 1 for a further reduction.
   - the *find* operations result in different vertices and more than one has a constructor. Then we can apply step 3, which tries rule reduce.

[4] $p.i$ is a direct suffix of p (or conversely p is a direct prefix of $p.i$) if p is any path and i is any positive integer.

3. Try *reduce*,

   - if at least two vertices have a different constructor then the process stops: the $\mathcal{C}$-expression is no longer reducible but it is not an $\mathcal{ECE}\Pi$ (Unification generates a mismatch).
   - if all vertices have the same constructor then, depending on the *weight* field, one of them is chosen, as the representative, and a *union* is performed. Unsolved equalities are generated for those elements which have corresponding descendants. Reduce is completed, hence we return to step 1 for a further reduction.

In the example above, step 1 selects the unique unsolved equality [1,2]. Step 2 performs $find(1) = 1$ and $find(2) = 2$. Both vertices 1 and 2 have S as their constructor, hence step 3 performs a *union* on the union-find trees of 1 and 2. Suppose 1 is chosen as the representative, the pointer *u-f* of 2 is set to 1. The constructor S in vertex 2 is deleted. The unsolved equality [1.1, 2.1] is generated, traversing fields *sons* of 1 and 2[5]. The situation resulting after the first reduction is shown in Figure 6. For the second reduction, step 1 selects the unique unsolved equality [1.1,2.1], step 2 performs $find(1.1) = 1.1$ and $find(2.1) = 2.1$. Since only 2.1 has a constructor a *union* is performed to complete *merge*. Because of the field *weight*, 1.1 is chosen as the root, hence the pointer *u-f* of 2.1 is set to 1.1. The constructor 0 in vertex 2.1 is deleted and is written in vertex 1.1. The structure resulting after the second and last reduction, representing the $\mathcal{ECE}\Pi$-expression computed, is shown in Figure 7.

8 Exploiting different control strategies

Deriving implementations of $\mathcal{EC}$-expressions and of system R_L, we can use different data structures or exploit different control strategies which are basic for the good behavior of well known (linear) unification algorithms. Firstly, we consider Paterson's linear algorithm. In this case we represent $\mathcal{ECE}\Pi$-expressions with graphs whose vertices, Figure 8, have a structure quite similar to the one adopted in the previous section. The second field, called *equiv*, is in this case a list of undirected edges representing *unmarked* (*unsolved*) equalities. The graph G representing the example in the previous section is shown in Figure 9. Superimposing the control strategy of Paterson and Wegman's algorithm means that *reduce* is only applied to equalities which correspond to *root* classes, if any, and such equalities are then marked as *solved* (or equivalently, the vertices of G, which are in correspondence to the paths occurring in such equalities, are disconnected). An equality corresponds to a root class if and only if its set of

[5] Note that, in the case in which only vertex 2 had sons, the unsolved equality would not have been generated but pointers to sons of vertex 2 would have been deleted and written in vertex 1, which is the representative of the class.

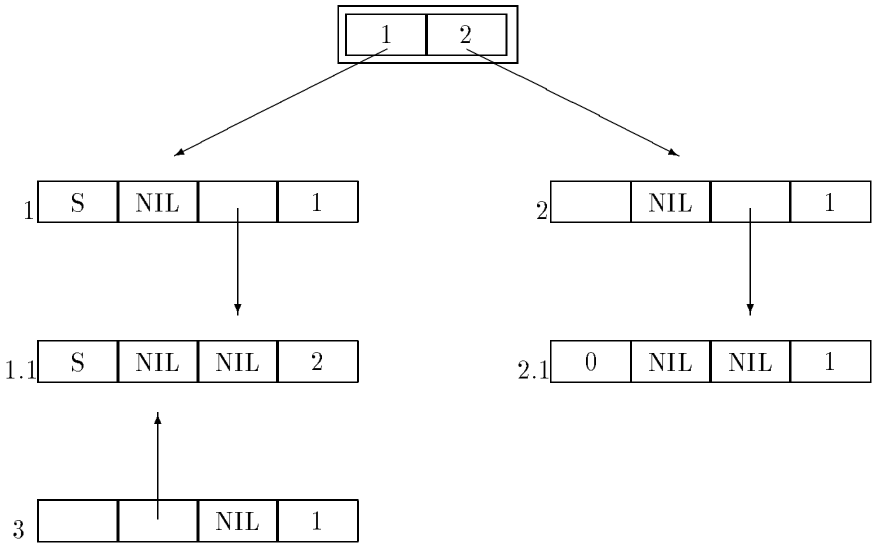


Figure 5: $[1,2]S(1,2)0(2.1)[1.1,3]$

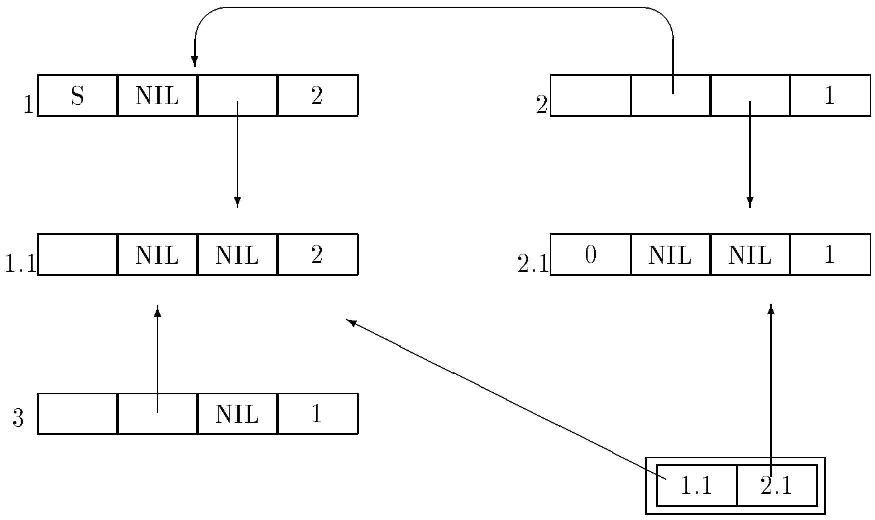


Figure 6: First reduction step: $[1,2]S(1)[1.1,2.1]0(2.1)[1.1,3]$

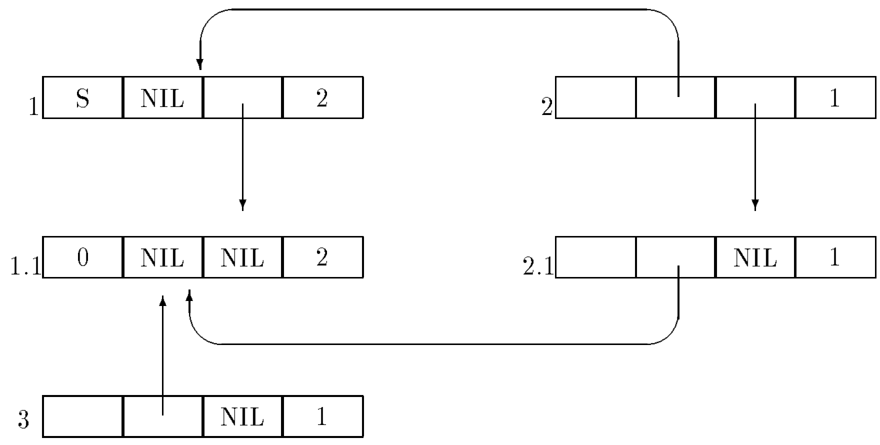


Figure 7: $[1,2]S(1)[1.1,2.1,3]0(1.1)\simeq((S\ 0)\ (S\ 0)\ 0)$

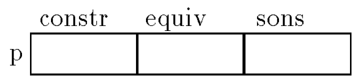


Figure 8: Structure of a vertex representing path p

paths contains no path which is a suffix of a path occurring in an unmarked equality. In the simple example in Figure 9 only the equality [1,2] corresponds to a root class. Hence [1,2]S({1,2}) is reduced generating [1.1,2.1], represented as an undirected edge in this implementation, see Figure 10, and [1,2] is marked solved. Note that this implementation leads to a natural embedding of *merge* into *reduce*. In fact, when equalities are implemented as linked structures of undirected edges, then $[P] \circ [Q]$ is the same as $[P \cup Q]$ for $P \cap Q \neq \{\}$ and *merge* has a constant cost. Finally, this control strategy constrains *reduce* to be applied when an equality, corresponding to a root class, exists. As a consequence, it stops reductions when an infinite rational term is involved. We have thus obtained an implementation of Paterson and Wegman's algorithm. It is not difficult to show that our rules have now complexity $O(N)$ and unify only on finite terms. This is not surprising since it is a consequence of the superimposed visiting strategy.

As another example, we could adopt a control strategy which concurrently applies reductions to all possible redexes which do not generate a conflict, i.e. do

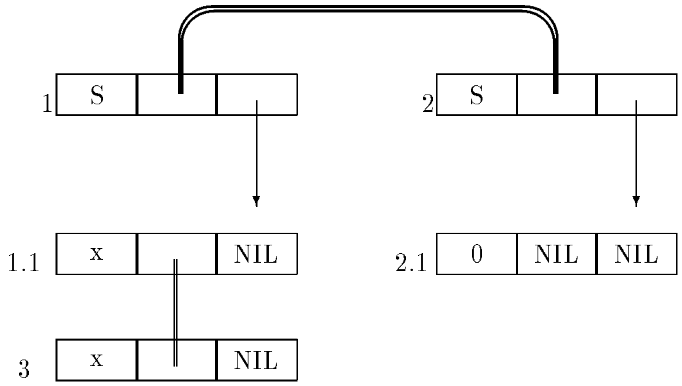


Figure 9: $[1,2]S(1,2)0(2.1)[1.1,3]$

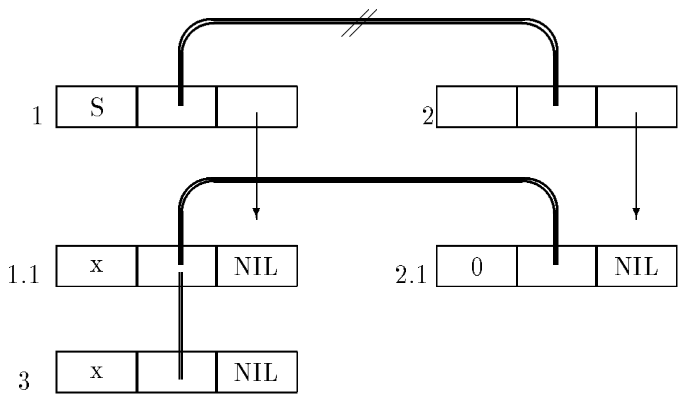


Figure 10: $[1,2]S(1)[1.1,2.1,3]0(2.1)$

not involve the same equalities. In this case a parallel algorithm, quite similar to the one defined in [Barklund 90], is obtained if we use union-find trees to represent $\mathcal{EC}E\Pi$-expressions.

9 Conclusions

In this paper we have introduced a two-rule reduction system for the lazy computation of unification. The most meaningful properties of the approach are:

1. The definition of a system which has linear complexity in time and in space and is fully abstract, i.e. each rule shows all it needs to compute at the estimated cost.

2. The introduction of combinators as functions on terms for constructor building and testing and for variable instantiation, which provides abstractions for terms.

3. The use of a specific combinator for the most general instance, *mgi*, which allows using $\mathcal{EC}$-expressions as denotations for unification problems.

4. The computation of unification as a normal form reduction process of $\mathcal{EC}$-expressions, which is at the basis of the following two points.

5. The choice of $\mathcal{EC}E\Pi$ as the normal form of $\mathcal{EC}$-expressions, which allows one to deal with cyclic structures and to cope with unification on rational terms.

6. The use of $\mathcal{EC}$-expressions as the denotation for expressing each step of the computation of the unification process, which allows $\mathcal{EC}$-expressions to be used as partially evaluated unifications in lazy calculi.

Unification is the kernel of any inference system, and obtaining a very efficient mechanism to compute unification is the key point for the implementation of machines for such systems. Attempts in this direction have deeply influenced the logic programming community since its origins [Moto-Oka 82] and also led to the design of a chip for unification [Robinson 85]. We believe that our work is a contribution to the definition of reduction machines [Berkling 75] since it incorporates efficient devices for term building and unification. Moreover, the introduction of partially evaluated unifications extends unification to a lazy operator that is able to cope with:

- possibly infinite computations [Friedman 76], as is the case when non-strict functions on terms are admitted,

- I/O and infinite data [Barendregt 90], as is the case when terms are extended with streams [Falaschi 90],

- computation control [Jaffar 94], as is the case when forward cutting of failure search trees is required,
- terms containing interpreted symbols [Hullot 80], as is the case of unification modulo an equational theory E.

There are many papers in the literature which deserve to be cited for their contribution to the theory, extensions and building procedures to efficiently compute unification. The Jouannaud and Kirkner's paper [Jouannaud 91] is fundamental reading on the subject. However, to the best of our knowledge, our system is the only one which is based on a combinatorial approach and, at the same time, provides reduction rules which compute unification at a linear cost. It would be very interesting to extend our combinatory approach to obtain a uniform framework, like the ones in [Jouannaud 91, Rydeheard 86], to express unification on non trivial theories (including commutativity, associativity and higher order).

This work is part of a project aimed at designing a reduction machine for the integration of functional computation and inference systems. The attempt to formalize a good notion of reduction based on combinators seemed to be hopelessly inefficient, especially for unification [Bellia 93]. This work shows that this is not the case. In the meantime, other projects with the same aim are under development and we refer to a recent paper [Lipton 97] for a discussion on the state of the art.

Acknowledgements

We would like to express our appreciation for the Referee's work, which was extremely accurate, stimulating and useful in helping us to improve the paper considerably.

References

[Aho 74] A.V.Aho, J.E.Hopcroft and J.D.Ullman, *The Design and Analysis of Computers Algorithms*, Addison Wesley Publishing Company, 1986.

[Aitkaci 90] H. Aït-Kaci, *The WAM: a (real) tutorial*, Digital Paris Research Laboratories, Paris 1990.

[Barendregt 90] H.P. Barendregt, *Functional Programming and Lambda Calculus*, Formal Models and Semantics, HandBook of Theoretical Computer Science, Vol. B, J.V. Leeuwen (ed.) Elsevier Science Publisher, 1990, 321-364.

[Barklund 90] J. Barklund, *Parallel Unification*, Ph.D Thesis, Upsala University, Sweden, 1990.

[Bellia 90] M. Bellia, M. Bugliesi and M.E. Occhiuto, *Combinatory Forms for Equational Programming: Instances, Unification and Narrowing*, CAAP'90, LNCS 431, Springer Verlag, 1990, 42-56.

[Bellia 93] M. Bellia and M.E. Occhiuto, *C-expressions: A Variable Free Calculus for Equational Logic Programming*, Theoretical Computer Science, 107, 1993, 209-252.

[Bellia 95] M. Bellia and M.E. Occhiuto, *Suprema of Open and Closed Formulas and their Application to Resolution*, Information and Computation, 117, 1, 1995, 136-150.

[Bellia 98] M. Bellia and M.E. Occhiuto, *Parallel Unification: Problems and Perspectives*, submitted for publication.

[Berkling 75] K.J.Berkling, *Reduction Languages for Reduction Machines*, Proc. Of the 2^{nd} Symposium on Computer Architecture, 1975, 133-140.

[Ciprietti 95] G. Ciprietti, *$\mathcal{EC}$-expressions:linear unification and lazy narrowing*, Thesis, Università di Pisa, 1995.

[Colmerauer 82] A. Colmerauer, *Prolog and Infinite Trees*, Logic Programming, K.L. Clark and SA. Tarnlund eds., Academic Press, 231-251, 1982.

[Corbin 83] J. Corbin and M Bidoit, *A Rehabilitation of Robinson's Unification Algorithm*, Information Processing 83, R.E.A.Mason (ed.) Elsevier Science Publisher B.V (North-Holland), IFIP, 1983, 909-914.

[Dershowitz 90] N.Dershowitz J.P. Jouannaud, *Rewrite Systems*, Formal Models and Semantics, HandBook of Theoretical Computer Science, Vol. B, J.V. Leeuwen (ed.) Elsevier Science Publisher, 1990, 245-320.

[Eder 85] E. Eder, *Properties of Substitutions and Unification*, J. of Symbolic Computation 1, 1985, 31-46.

[Falaschi 90] M. Falaschi and G. Levi, *Finite Failure and Partial Computations in Concurrent Logic Programming*, TCS 75, 1990, 45-66.

[Hullot 80] J.M. Hullot, *Canonical forms and unification*, Proc. 5^{th} Conf on Automated Reasoning, *LNCS* Vol. 87, 1980, 318-334.

[Friedman 76] D.P.Friedman and D.S. Wise *Cons should not evaluate its arguments*, in Automata, Languages and Programming,S. Michaelson and R. Milner (eds), Edinburgh Univ. Press, Edinburgh 1976, 257-284.

[Knight 89] K. Knight, *Unification: A Multidisciplinary Survey*, ACM Computing Survey, 21, 1, March 1989.

[Jaffar 84] J. Jaffar, *Efficient Unification over Infinite Terms*, New Generation Computing 2, 3 1984, 207-219.

[Jaffar 94] J. Jaffar and M.J. Maher, *Constraint Logic Programming: A Survey*, Journal of Logic Programming 19-20, 1994, 503-581.

[Jouannaud 91] J.P. Jouannaud and C. Kirkner, *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification*, Essays in Honor of Alan Robinson, Computational Logic, J.L. Lassez and G. Plotkin (eds.) The MIT Press, Cambridge, MA 1991, 257-321.

[Hansson 82] A. Hansson, S. Haridi, S.A Tärnulnd, *Properties of a Logic Programming Language*, K.L. Clark and S.A. Tärnulnd (eds.), Logic Programming, Academic Press, 1982, 267-280.

[Herbrand 30] J. Herbrand, *Investigation in proof theory: The properties of true propositions*, in From Frege to Gödel: A source book in Mathematical logic, 1879-1931, van Heijenoort J. (ed.), Harvard University Press, 1967, 525-581.

[Huet 76] G. Huet, *Resolution d'equations dans les language d'ordre 1,2,...,w*, Ph.D dissertation, Univ. de Paris VII, France, 1976.

[Lassez 88] J.L. Lassez, M.J. Maher, K. Marriot, *Unification Revisited*, In Minker(ed.), Foundation of Deductive Databases and Logic Programming, Morgan Kaufman, San Matteo(CA), 1988, 587-625.

[Lipton 97] J. Lipton and E. Chapman, *Some Notes On Logic Programming with a Relational Machine*, submitted for publication.

[Martelli 82] A. Martelli and U. Montanari, *An Efficient Unification Algorithm*, ACM Trans. on Progr. Lang. and Sys., 4, 2, 1982, 259-282.

[Moto-Oka 82] T. Moto-Oka (ed.), *Fifth Generation Computer Systems*, North-Holland, 1982.

[Mukai 83] K. Mukai, *A Unification Algorithm for Infinite Tree*, Proc. IJCAI'83, Karlsruhe, 1983, 547-549.

[Paterson 78] M.S. Paterson and M.N. Wegman, *Linear Unification*, J. Comput. System Science 16, 1978, 158-167.

[Siekmann 90] J.H. Siekmann, *Unification Theory*, Unification, in C. Kirchner (ed.), Academic Press, New York, 1990, 1-44.

[Reynolds 70] J.C. Reynolds, *Transformational System and Algebraic structure of Atomic Formulas*, Mach. Intell. 5, 1970, 135-151.

[Robinson 65] J.A. Robinson, *A Machine-oriented logic based on the resolution principle*, J.ACM 12, 1965, 23-41.

[Robinson 92] J.A. Robinson, *Logic and Logic Programming*, Comm. ACM 3, 35, 1992, 40-65.

[Robinson 85] P.Robinson, *The SUM: an AI Coprocessor*, Byte, June 1985, 169-180.

[Ruzicka 89] P. Ruzicka and I. Privara, *An Almost Linear Robinson Unification Algorithm*, Acta Informatica 27, 1, 1989, 61-71.

[Rydeheard 86] D. Rydeheard, R.Burstall, *A categorial unification algorithm*, Proc. of the Workshop on Category, Theory and Computer Programming , LNCS 240, Springer Verlag, 1986, 42-56.

[Tarjan 75] R. E. Tarjan, *Efficiency of a Good But Not Linear Set Union Algorithm*, JACM 22, 2, 1975, 215-225.