4

Compiling Pattern Matching by Term Decomposition

Laurence Puel
Ascánder Suárez

January 1990

Publication Notes

Laurence Puel's address is: Laboratoire d'Informatique LIENS URA CNRS 1327, Ecole Normale Supérieure, 45 rue d'Ulm, 72230 Paris Cédex 05, FRANCE. This article will also appear as the "Rapport de Recherche du LIENS 90-7."

© Digital Equipment Corporation and Ecole Normale Supérieure 1991

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for non-profit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by joint permission of the Paris Research Laboratory of Digital Equipment Centre Technique Europe (Rueil-Malmaison, France) and of the Laboratoire d'Informatique LIENS URA CNRS 1327 (Paris, France); an acknowledgement of the authors and individual contributors to the work; and all applicable portions of the copyright notice. All rights reserved.

Abstract

We present a method for compiling pattern matching on lazy languages based on previous work by Laville and Huet-Lévy. It consists of coding ambiguous linear sets of patterns using "Term Decomposition," and producing non ambiguous sets over terms with structural constraints on variables. The method can also be applied to strict languages giving a match algorithm that includes only unavoidable tests when such an algorithm exists.

Résumé

Nous présentons une méthode de compilation de l'appel par filtrage pour les langages paresseux dans le prolongement du travail de Laville et Huet-Lévy. Nous transformons des ensembles ambigus de motifs linéaires à l'aide de la "Décomposition des Termes" pour produire des ensembles non-ambigus de termes dont les variables sont munies de contraintes structurelles. Cette méthode peut aussi être appliquée à des langages stricts et donne un algorithme de filtrage ne nécessitant aucun travail inutile quand un tel filtrage existe.

Keywords

Compilation, Call by Pattern Matching, Term Decomposition, Sequentiality

Acknowledgements

We are grateful to Jean-Jacques Lévy who made numerous suggestions on the presentation of this work. We would like also to acknowledge helpful comments made by Gérard Huet and Hassan Aït-Kaci.

Contents

1 Introduction 1
- 1.1 Constrained Terms . . . 1
- 1.2 Pattern Matching . . . 2
- 1.3 Compilation . . . 2

2 Terms and Constraints 5
- 2.1 Terms . . . 5
- 2.2 Constraints . . . 6
- 2.3 Constraint Simplification . . . 8
- 2.4 Constrained Terms . . . 9
- 2.5 Substitution . . . 10

3 Term Decomposition 14
- 3.1 Decomposition . . . 14
- 3.2 Decomposition Procedure . . . 15
- 3.3 Decomposition Normalization . . . 16

4 Pattern Matching 17
- 4.1 Sequentiality . . . 19

5 Examples 22

6 Conclusion 24

References 26

1 Introduction

We are interested in compiling pattern matching in case of partially evaluated terms in order to do only necessary computations for the match. This is a kind of lazy computation over partially defined terms. In 1979 G. Huet and J-J. Lévy [5] defined a method for constructing match trees for non-ambiguous linear term rewriting systems. However, the application of their results to the problem of compiling pattern matching as in the ML language was not clear until 1988 when A. Laville [6, 7] showed that it is possible to use their method for ambiguous term rewriting systems with a given priority on rules. This priority is necessary to decide which rule has to be used in case of conflict. Laville designed a new match predicate that takes into account the priority when building the match trees. When this construction is successful, the leaves of the match tree form a *Minimal Extended Set of Patterns* equivalent (from the match point of view) to the original system in the case of finite signatures.

Our method is to code ambiguous ordered term rewriting systems into non-ambiguous ones over constrained terms. We replace the priority rule between left parts of the rewriting system by constraints over terms. Therefore the match predicate is that of Huet and Lévy but over constrained terms. Their results are then extended to these terms. Furthermore, as a result of the computation of the non-ambiguous set of terms of the system, we also obtain a characterization of the set of partially evaluated terms for which every matching algorithm will loop. We call it the strict set of the system. Although some algorithms may loop on other terms, an optimal algorithm, if it exists, will only loop on the strict set.

1.1 Constrained Terms

A term with variables is a representation of all ground terms obtained by replacing its variables by terms with no variables. A subset of a given set can be defined either by a description of its elements or as the complement of another subset. For example, a variable x represents the set of all the ground terms and $F(A, y)$ a subset of x. We can partition the set x into three subsets. First, the set of instances of $F(A, y)$. Then, the set of terms for which we can decide that they are not instances of $F(A, y)$. Finally, the set that contains partially evaluated terms of the form $F(\ldots)$ whose first argument cannot be evaluated (its computation loops) denoted $F(\bullet, y)$ as well as the non-evaluated term that we denote $\bullet$. With the set notation, this partition is written:

$$\{x\} = \{F(A, y)\} \cup \{x \mid x \not\in F(A, y)\} \cup \{\bullet, F(\bullet, y)\}$$

Following this idea, we define the concept of constrained terms and give some of their algebraic properties.

With this formalism an ordered ambiguous set of terms can be transformed into a non-ambiguous set of constrained terms. For instance, the set of terms $F(A, y), F(x, y)$ is ambiguous as the term $F(A, B)$ is an instance of both of them. The set of constrained terms $F(A, y), \{F(x, y) \mid x \neq A\}, \{x \mid x \neq F(\ldots)\}, \{\bullet, F(\bullet, y)\}$ is not ambiguous. Now a given term is an instance of exactly one of these constrained terms.

1.2 Pattern Matching

Call by pattern matching is one of the main features of the ML language [4, 10] and was inherited from HOPE [2]. It may be viewed as a generalization of the "case" statement of imperative languages. In ML, one can define one's own structural types and very easily write operations over them. We will introduce call by pattern matching by extending the Pascal definitions of "enumerated types"

In Pascal, it is possible to use the case statement to select among different cases by the value of an expression of an "enumerated type":

```
type T = (C1,...,Cn);          case x of
var x:T;                          C1 : <<exp1>>
...                             | ...
                                | Ci : <<expi>>
                                | otherwise  : <<exp>>
```

The natural extension of this construct is to allow matching not only constant values but more general data structures as in the following example in the language ML where there are two cases in the definition of the type of trees: `Leaf` to represent the leaves of trees and the constructor `Tree` for the other nodes.

```
type Tree =                    case tree of
   Leaf of number                  Leaf(3) → <<exp1>>
 | Tree of number*tree*tree;     | Tree(_,Leaf(_),_) → <<exp2>>
                                 | Tree(_,Tree(_,_,_),_) → <<exp3>>
                                 | otherwise → <<exp4>>

val tree: Tree = Tree(3,Leaf(2),Tree(4,Leaf(7),Leaf(9)));
 ...
```

In this example the value of the variable "tree" will match the second and the fourth cases but taking the first one as the priority holder, the expression `<<exp2>>` will be executed.

1.3 Compilation

If patterns are non-ambiguous, there is a decision procedure due to Huet and Lévy [5] that determines whether an optimal match exists for a set of patterns and, in the case where such a

match exists, produces a search tree that allows to compile the match problem. This method can be illustrated with the following example.

Suppose that we want to match pairs of terms `(x,y)` of Booleans by the set of patterns `(true,true)`, `(_,false)` and `(false,true)`. We choose to look first at a column that only contains constants (in our example the second one), and divide the patterns by the constants appearing in that column. The result is a transformed program in which there is always one column in the pattern to look at.

```
case (x,y) of                                  case y of
    ( true  , [true  ] ) → 1                       true  : (case x of
  | ( _     , [false ] ) → 2        ⇒                        true  → 1
  | ( false , [true  ] ) → 3                               | false → 3)
                                                 | false → 2
```

There are some sets of patterns, namely non-sequential patterns, for which the method of [5] fails. The typical example was proposed by Berry [1]:

`(A,A,_), (B,_,A), (_,B,B)`

In this example the patterns are non-ambiguous but there is no column in which we can make the decision. So if we want to avoid looping in the evaluation of this match, a parallel mechanism that inspects simultaneously all three columns is necessary.

But the restriction that patterns must be non-ambiguous is a burden to the programmer especially when the program contains data structures with many different constructors. This is one of the reasons why most programming languages that feature call by pattern matching accept ambiguities and impose a priority rule between different patterns. In this paper we do not discuss assignment of priorities. In ML and other programming languages, for instance, the order of patterns in the text is used and the programmer has to write the more specific cases before the more general ones. Another possibility to automatically assign higher priority to specific cases and still use textual ordering for those that are compatible. Both priority rules have the same expressive power as any set of patterns can be ordered to work exactly in the same way with any of them.

When ambiguities are allowed and a priority rule is imposed, the method of [5] does not apply directly, as shown below:

```
case (x,y) of
     ( _     , true ) → 1
   | ( false , _    ) → 2
   | ( _     , _    ) → 3
```

Now take any pair `(x,y)`, if `y=true` then the pair `(x,y)` matches the first case. Otherwise, if `x=false`, it matches the second one. Finally in every other case, it matches the third one. Remark that it is slightly subtle to find the set of pairs which match this three cases. The first case corresponds to any pair `(x,true)`, the second one to `(false,false)` and the third one to `(true,false)`. In this example, where both the first and the second column only have one constant, the method of [5] does not apply directly. It can be adapted (as in [6]) by imposing priorities to the patterns to make them non-ambiguous.

Our approach in this work is to use the data structure of constrained terms to represent sets of patterns ordered by priorities such that the disambiguating rule becomes part of the representation. In the previous example, the set of constrained terms that represents the match problem is:

$$(_,\mathtt{true}),\ (\mathtt{false},\neq\mathtt{true}) \text{ and } (\neq\mathtt{false},\neq\mathtt{true})$$

in which $\neq C$ represents any value different from C and the strict set is:

$$(_,\bullet),(\bullet,\neq\mathtt{true})$$

Notice that an algorithm that evaluates from left to right will also loop on the term $(\bullet,\mathtt{true})$ while an algorithm that evaluates this pair from right to left will not. With the non-ambiguous set of terms given above it becomes possible to apply the method of [5] and choose the second column as the one to look at first (where $\neq C$ is considered as a constant).

In [6], a program for the compilation of patterns with priorities was written in CAML [10]. The construction of the new set of non-ambiguous patterns is embedded in the control of the program. In our work, the transformation from ambiguous to non-ambiguous patterns will be achieved at the level of source programs. This makes the program transformation explicit and independent of the pattern matching process. Furthermore, the algorithm presented here produces very compact representations, especially in the matching of terms with arbitrarily large signature.

2 Terms and Constraints

2.1 Terms

Let X be a denumerable set of variables and Σ a set containing function symbols and an additional symbol $\bullet$. To each function symbol is associated its arity. For our purpose the language of terms $T(\Sigma, X)$ is defined by:

$$\text{terms}: \quad t \quad ::= \quad F(t_1, \ldots, t_n) \mid x \mid \bullet$$

where the function symbol F is a symbol of Σ of arity n, the variable x is in X and $t_1, \ldots, t_n$ are terms. The set of terms without variables is the set $T(\Sigma)$ of *ground terms*. The set of *partially evaluated terms* is the set $T(\Sigma - \{\bullet\}, X)$. A *linear term* is a term in which all the variables are different.

The set $\mathcal{O}(t)$ of occurrences of a term t is recursively defined by:

$$\begin{array}{rcl} \epsilon & \in & \mathcal{O}(t) \\ i.u & \in & \mathcal{O}(F(t_1, \ldots, t_n)) \text{ if } u \in \mathcal{O}(t_i)\ (1 \leq i \leq n) \end{array}$$

if $u \in \mathcal{O}(t)$ the subterm t/u of t is defined by:

$$\begin{array}{rcl} t/\epsilon & = & t \\ F(t_1, \ldots, t_n)/i.u & = & t_i/u \end{array}$$

Definition 1

1. *A (ground)* substitution σ*, is a mapping over terms defined by replacing a finite set of variables by (ground) terms which transforms any term t into $\sigma(t)$. The term $\sigma(t)$ is called an* instance *of t.*

2. *The quasi-ordering $\preceq$ over terms is defined by $t \preceq t'$ if there exists a substitution σ such that $\sigma(t) = t'$ and t is said to be a* prefix *of t'. Its extension to substitutions is defined by $\sigma \preceq \sigma'$ if and only if there exists a substitution η such that $\sigma' = \eta \circ \sigma$. Thus σ is said to be* more general *than σ'.*

3. *Two terms t and t' are* comparable *if either $t \preceq t'$ or $t' \preceq t$ and* compatible *or* unifiable *if there exists a substitution σ, such that $\sigma(t) = \sigma(t')$, in which case σ is a unifier of t and t'.*

4. *The* least upper bound *of two terms t and t', denoted $t \sqcup t'$ is the smallest term that has both t and t' as prefixes. The unifier that produces this bound if it exists is called the* most general unifier *(m.g.u.). The* greatest lower bound *of two terms t and t', denoted $t \sqcap t'$ is the greatest prefix of t and t' (with respect to $\preceq$).*

In the following it will be convenient to identify a term with the set of its ground instances. For example, let $\Sigma = \{F, A, B, \bullet\}$. The term $t = F(x, y)$ represents the set $\{F(A, A), F(B, A), F(\bullet, A), F(F(A, A), A), \ldots\}$. Any ground term t represents the set $\{t\}$. Two incompatible terms represent disjoint sets of ground terms.

The relation $\preceq$ is the opposite of the set inclusion of the ground instances. A substitution can be seen as an operation that allows to build terms from the root to the leaves. The special term $\bullet$ will denote terms that cannot be built as for instance those whose construction does not terminate; a substitution σ such that $\sigma(x) = \bullet$ can be assimilated to a construction that never ends.

Now we want to represent more precisely sets of terms; for instance the subset of all terms which are not instances of $F(x, y)$. Thus we classify all the terms in three parts: those that are instances of $F(x, y)$, the term $\bullet$ which represents terms that cannot be built and those that are instances of any $G(\ldots)$ with $G \neq F$ and $G \neq \bullet$. The last part represents terms for which we do know that they are not instances of $F(x, y)$ while the second part represents terms for which we cannot say anything. The subset of $F(x, y)$ of all terms different from $F(A, B)$ is the set of ground terms $\{F(\sigma(x), \sigma(y)) \mid \sigma(x) \notin \{A\} \text{ or } \sigma(y) \notin \{B\}\}$. With the finite signature $\Sigma = \{F, A, B, \bullet\}$, this set is represented as the union of $F(x, A)$, $F(x, F(y, z))$, $F(B, x)$ and $F(F(x, y), z)$. This representation depends on the number of elements of the signature, for instance using $\Sigma' = \{F, A, B, C, \bullet\}$ the representation as union of terms has two extra components: $F(x, C)$ and $F(C, y)$. With an infinite signature it is not possible to represent this set as a finite union of instances of terms. Notice that the two terms $F(\bullet, y)$ and $F(x, \bullet)$, which are instances of $F(x, y)$, do not belong to $\{F(\sigma(x), \sigma(y)) \mid \sigma(x) \notin \{A\} \text{ or } \sigma(y) \notin \{B\}\}$. It is more concise to represent those sets by terms with variables with constraints. This can be illustrated as follows:

1. $\{t = F(x, y) \text{ such that } t \neq F(A, A)\} = F(x \notin \{A\}, y) \cup F(x, y \notin \{B\})$
2. $F(B, A) \cup F(B, F(x, y)) \cup F(F(x, y), A) \cup F(F(x, y), F(z, t)) = F(x \notin \{A\}, y \notin \{B\})$
3. $F(F(x, y), A) \cup F(F(x, y), F(z, t)) = F(x \notin \{A, B\}, y \notin \{B\})$

We will now formally introduce the notions of constraint and of constrained term in order to represent such sets of ground terms. Roughly speaking, a constrained term is composed of a term and a constraint which is a predicate over the variables of the term. This predicate restricts the possible instances of the variables in subsequent substitutions as we will see below.

2.2 Constraints

Definition 2 *Let t and t' be two terms. The quasi-ordering $\sqsubseteq$ between two terms is defined by: $t \sqsubseteq t'$ if and only if there exists a term t'' such that $t \sqsubseteq_0 t'' \preceq t'$ where $\sqsubseteq_0$ is characterized by the following rules:*

Let x and y be two variables, F a symbol in Σ and t a term:

$$x \quad \sqsubseteq_0 \quad y$$

$$\begin{array}{rcl} F(t_1,\ldots,t_n) & \sqsubseteq_0 & F(t'_1,\ldots,t'_n) \textit{ if and only if for every } i\ (1 \leq i \leq n)\ t_i \sqsubseteq_0 t'_i \\ t & \sqsubseteq_0 & \bullet \end{array}$$

Lemma 1 *let t be a linear term and t' a term. $t \sqsubseteq t'$ if and only if there exist t'' such that $t \preceq t'' \sqsubseteq_0 t'$*

Proof: Let $U = \{u \in \mathcal{O}(t) \cap \mathcal{O}(t') \mid t'/u = \bullet\}$ and $t'' = t'[u \leftarrow t/u \mid u \in U]$. Clearly $t \preceq t'' \sqsubseteq_0 t'$.

When the name of variables in a term is not important (that will be the case for linear terms in the following) we will use the symbol Ω instead of the names of variables.

The greatest lower bound of two terms is equal to the one for the prefix ordering. The least upper bound can be characterized by the following rules: let l be an term, x a variable and F and G two different symbols in Σ.

$$\begin{array}{rcl} x \sqcup l & = & l \\ l \sqcup x & = & l \\ F(\ldots) \sqcup G(\ldots) & = & \bullet \\ F(l_1,\ldots,l_n) \sqcup F(l'_1,\ldots,l'_n) & = & F(l_1 \sqcup l'_1,\ldots,l_n \sqcup l'_n) \end{array}$$

The relation $\sqsubseteq$ is used to define predicates over terms that we call constraints. To each set L of linear terms is associated a predicate over terms denoted $t \diamond L$ which is true if and only if $l \not\sqsubseteq t$ for every l in L. These constraints are said to be structural as they are specific to the term structure only as opposed to arbitrary predicates.

Definition 3 (Constraint) *A constraint is recursively defined as either an atomic predicate $t \diamond L$ or the disjunction of two constraints or the conjunction of two constraints.*

$$\begin{array}{rrcl} \textit{Constraint :} & P & ::= & \textit{term} \diamond \textit{Set of linear terms} \\ & & \mid & P \vee P \\ & & \mid & P \wedge P \end{array}$$

When $(t \diamond L)$ we say that L is a constraint over t.

The truth value of compound constraints is obtained by standard interpretation of the logical connectives. We write $\models P$ if and only if the predicate P is true. By using the usual equivalences on connectives *or* ($\vee$) and *and* ($\wedge$), a constraint can always be written in disjunctive normal form (as a disjunction of conjunctions of atomic constraints).

Definition 4 (Substitution over a Constraint) *Let σ be a substitution, $t \diamond L$ an atomic constraint, P_1, P_2 two constraints. By definition,*

$$\sigma(t \diamond L) = \sigma(t) \diamond L, \quad \sigma(P_1 \vee P_2) = \sigma(P_1) \vee \sigma(P_2) \quad \textit{and} \quad \sigma(P_1 \wedge P_2) = \sigma(P_1) \wedge \sigma(P_2).$$

A substitution σ satisfies a constraint P if and only if $\models \sigma(P)$.

For instance, $\not\models(F(A,B)\diamond\{F(A,\Omega)\})$ and $\models F(A,B)\diamond\{F(A,\bullet)\})$.

Two constraints P and P' are said to be equivalent, denoted $P{\equiv}P'$, if and only if, the sets of substitutions satisfying P and P' are the same. A constraint P implies a constraint Q, denoted $P \Rightarrow Q$, if and only if every substitution satisfying P also satisfies Q.

Remarks: For every term t and for every substitution η, $\not\models(\eta(t)\diamond\{\Omega,\ldots\})$ and $\models(\eta(t)\diamond\{\})$. In what follows $\mathcal{F}$ and $\mathcal{T}$ will denote respectively $t\diamond\{\Omega\}$ and $t\diamond\{\}$. Notice that when $\not\models P$ then for every substitution η, $\not\models(\eta(P))$ and thus $P{\equiv}\mathcal{F}$. Also if there is a substitution that satisfies P (noted $\models P$), P is not always equivalent to $\mathcal{T}$ as there may be some substitutions that do not satisfy P.

2.3 Constraint Simplification

We can prove by induction on the structure of l that $t\diamond\{l\} \vee t\diamond\{l'\}{\equiv}t\diamond\{l\sqcup l'\}$ and that if $l \sqsubseteq l'$ then $t\diamond\{l\} \wedge t\diamond\{l'\}{\equiv}t\diamond\{l\}$. From those properties we deduce the following simplification rules that associate to each constraint an equivalent normal form where the term in each atomic constraint is a variable:

Let $t, t_1, \ldots, t_n$ be terms, $t'_1, \ldots, t'_n$ be linear terms, L be a set of linear terms and x a variable.

$$
\begin{array}{rcl}
F(t_1,\ldots,t_n)\diamond\{F(t'_1,\ldots,t'_n)\} \cup L & \equiv & (\bigvee_{1\le i\le n} t_i\diamond\{t'_i\}) \wedge F(t_1,\ldots,t_n)\diamond L \\
F()\diamond\{F()\} \cup L & \equiv & \mathcal{F} \\
F(t_1,\ldots,t_n)\diamond\{G(\ldots)\} \cup L & \equiv & F(t_1,\ldots,t_n)\diamond L \\
t\diamond\{\} & \equiv & \mathcal{T} \\
t\diamond\{\Omega\} \cup L & \equiv & \mathcal{F} \\
t\diamond\{l,\sigma(l)\} \cup L & \equiv & t\diamond\{l\} \cup L \\
t\diamond L \wedge t\diamond L' & \equiv & t\diamond L \cup L' \\
\wedge_i t\diamond\{l_i\} \vee \wedge_j t\diamond\{l'_j\} & \equiv & \wedge_{i,j} t\diamond\{l_i \sqcup l'_j\}
\end{array}
$$

These simplification rules define a function, denoted $simpl$, which transforms a constraint P into $\mathcal{F}$, $\mathcal{T}$, or an equivalent constraint over variables. Notice that these rules are different from those of disequations in [3] because we deal explicitly with the symbol $\bullet$ that represents non-evaluable terms. For example $x\diamond\{A\} \vee x\diamond\{B\} \not\equiv x\diamond\{A\} \vee A\diamond\{B\}$ because $\bullet$ does not satisfy the left part while the right part is equivalent to $\mathcal{T}$.

The *restriction* of a simplified constraint P to a given set of variables V is $simpl(P')$ where P' is the constraint obtained when replacing by $\mathcal{T}$ all of the atomic constraints of the form $x\diamond L_x$ such that $x \notin V$. For instance the constraint $(x\diamond\{F(\Omega,\Omega)\} \wedge y\diamond\{A\})$ restricted to $\{x\}$ is $(x\diamond\{F(\Omega,\Omega)\} \wedge \mathcal{T}){\equiv}x\diamond\{F(\Omega,\Omega)\}$; the restriction of $(x\diamond\{F(\Omega,\Omega)\} \vee y\diamond\{A\})$ to $\{x\}$ is $\mathcal{T}$. Notice that when $\models P$ we have $\models P'$ for any restriction P' of P.

The following lemma shows the relation between the constraints and the prefix ordering.

Lemma 2

1. *Let σ be a substitution satisfying a constraint $P = t\diamond L$. Every substitution ρ more general than σ satisfies P.*

2. *Let t and l be two terms. If $t \sqsubseteq l$ and $t \not\preceq l$ then $t\diamond\{l\} \equiv \mathcal{T}$.*

3. *Let t be a term and σ, ρ two substitutions. $t\diamond\{\sigma(t)\} \Rightarrow t\diamond\{\rho(t)\}$ if and only if $\sigma(t) \sqsubseteq \rho(t)$. More generally, $\bigwedge_i t\diamond\{\sigma_i(t)\} \Rightarrow t\diamond\{\rho(t)\}$ if and only if there exists σ_i such that $\sigma_i(t) \sqsubseteq \rho(t)$*

4. *Let t and l be two terms and L a set of terms. Either $t\diamond\{l\} \equiv \mathcal{T}$ or there exists a substitution σ such that $t\diamond\{l\} \equiv t\diamond\{\sigma(t)\}$. More generally for any predicate $t\diamond L$ there exists a possibly empty set of terms $L' = \{l_1, \ldots, l_n\}$ such that $t \preceq l_i$ and $t\diamond L \equiv t\diamond L'$.*

Proof: These properties are proved by induction on the structure of terms.

1. Let us suppose that $P = t\diamond L$. If $\Omega \in L$ or $t = \bullet$, the left part of the implication is never satisfied. The only property to prove is that for every term $t \neq \bullet$, every $l = F(l_1, \ldots, l_n)$ and substitution σ, $\not\models t\diamond\{l\}$ implies $\not\models \sigma(t)\diamond\{l\}$. The hypothesis implies $t = F(t_1, \ldots, t_n)$ and for every i $(1 \leq i \leq n)$, $\not\models t_i\diamond\{l_i\}$. By induction $\not\models \sigma(t_i)\diamond\{l_i\}$ and by definition $\not\models \sigma(t)\diamond\{l\}$. The proof easily extends to arbitrary constraints but the extension is not necessary because, as we will see below, any predicate is equivalent to one of the form $t\diamond L$.

2. If $t \sqsubseteq l$ and $t \not\preceq l$ then there exists an occurrence u of both terms such that $l/u = \bullet$ and t/u is not a variable and is different from $\bullet$. Thus for every substitution η the subterm $\eta(t)/u$ is also different from $\bullet$ and is not a variable which implies $l \not\sqsubseteq \eta(t)$.

3. If $t\diamond\{\sigma(t)\} \Rightarrow t\diamond\{\rho(t)\}$, ρ does not satisfy $t\diamond\{\sigma(t)\}$ because it does not satisfy $t\diamond\{\rho(t)\}$ and thus $\sigma(t) \sqsubseteq \rho(t)$. Conversely, for every substitution η satisfying $t\diamond\{\sigma(t)\}$, $\sigma(t) \not\sqsubseteq \eta(t)$. As $\sigma(t) \sqsubseteq \rho(t)$, $\rho(t) \not\sqsubseteq \eta(t)$ and we conclude that η satisfies $t\diamond\{\rho(t)\}$. The generalization is made by simple manipulation of logical connectives.

4. We remark that $t\diamond\{l\} \equiv t\diamond\{l\} \vee t\diamond\{t\} \equiv t\diamond\{l \sqcup t\}$. As $t \sqsubseteq l \sqcup t$, by part (2) either $t\diamond\{l \sqcup t\} \equiv \mathcal{T}$ or $t \preceq l \sqcup t$ that proves the property. The generalization is made by simple manipulation of logical connectives.

2.4 Constrained Terms

Definition 5 (Constrained Term) *Let t be a term and P a constraint. A constrained term $\{t|P\}$ is the set of ground instances of t satisfying the restriction P' of P to the variables of t.*

$$\textit{constrained terms :} \quad T \quad ::= \quad \{t|P\}$$

In what follows we will call t the *pure* part of T and substitutions over pure terms will be called *pure substitutions*. The set of *occurrences* of T is that of its pure part. The *subterms* of $\{t|P\}$ are of the form $\{t'|P\}$ where t' is a (pure) subterm of t. Notice that by definition, the constraint part of a subterm is restricted to the variables occurring in its pure part. When $\not\models P$ the term $\{t|P\}$ represents the empty set of terms that we note $\emptyset$. The following properties on the sets represented by constrained terms are easy to check:

1. When $P \equiv Q$, the terms $\{t|P\}$ and $\{t|Q\}$ are the same.
2. $\{t|P \vee Q\} = \{t|P\} \cup \{t|Q\}$
3. $\{t|P \wedge Q\} = \{t|P\} \cap \{t|Q\}$

Any constraint P is equivalent to its disjunctive normal form $\bigvee_i P_i$ where each P_i is a conjunction of atomic constraints over variables and thus $\{t|P\} = \bigcup_i \{t|P_i\}$. This gives a practical representation of constrained terms which is very close to their implementation.

Example: Let $T = \{F(x,y)|P\}$ where $P = F(x,y)\diamond\{F(A,B)\} \wedge y\diamond\{C\} \wedge z\diamond\{A\}$. As the variable z does not appear in T, the restriction of P is $F(x,y)\diamond\{F(A,B)\} \wedge y\diamond\{C\}$ and $T = T_1 \cup T_2$ where $T_1 = \{F(x,y)|x\diamond\{A\} \wedge y\diamond\{C\}\}$ and $T_2 = \{F(x,y)|y\diamond\{B,C\}\}$. Notice that in general the terms T_i are not disjoint as in this example where the term $F(B,A)$ belongs to both T_1 and T_2.

Even in the case of an infinite alphabet, term representation with constraints can be finite which is not the case with the classical representation.

2.5 Substitution

Definition 6 *Let σ be a substitution and Q a constraint. The constrained substitution $\overline{\sigma} = (\sigma, Q)$ is the mapping over constrained terms defined by $\overline{\sigma}(\{t|P\}) = \{\sigma(t)|\sigma(P) \wedge Q\}$. When $\models \sigma(P) \wedge Q$, $\overline{\sigma}$ is admissible for $\{t|P\}$.*

Notice that when $\not\models Q$ the substitution $\overline{\sigma} = (\sigma, Q)$ maps every term to the term $\emptyset$.

We compose two constrained substitutions $\overline{\sigma_1} = (\sigma_1, Q_1)$ and $\overline{\sigma_2} = (\sigma_2, Q_2)$ as usual and check easily that $\overline{\sigma_2} \circ \overline{\sigma_1} = (\sigma_2 \circ \sigma_1, simpl(Q_2 \wedge \sigma_2(Q_1)))$.

The definition of quasi-ordering ($\preceq$) is extended to constrained terms using constrained substitutions instead of pure substitutions. The least upper bound ($\sqcup$) is defined with respect to $\preceq$. The notion of unifier of two terms T and T' is also extended but the unifier has to be admissible for both terms to avoid the empty term as the only common instance of T and T'. The *equality* (=) of two terms T and T' is defined by: $T_1 = T_2$ if and only if $T_1 \preceq T_2$ and $T_2 \preceq T_1$. We give now a characterization of these concepts in order to compute separately the pure part and the constraint.

Lemma 3 *Let $T_1 = \{t_1|P_1\}$ and $T_2 = \{t_2|P_2\}$ be two constrained terms.*

1. *$T_1 \preceq T_2$ if and only if there exists a substitution σ such that $\sigma(t_1) = t_2$ and $P_2 \Rightarrow \sigma(P_1)$.*

2. *$\overline{\sigma} = (\sigma, Q)$ unifies T_1 and T_2 if and only if $\overline{\sigma}$ is admissible for T_1 and T_2, $\sigma(t_1) = \sigma(t_2)$ and $\sigma(P_1) \wedge Q \equiv \sigma(P_2) \wedge Q$. As usual, we say that two terms T_1 and T_2 are unifiable or compatible, denoted $T_1 \uparrow T_2$, if and only if there exists a unifier for them.*

3. *$T_1 \sqcup T_2 = \{t_1 \sqcup t_2 | \sigma(P_1 \wedge P_2)\}$ where σ is the most general unifier of t_1 and t_2. $\sqcup$ is not defined when t_1 and t_2 are not unifiable or when $\not\models \sigma(P_1 \wedge P_2)$. The substitution $\overline{\sigma} = (\sigma, \sigma(P_1 \wedge P_2))$ is a principal unifier for T_1 and T_2.*

4. *Let $T = \{t | t \diamond \{l\}\}$ and $M = \{m | \mathcal{T}\}$ be two constrained terms (in fact the last one is a pure term). $T \preceq M$ if and only if $t \preceq m$ and $m \not\uparrow l$.*

Proof:

1. $T_1 \preceq T_2$ if and only if there exists $\overline{\sigma} = (\sigma, Q)$ such that $\sigma(t_1) = t_2$ and $P_2 \equiv \sigma(P_1) \wedge Q$. This implies $P_2 \Rightarrow \sigma(P_1)$. Conversely, if there exists σ such that $\sigma(t_1) = t_2$ and $P_2 \equiv \sigma(P_1)$, as $P_2 \equiv \sigma(P_1) \wedge P_2$, $\overline{\sigma} = (\sigma, P_2)$ satisfies $\overline{\sigma}(T_1) = T_2$.

2. This is a simple consequence of the definition of equality.

3. Obviously $\{t_1 \sqcup t_2 | \sigma(P_1 \wedge P_2)\}$ is an upper bound of T_1 and T_2. Now, let $T = \{t | P\}$ be an upper bound of T_1 and T_2. By definition of the least upper bound of pure terms, there exists a substitution ρ such that $\rho(t_1 \sqcup t_2) = t$; as T is an upper bound of T_1 and T_2, there exist σ_1 and σ_2 such that $\sigma_1(t_1) = \sigma_2(t_2) = t$, $P \Rightarrow \sigma_1(P_1)$ and $P \Rightarrow \sigma_2(P_2)$. Consequently $\rho(\sigma(t_1)) = \sigma_1(t_1)$ and $\rho(\sigma(t_2)) = \sigma_2(t_2)$. These equalities hold on the variables of t_1 and of t_2 that, when applied to the constraints, give $\sigma_1(P_1) = \rho(\sigma(P_1))$ and $\sigma_2(P_2) = \rho(\sigma(P_2))$. In conclusion $P \Rightarrow \rho(\sigma(P_1 \wedge P_2))$.

4. Let σ be the substitution such that $\sigma(t) = m$. As M is a pure term, for every substitution η, $l \not\sqsubseteq \eta \circ \sigma(t) = \eta(m)$. Thus $l \not\preceq \eta(m)$ and, by definition, for every substitution ρ, $\rho(l) \neq \eta(m)$; that means, l and m are incompatible. This result is easily generalized to a set of constraints, $T = \{t | t \diamond \{l_1, \ldots, l_n\}\}$. In that case, $T \preceq M$ if and only if $t \preceq m$ and, for every i $(1 \leq i \leq n)$, $m \not\uparrow l_i$.

Example: The most general unifier $\overline{\sigma}$ of the terms:

$$\begin{array}{rcl} T & = & \{G(x, y, z) | (x \diamond \{H(u)\},\ y \diamond \{C\},\ z \diamond \{H(H(B)\})\} \\ T' & = & \{G(P(A), y', H(z')) | (y' \diamond \{B\},\ z' \diamond \{C\})\} \end{array}$$

is defined by:

$$\begin{array}{rcl} \sigma(x) & = & P(A) \\ \sigma(y) & = & y' \\ \sigma(z) & = & H(z') \end{array} \qquad \overline{\sigma} = (\sigma, (y' \diamond \{B, C\},\ z' \diamond \{H(B), C\}))$$

In general, the greatest lower bound of two terms does not exist. For instance, the common prefixes of $\{A|\mathcal{T}\}$ and $\{B|\mathcal{T}\}$ (with $A \neq B$) are of the form $\{x|x\diamond L\}$, but for each prefix there is a constant C not belonging to L and different from A and B such that $\{x|x\diamond L \cup \{C\}\}$ is a prefix of both terms less general than $\{x|x\diamond L\}$. We give now a sufficient condition for the existence of the greatest lower bound:

Lemma 4 *Let $T_1 = \{t_1|P_1\}$ and $T_2 = \{t_2|P_2\}$ be two compatible constrained terms. Let $\sigma_i(t_1 \sqcap t_2) = t_i$ and σ'_i the substitutions defined by $\sigma'_i(x') = x$ if $\sigma_i(x) = x'$ and $\sigma'_i(x') = x'$ otherwise. The greatest lower bound of T_1 and T_2 exists and is the term $T = \{t_1 \sqcap t_2|\sigma'_1(P_1) \vee \sigma'_2(P_2)\}$.*

Proof: Remember that $\{t_1 \sqcap t_2|\sigma'_1(P_1) \vee \sigma'_2(P_2)\} = \{t_1 \sqcap t_2|Q_1 \vee Q_2\}$ where each Q_i is the restriction of $\sigma'_i(P_i)$ to the variables of $t_1 \sqcap t_2$ and thus $\sigma_i(Q_i) = Q_i$. As each P_i implies Q_i, T is a prefix of each T_i. Let T' be a prefix of both T_1 and T_2 greater than T. Thus $T' = \{t_1 \sqcap t_2|P'\}$ such that $P' \Rightarrow Q_1 \vee Q_2$ and $P_i \Rightarrow \sigma_i(P')$. Consequently, $(Q_i =)\ \sigma'_i(P_i) \Rightarrow \sigma'_i \circ \sigma_i(P')\ (= P')$, and thus $P' \equiv Q_1 \vee Q_2$.

Example: The terms $T_1 = \{F(x,y)|x\diamond\{A\}, y\diamond\{B\}\}$ and $T_2 = \{F(B,y)|y\diamond\{A,C\}\}$ are compatible and $T_1 \sqcap T_2 = \{F(x,y)|x\diamond\{A\}, y\diamond\{\bullet\}\}$. Incompatible terms may also have a greatest lower bound, for instance $\{A|\mathcal{T}\} \sqcap \{x|x\diamond\{A\}\} = \{x|x\diamond\{\bullet\}\}$.

Definition 7 (Restriction by a Substitution) *Let $T = \{t|P\}$ be a constrained term and σ a pure substitution. The restriction of T by σ is the constrained term:*

$$T|_\sigma = \{t|simpl(P \wedge t\diamond\{\sigma(t)\})\}$$

Notice that restriction is defined only for a pure substitution because there is no constrained term in a constraint.

Example: For the term $T = \{F(x,y)|x\diamond\{H(A)\}\}$ and the substitution σ defined by $\sigma(x) = H(z)$, $\sigma(y) = A$, we obtain:

$$\begin{array}{rcl} \sigma(T) & = & \{F(H(z),A)|z\diamond\{A\}\} \text{ and} \\ T|_\sigma & = & \{F(x,y)|F(x,y)\diamond\{F(H(\Omega),A)\} \wedge x\diamond\{H(A)\}\} \\ & \equiv & \{F(x,y)|x\diamond\{H(z),H(A)\}\} \cup \{F(x,y)|(x\diamond\{H(A)\},\ y\diamond\{A\})\} \end{array}$$

Notice that in general the terms of $T|_\sigma$ may have common instances like the term $F(A,B)$ in this example.

To each substitution σ we also associate the set $\dot{\sigma}$ of substitutions corresponding to non-evaluated parts of σ.

Definition 8 *Let σ be a substitution. $\dot{\sigma}$ is the set of substitutions $\dot{\sigma}$ defined by $\dot{\sigma}(x)$ is $\sigma(x)$ in which some non-variable subtrees different from $\bullet$ have been replaced by $\bullet$.*

A substitution σ is considered as the semantics of an operation that defines more precisely a partial term T and the terms $\dot{\sigma}(T)$ represent those instances that failed to be evaluated. This is why we call the set $\{\dot{\sigma}(T) \mid \dot{\sigma} \in \dot{\mathcal{O}}\}$, the *strict* part of T with respect to σ. The set $\sigma(T) \cup T|_\sigma$ is the *calculable* part of T with respect to σ.

Lemma 5 *Let σ be a pure substitution, Id the identity substitution and $T = \{t|P\}$ a constrained term. $T|_{Id} = \emptyset$ and $\sigma(T) \cap T|_\sigma = \emptyset$. The set of instances of T is the union of instances of $\sigma(T)$, $T|_\sigma$ and $\dot{\sigma}(T)$.*

Proof: By definition if $\eta(t)$ is an element of $\sigma(T)$ then $\sigma(t) \preceq \eta(t)$ and thus $\not\models \eta(t) \diamond \{\sigma(t)\}$. As a consequence, the two sets are disjoints and $T|_{Id}$ is empty. The last property is proved by cases on the definition of $\sqsubseteq$.

The proposition below will be useful in the definition of the decomposition of a term.

Proposition 1 *For every term T and substitution σ, T is the greatest lower bound of $\sigma(T)$, $T|_\sigma$ and all the $\dot{\sigma}(T)$.*

Proof: Let $T = \{t|t \diamond \{L\}\}$. It is easy to prove that T is a prefix of $\sigma(T)$, $T|_\sigma$ and all the $\dot{\sigma}(T)$. Furthermore, if there exists a common prefix T_0 of these terms that is not a prefix of T, $T \sqcup T_0$ is also a common prefix. Now let $T' = \{t'|P'\}$ be a lower bound of $\sigma(T)$, $T|_\sigma$ and all the $\dot{\sigma}(T)$ such that $T \preceq T'$. By hypothesis on the pure terms, $t \preceq t' \preceq t$ and thus $t = t'$. The hypothesis on the constraints are:

$$\begin{aligned} t \diamond L \wedge t \diamond \{\sigma(t)\} &\Rightarrow P' \Rightarrow t \diamond L \\ \sigma(t) \diamond L &\Rightarrow \sigma(P') \\ \dot{\sigma}(t) \diamond L &\Rightarrow \dot{\sigma}(P') \end{aligned}$$

The first property implies $P' \equiv t \diamond L \bigwedge P''$ and $t \diamond \{\sigma(t)\} \Rightarrow P''$. By lemma 2 either $P'' \equiv \mathcal{T}$ which implies $P' \equiv P$ or $P'' \equiv \bigwedge_i t \diamond \{\rho_i(t)\}$. As $t \diamond \{\sigma(t)\} \Rightarrow \bigwedge_i t \diamond \{\rho_i(t)\}$, by lemma 2 again, for each i, $\sigma(t) \sqsubseteq \rho_i(t)$. Consequently the following implications are satisfied:

$$t \diamond L \wedge t \diamond \{\sigma(t)\} \Rightarrow t \diamond L \wedge \bigwedge_i t \diamond \{\rho_i(t)\} \tag{1}$$

$$\sigma(t) \diamond L \Rightarrow \sigma(t) \diamond L \bigwedge_i \sigma(t) \diamond \{\rho_i(t)\} \tag{2}$$

$$\dot{\sigma}(t) \diamond L \Rightarrow \dot{\sigma}(t) \diamond L \bigwedge_i \dot{\sigma}(t) \diamond \{\rho_i(t)\} \tag{3}$$

Either $\sigma(t) \leq \rho_i(t)$ or there exists t'' such that $\sigma(t) \sqsubseteq_0 t'' \leq \rho_i(t)$. In the second case, no $\bullet$ is inserted inside t otherwise t'' could not be a prefix of $\rho_i(t)$, and $t'' = \dot{\sigma}(t)$. In both cases, as a consequence of lemma 2, (2) and (3) respectively imply $t \diamond L \Rightarrow t \diamond \{\rho_i(t)\}$ and thus $P \Rightarrow P'$ that means $T = T'$.

3 Term Decomposition

We want to partition, following an ordered list of linear terms $S = (s_1, \ldots, s_n)$ named *patterns*, the set of all terms represented by T into a set of disjoint ones. The decomposition of T with respect to S consists to split the set associated to T into subsets such that each subset contains instances of at most one element of S. For example, let $S = \{F(A, H(\Omega)), F(A, \Omega)\}$ and $T = \{F(x, y)|\mathcal{T}\}$. The evaluable part of T is the disjoint union of T_1, T_2 and T_3 where $T_1 = \{F(A, H(x))|\mathcal{T}\}$, $T_2 = \{F(A, y)|y\diamond\{H(\Omega)\}\}$ and $T_3 = \{F(x, y)|x\diamond\{A\}\}$. Constrained terms and decomposition were introduced in [8] to deal with recursive path orderings with unavoidable sets.

3.1 Decomposition

Definition 9 (Decomposition w.r.t. a Pattern) *Let T be a constrained term, and s a pattern. If T and s are unifiable with σ as their most general unifier, the decomposition of T w.r.t. s, denoted compat(T, s), is equal to $\sigma(T)$.*

With this definition, compat(T, Ω) and T represent the same set of terms.

Definition 10 (Decomposition w.r.t. an Ordered Set of Patterns) *Let T be a constrained term and $S = \{s_1, \ldots, s_n\}$ an ordered set of patterns. The decomposition of T w.r.t. S, Decomp(T, S), is recursively defined as:*

$$\begin{array}{lcl} \mathit{Decomp}(\emptyset, S) & = & \emptyset \\ \mathit{Decomp}(T, \emptyset) & = & \emptyset \\ \mathit{Decomp}(T, S) & = & \mathit{Decomp}(T, \{s_2, \ldots, s_n\}) \text{ if } T \text{ and } s_1 \text{ are incompatible} \\ & = & \{\mathit{compat}(T, s_1)\} \cup \mathit{Decomp}(T|_{\sigma_1}, S) \\ & & \text{where } \sigma_1 \text{ is the m.g.u. of } T \text{ and } s_1 \text{ otherwise.} \end{array}$$

Notice that Decomp stops when a pattern is already a factor of T because the restriction of a term by the identical substitution is the emptyset. The instances of T that do not belong to Decomp$(T, S \cup \{\Omega\})$ are those for which there is no way to decide if they are instances of one of the elements of S.

Proposition 2 *Let T be a constrained term, u an occurrence of T and $S = \{s_1, \ldots, s_n\}$ a set of patterns. Then T is the greatest lower bound of Decomp$(T, \{s_1, \ldots, s_n, \Omega\}) \cup \bigcup_{1\leq i\leq n}\{\dot{\sigma}_i(T) \mid \dot{\sigma}_i \in \dot{\sigma}_i\}$*

Proof: This property is a consequence of the definition of Decomp and of Proposition 1. Notice that, in the decomposition of a term, S is used as an ordered list and thus Ω is the last element of this list. This decomposition is a partition of the evaluable part of T.

3.2 Decomposition Procedure

Let $T = \{t|P\}$ be a constrained term and $S = \{s_1, \ldots, s_n\}$ a set of patterns.

Initialization step
$\theta_1 \leftarrow T$; $S \leftarrow S$
Current step
$(\tau_i, \theta_{i+1}) \leftarrow (\sigma_i(\theta_i), \theta_i|_{\sigma_i})$ if θ_i and s_i are unifiable with m.g.u. σ_i
$\leftarrow (\emptyset, \theta_i)$ if not.

Then Decomp$(T, S) = \{\tau_1, \ldots, \tau_n\}$.

The following lemmas will be used for the pattern matching:

Lemma 6 *Let $S = \{s_1, \ldots, s_n\}$ be a set of patterns and $\{\tau_1, \ldots, \tau_n\}$ the decomposition of a variable x by the set S. For every i, $\tau_i = \{s_i| \bigwedge_{j<i} s_i \diamond \{s_j\}\}$ and for every i and $j \neq i$, $\tau_i \cap \tau_j = \emptyset$.*

Proof: The first part is proved by induction over i: As $\theta_1 = x$, $\tau_1 = \{s_1|\mathcal{T}\}$ and $\theta_2 = \{x|x\diamond\{s_1\}\}$. Now suppose that $\tau_i = \{s_i| \bigwedge_{j<i} s_i\diamond\{s_j\}\}$ and $\theta_{i+1} = \{x| \bigwedge_{j\leq i} x\diamond\{s_j\}\}$. Then $\tau_{i+1} = \{s_{i+1}| \bigwedge_{j\leq i} s_{i+1}\diamond\{s_j\}\}$ and $\theta_{i+2} = \{x| \bigwedge_{j\leq i} x\diamond\{s_j\} \wedge x\diamond\{s_{i+1}\}\}$, that finishes the proof. The second part is a direct consequence of Lemma 5.

Example: When we take the set of patterns:

$$F(x, B),\ F(P(y), z),\ F(t, u),\ F(H(v), w)$$

The decomposition of the term $T = F(x, y)$ gives the following set of constrained terms (in the examples we write $x\diamond\{F\}$ instead of $x\diamond\{F(\Omega, \ldots, \Omega)\}$):

$$F(x, B),\ \{F(P(y), z)|z\diamond\{B\}\}\ ,\ \{F(t, u)|u\diamond\{B\} \wedge t\diamond\{P\}\}$$

If we decompose the term $T = x$ the result is:

$$F(x, B),\ \{F(P(y), z)|z\diamond\{B\}\}\ ,\ \{F(t, u)|t\diamond\{P\} \wedge u\diamond\{B\}\}\ ,\ \{v|v\diamond\{F\}\}$$

The strict set of x with respect to the patterns is:

$$\bullet\ ,\ F(x, \bullet)\ ,\ \{F(\bullet, y)|y\diamond\{B\}\}$$

Notice that in this example redundant patterns disappear. As we decompose a variable, the new set of constrained patterns and the strict set of x represent the set of all the terms.

The following lemma allows us use the decomposition of a variable to compute the decomposition of any term. Afterwards, the decomposition of a term is a unification with these new patterns as was illustrated with the previous example.

Lemma 7 *Let $S = \{s_1, \ldots, s_n\}$ be a set of patterns, $\{s'_1, \ldots, s'_n\}$ = Decomp(x, S) and $T = \{t|P\}$ a constrained term. Then Decomp(T, S) = $\{\overline{\sigma_i}(s'_i) \mid 1 \leq i \leq n\}$ where, for every i, $\overline{\sigma_i}$ is the most general unifier of s'_i and T.*

Proof: Let σ_i be the most general unifier of t and s_i for every i ($1 \leq i \leq n$). By definition Decomp(T, S) = $\{\{\sigma_i(t)|\sigma_i(P) \wedge (\bigwedge_{1\leq j\leq i-1} \sigma_i(t)\diamond\{\sigma_j(s_j)\})\} \mid 1 \leq i \leq n\}$. As $\{\overline{\sigma_i}(s'_i) \mid 1 \leq i \leq n\}$ = $\{\{\sigma_i(t)|\sigma_i(P) \wedge (\bigwedge_{1\leq j\leq i-1} \sigma_i(s_i)\diamond\{s_j\})\} \mid 1 \leq i \leq n\}$, in order to prove the lemma, it is sufficient to prove the equivalence of the constraints $\sigma_i(t)\diamond\{\sigma_j(s_j)\}$ and $\sigma_i(s_i)\diamond\{s_j\}$ for every integers i,j such that $j < i$. $\sigma_i(t)\diamond\{\sigma_j(s_j)\}\equiv\sigma_i(s_i)\diamond\{s_j\}$ if and only if for all substitution η, $\sigma_j(s_j) \not\sqsubseteq \eta \circ \sigma_i(t)$ if and only if $s_j \not\sqsubseteq \eta \circ \sigma_i(s_i)$. The if part is clear. Let us suppose that there exists η such that $\sigma_j(s_j) \not\sqsubseteq \eta \circ \sigma_i(t)$ and $s_j \sqsubseteq \eta \circ \sigma_i(s_i)$ = $\eta \circ \sigma_i(t)$. As s_j and t are unifiable with the m.g.u. σ_j, if $s_j \preceq \eta \circ \sigma_i(t)$ there exists a substitution ρ_j such that $\rho_j \circ \sigma_j(s_j) = \eta \circ \sigma_i(t)$. Thus $\sigma_j(s_j) \sqsubseteq \eta \circ \sigma_i(t)$ that is a contradiction. Otherwise, let $U = \{u\in\mathcal{O}(s_j) \mid s_j/u\neq\bullet$ and $\eta \circ \sigma_i(t)/u = \bullet\}$. Notice that $U \cap \mathcal{O}(t) = \emptyset$ because t and s_j are unifiable. Thus $s_j \preceq \eta \circ \sigma_i(t)[u \leftarrow s_j/u \mid u\in U]$ which is an instance of t. Then, we conclude as in the previous case.

3.3 Decomposition Normalization

It is useful to transform constraints into an easily readable shape and we propose now a normalization algorithm. Its first step is to split these terms into terms with constraints with only one function symbol. Its second step is to normalize the constraint associated to a variable appearing at the same occurrence in two trees in the decomposition, in order to get the same constraint at common occurrences.

Definition 11 *A decomposition $T = \{T_1, \ldots, T_n\}$ is in normal form if and only if:*

1. *Each constraint occurring in each T_i, has only one symbol.*
2. *If there exist $i\neq j$ and an occurrence u of T_i and T_j such that, for every $u' <_{prefix} u$, T_i and T_j have the same symbol at u', $T_i/u = \{x|x\diamond L_x\}$, $T_j/u = \{y|y\diamond L_y\}$ then, $L_x = L_y$.*

Normalization Algorithm Let $\{T_1, \ldots, T_n\}$ be a set of constrained terms.

let $T = \{t|(x\diamond\{C(t'_1, \ldots, t'_n)\} \cup L) \wedge P\}$. If there exists t'_i non-variable:

$$T \Rightarrow \{t|(x\diamond\{C(\Omega, \ldots, \Omega)\} \cup L) \wedge P\} \cup T[x \leftarrow C(x_1, \ldots, x_n)]$$

If there exist T_i and T_j satisfying hypothesis 2. above with $C(\Omega, \ldots, \Omega)\in L_y - L_x$ and $L_x\neq\emptyset$:

$$T_i \Rightarrow T_i[u \leftarrow \{x|x\diamond\{C(\Omega, \ldots, \Omega)\} \cup L_x\}] \cup T_i[u \leftarrow \{C(x_1, \ldots, x_n)|\mathcal{T}\}]$$

The normalization algorithm does not change the strict set of T, as it only makes substitutions to constrained variables.

Example: The decomposition of the following set of patterns $F(G(A), B)$, $F(y, B)$, $F(C, z)$, x gives as result:

$$F(G(A), B) \quad \{F(y, B)|y\diamond\{G(A)\}\} \quad \{F(C, z)|z\diamond\{B\}\} \quad \{x|x\diamond\{F(\Omega, B), F(C, \Omega)\}\}$$

We notice that the constraints over x and y have to be normalized. The first step of normalization transforms the patterns $\{F(y, B)|y\diamond\{G(A)\}\}$ and $\{x|x\diamond\{F(\Omega, B), F(C, \Omega)\}\}$ into these new patterns:

$$\begin{array}{ll} \{F(y, B)|y\diamond\{G\}\} & \{F(G(t), B)|t\diamond\{A\}\} \\ \{x|x\diamond\{F\}\} & \{F(y, z)|y\diamond\{C\}\ z\diamond\{B\}\} \end{array}$$

The second step gives the resulting set:

$$\begin{array}{lll} F(G(A), B) & \{F(G(t), B)|t\diamond\{A\}\} & F(C, B) \\ \{F(y, B)|y\diamond\{G, C\}\} & \{F(C, z)|z\diamond\{B\}\} & \{F(G(t), z)|z\diamond\{B\}\} \\ \{F(y, z)|y\diamond\{G, C\}\ z\diamond\{B\}\} & \{x|x\diamond\{F\}\} & \end{array}$$

4 Pattern Matching

In this section we use constrained terms to reason about pattern matching over pure terms.

Definition 12 *A set of patterns* Π *is complete for a term* N *if every ground instance of* N *is also an instance of an* $M \in \Pi$.

Let $\Pi = \{M_1, \ldots, M_n\}$ be a set of patterns. The simplest matching predicate over Π is defined by $match_{\Pi}(t) = True$ if and only if there exists $M_i \in \Pi$ such that $M_i \preceq t$ where t is a pure term. This predicate does not take account of any priority over Π and is not suitable for pattern matching over partially evaluated terms. A. Laville in [6] defines a matching predicate over pure terms which takes care of the ordering.

Definition 13 *Let* $\Pi = \{M_1, \ldots, M_n\}$ *be a set of patterns ordered by priority, and* t *be a pure term.* $Match_{\Pi}(t) = True$ *if and only if there exists* $i (1 \leq i \leq n)$ *such that* $M_i \preceq t$ *and for every* $j < i$, $t \not\uparrow M_j$.

The priority on patterns is necessary to force the matching with a chosen pattern when several patterns are compatible.

With the concept of constrained terms, we replace priority by constraints. We transform the ordered set of patterns into an unordered set of constrained ones using the decomposition

algorithm and without loosing generality, we work on the evaluable part of terms with respect to the set of patterns.
Let $\Pi = \{M_1, \ldots, M_n, \Omega\}$ be an ordered set of patterns and x a variable. The decomposition algorithm computes $\Pi' = \text{Decomp}(x, \Pi)$ which is the set of constrained patterns $M'_i = \{M_i | \bigwedge_{j<i} M_i \diamond \{M_j\}\}$. Remember that $M'_i \cap M'_j = \emptyset$ for $i \neq j$ and that the redundant patterns, represented by empty constrained terms, are eliminated.

Definition 14 (Pattern Matching) *Let $\Pi = \{M_1, \ldots, M_n\}$ be a set of disjoint constrained patterns, and T be a constrained term. RMatch$_\Pi(T) = True$ if and only if there exists i $(1 \leq i \leq n)$ such that $M_i \preceq T$.*

Notice that the relation $\preceq$ over constrained terms is transitive and the predicate Rmatch$_\Pi$ is monotonic with the ordering $False < True$.

In the following theorem, we prove that the predicate RMatch$_{\Pi'}$, which only uses the prefix ordering, is as powerfull as Match$_\Pi$ which uses the prefix ordering and incompatibility tests.

Theorem 1 *Let $\Pi = \{m_1, \ldots, m_n\}$ be an ordered set of patterns and Π' the decomposition of x by Π. Π' is the set of minimal generators of the terms satisfying the predicate RMatch$_{\Pi'}$ and for every pure term t:*

$$\text{Match}_\Pi(t) \equiv \text{RMatch}_{\Pi'}(\{t|\mathcal{T}\})$$

Proof: By definition RMatch$_{\Pi'}(T) = True$ if and only if there exists $M' \in \Pi'$ such that $M' \preceq T$, that means T is generated by M'. Conversely each non-empty $M' \in \Pi'$ generates its ground instances. Furthermore, as the elements of Π' are incompatible they are minimal generators. Let $\Pi' = \{M'_1, \ldots, M'_n\}$. By definition and lemma 3, RMatch$_{\Pi'}(\{t|\mathcal{T}\}) = True$ if and only if there exists $M'_i \in \Pi'$ such that $m_i \preceq t$ and for every $j < i$, $t \not\uparrow m_j$. We recognize there the definition of the predicate Match$_\Pi$ over pure terms.

Notice that Π' generates the set of pure terms satisfying Match$_\Pi$ and gives a set of minimal generators more compact than the minimal set of generators described in [6], page 44. For instance, the decomposition of the patterns $F(A, B, z)$, $F(A, A, z)$, $F(x, y, C)$ and $F(x, y, z)$ is the set:

$$\begin{array}{ll} F(A, B, z), & \{F(x, y, C)|F(x, y, C)\diamond\{F(A, B, \Omega), F(A, A, \Omega)\}\}, \\ F(A, A, z), & \{F(x, y, z)|F(x, y, z)\diamond\{F(A, B, \Omega), F(A, A, \Omega)\}\} \end{array}$$

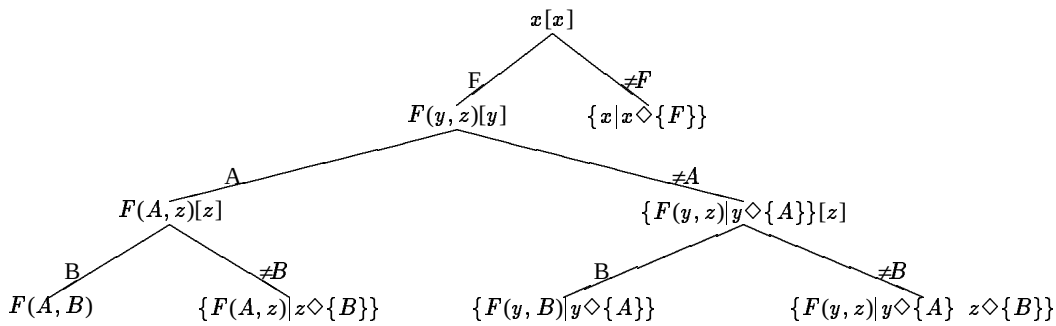
Normalization gives the following set:

$$\begin{array}{lll} F(A, B, z), & \{F(x, y, C)|x\diamond\{A\}\}, & \{F(x, y, z)|x\diamond\{A\}, z\diamond\{C\}\}, \\ F(A, A, z), & \{F(x, y, C)|y\diamond\{A, B\}\}, & \{F(x, y, z)|y\diamond\{A, B\}, z\diamond\{C\}\} \end{array}$$

There are several algorithms to check the match of a term by a given set of patterns. We will use *Search Trees* to represent these algorithms. These trees have as labels, pairs of a

constrained term and an occurrence of a variable in it. The label of the root is a variable and on each branch the labels are terms more and more instanced. The sons of a term with label T, u have as term $T[u \leftarrow T']$ where T' contains at most one function symbol and is a prefix of a pattern compatible with T. The leaves of the tree are compatible with exactly one pattern (the occurrence is of no use). The only freedom in the construction is the choice of the occurrence used to develop the subtrees.

For instance, if the choice of the occurrence is always the leftmost variable that leads to the pattern having priority, as it is the choice of many compilers for functional languages, the search tree associated to the patterns $F(A, B)$, $F(y, B)$ and x is:



The strict set of the match is $\bullet$, $F(y, \bullet)$ and $F(\bullet, B)$. This algorithm will not give a result for the term $F(\bullet, A)$, which does not belong to the strict set of the match.

Definition 15 *A pattern matching algorithm is optimal if and only if it fails to produce a result only on the strict set of the match.*

In the following section we give a characterization for the optimality of the pattern matching algorithm.

4.1 Sequentiality

We say that a pattern matching problem is sequential when it can be computed without looking ahead on a sequential machine. In this section we describe how to decide if a match problem is sequential and in such case, how to build the search tree associated to it. This section adapts the definitions and proofs of [5] to the case of constrained terms.

Definition 16 (Index, Sequential)
Let $\mathcal{P}$ be a monotonic predicate on constrained terms (with the truth values domain ordered as $False < True$).

- *An occurrence u of T is said to be an $index$ of $\mathcal{P}$ in T if and only if*
  1. $T/u = \{\Omega|\mathcal{T}\}$

2. *For every* $M \succeq T$, $\mathcal{P}(M) = True$ *implies* $(M/u) \not\preceq (T/u)$ *(i.e.* $(M/u) \neq \{\Omega|\mathcal{T}\}$*).*

- *Then* $\mathcal{P}$ *is* $sequential$ *at* T *if and only if whenever* $\mathcal{P}(T) = False$ *and there exists* $M \succeq T$ *such that* $\mathcal{P}(M) = True$*, it follows that there exists an index of* $\mathcal{P}$ *in* T*.*
- *Finally* $\mathcal{P}$ *is said to be sequential if and only if it is sequential at every calculable constrained term.*

As the predicate Rmatch_Π is monotonic, we look for its sequentiality at every term, called the sequentiality of Π. The set $\text{Dir}_\Pi(T)$ of the indexes of Rmatch_Π in T is the set of directions from T to Π.

Lemma 8 *Let* T *be a constrained term and* Π *a set of disjoint constrained patterns.* $u \in Dir_\Pi(T)$ *if and only if* $T/u = \{\Omega|\mathcal{T}\}$ *and, for all* $M \in \Pi$ *such that* $M \uparrow T$*, one has* $u \in \mathcal{O}(M)$ *and* $M/u \neq \{\Omega|\mathcal{T}\}$*.*

Proof: Let $u \in \text{Dir}_\Pi(T)$ and $M \in \Pi$ such that $M \uparrow T$. Thus, $T/u = \{\Omega|\mathcal{T}\}$ and there exists T' such that $T \preceq T'$ and $M \preceq T'$. Suppose that $u \notin \mathcal{O}(M)$. There exists a proper prefix u' of u such that $u = u'w$ with $w \neq \epsilon$ and $M/u' = \{\Omega|\Omega\Diamond L_\Omega\}$. As $M \preceq T'$, the subterm T'/u' satisfies the constraints and $T'/u'[w \leftarrow \{\Omega|\mathcal{T}\}]$ also. Therefore $M \preceq T'[u \leftarrow \{\Omega|\mathcal{T}\}]$, which contradicts the second condition of the definition of a direction and also our hypothesis. Knowing that $u \in \mathcal{O}(M)$, obviously $M/u \not\preceq T/u$. Conversely, if there is a term $T' \succeq T$ such that $\text{Rmatch}_\Pi(T') = True$, there is a pattern $M \in \Pi$ compatible with T. Thus $M/u \neq \{\Omega|\mathcal{T}\}$ that implies $T'/u \not\preceq T/u$ and the equivalence is clear.

Remark: This lemma gives a simple characterization of directions. By normalization, a pattern $M \in \Pi$ is split in several terms $M_1, \ldots, M_n$ which may be compatible. As a consequence of the simplification rules, $M/u \neq \{\Omega|\mathcal{T}\}$ if and only if each $M_i/u \neq \{\Omega|\mathcal{T}\}$ and thus the set of directions $\text{Dir}_\Pi(T)$ is the set of directions from T to the normalization of Π.

Lemma 9 *Let* Π *be a set of disjoint constrained patterns,* $T = \{t|P\}$ *a term and* $M \in \Pi$ *a pattern compatible with* T*. Then:*

$$Dir_\Pi(T) = Dir_{\Pi'}(T \sqcap M) \text{ where } \Pi' = \{M \in \Pi \mid T \uparrow M\}$$

Proof: Suppose $u \in \text{Dir}_{\Pi'}(T)$. Then $T/u = \{\Omega|\mathcal{T}\}$ and for every $M \in \Pi'$, $u \in \mathcal{O}(M)$ and $M/u \neq \{\Omega|\mathcal{T}\}$ by Lemma 8. As M belongs to Π', $u \in \mathcal{O}(M)$, thus $u \in \mathcal{O}(T \sqcap M)$ and $(T \sqcap M)/u = \{\Omega|\mathcal{T}\}$. In conclusion $u \in \text{Dir}_{\Pi'}(T \sqcap M)$. Conversely, let $u \in \text{Dir}_{\Pi'}(T \sqcap M)$. Then for every $M \in \Pi'$, $M/u \neq \{\Omega|\mathcal{T}\}$ and $(T \sqcap M)/u = \{\Omega|\mathcal{T}\}$. Remember that $\{\Omega|\Omega\Diamond L \vee \Omega\Diamond L'\}$ is always different from $\{\Omega|\mathcal{T}\}$ because $\bullet$ is an instance of $\{\Omega|\mathcal{T}\}$ but not of $\{\Omega|\Omega\Diamond L \vee \Omega\Diamond L'\}$. Consequently $T/u = \{\Omega|\mathcal{T}\}$. Now take any $M \in \Pi$ compatible with T. Then $M \in \Pi'$ and $M/u \neq \{\Omega|\mathcal{T}\}$. In conclusion, $u \in \text{Dir}_\Pi(T)$.

This property allows to look for directions only in the prefixes of patterns.

Theorem 2 *Let* Π *be a set of disjoint constrained patterns. If* Π *is finite, one can decide if* Π *is sequential, one just checks that* $Rmatch_\Pi$ *is sequential at every prefix of* Π.

Proof: If Π is sequential, then Rmatch_Π is sequential at every term T and in particular at every prefix of some element of Π. Conversely, Π is sequential if and only if $\text{Dir}_\Pi(T) \neq \emptyset$ for all T such that $\text{Rmatch}_\Pi(T) = False$. If T is not compatible with Π, there is no instance of T which satisfies the predicate and thus, by definition of the sequentiality, Rmatch_Π is sequential at T. Otherwise, there exists $M \in \Pi$ compatible with T and $\text{Dir}_\Pi(T) \supset \text{Dir}_\Pi(T \sqcap M)$ by Lemma 9. If $\text{Rmatch}_\Pi(T \sqcap M)$ were $True$, either there would exist $M' \in \Pi$ more general than $T \sqcap M$ that would contradict the fact that $\text{Rmatch}_\Pi(T) = False$. Thus $\text{Rmatch}_\Pi(T \sqcap M) = False$ and $\text{Dir}_\Pi(T \sqcap M) \neq \emptyset whichimplies \text{Dir}_\Pi(T) \neq \emptyset$.

Theorem 3 *Optimality and sequentiality are equivalent on the pattern matching algorithms.*

Proof: Let Π be a complete decomposition. Π is sequential if and only if there exists a search tree in which each label (T, x) satisfies $x \in \text{Dir}_\Pi(T)$. The set of terms for which the algorithm does not terminate is generated by the terms $T[x \leftarrow \bullet]$ where (T, x) is a label of the search tree. By definition the algorithm is optimal if and only if the set of terms for which the algorithm does not terminate is generated by the strict set. Thus, we only need to prove that for every prefix T of Π, $x \in \text{Dir}_\Pi(T)$ if and only if $T[x \leftarrow \bullet]$ belongs to the strict set of Π.
An occurrence u of a variable is a direction from T to Π if and only if for every pattern M compatible with T, $M/u \neq \{\Omega | \mathcal{T}\}$ which is equivalent to $T[x \leftarrow \bullet]$ is incompatible with each M in Π. That means $T[x \leftarrow \bullet]$ belongs to the strict set because Π is a complete decomposition.

The theorems state that in order to verify the sequentiality of a match problem it is sufficient to verify it on the set of prefixes of the patterns, so the match is sequential if and only if the search tree of a variable can be built.
We can build now a search tree for a complete decomposition Π which is optimal both in the number of test in each path of the tree and in the number of terms for which the algorithm terminates.

SearchTree(N, Π) =
  T where $Root(T) = N$ and
    if there is no direction from N to Π,
      if $N \in \Pi$, ϵ is the only occurrence of T.
      otherwise the algorithm fails
    otherwise
    let u be one direction of N
    and L be the set $\{F(\ldots) \mid \exists M \in \text{Decomp}(N, \Pi)$ such that $F(\ldots) \uparrow M\}$.
      For each element $l_i \in L$, i is an occurrence of T
      and T/i = SearchTree($N[u \leftarrow l_i], \Pi$)

We have extended the sequentiality to constrained terms which allows to compute optimal algorithms for call by pattern matching. If we complete the initial set of patterns by Ω in order to cover all the cases, we optimize both the success and the failure of the matching. The sequentiality of the set of patterns can be modified by the inclusion of the new element Ω, but, as the search tree covers anyway all the cases, this restriction of the sequentiality has a positive effect on the result.

In case of non-sequential sets of patterns, it is possible to build a search tree, by ignoring some of the patterns. Two possibilities appear: to ignore, during the direction search, either pattern with lower priority or those that prevent the existence of directions.

5 Examples

We wrote a prototype of this method in CAML [10] which is used to generate mechanically all the examples in the paper. In this prototype we only represent constraints of depth 1, other constraints are normalized during the application of substitutions. In all the examples we add the term x at the end of the list of patterns to complete the set.

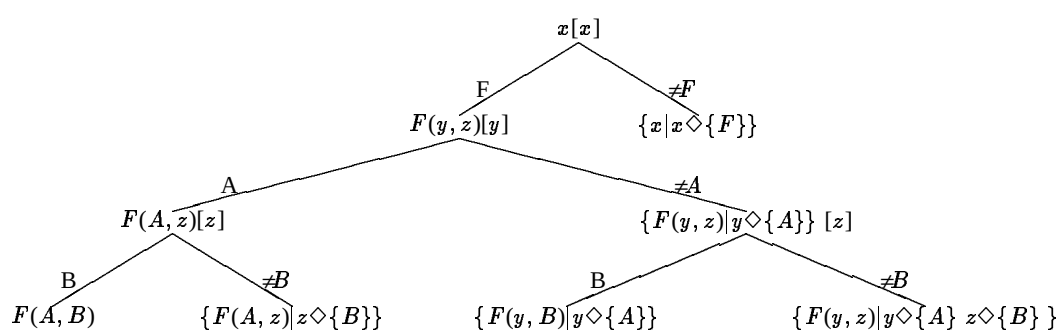
1. With the set of patterns $F(A,B)$, $F(A,z)$, $F(y,B)$ the decomposition produces the following constrained terms:

$$\begin{array}{lll} F(A,B) & \{F(A,z)|z\diamond\{B\}\} & \{F(y,B)|y\diamond\{A\}\} \\ \{F(y,z)|y\diamond\{A\}\ z\diamond\{B\}\} & \{x|x\diamond\{F\}\} & \end{array}$$

   And the strict set is:

$$\bullet\ ,\ F(x,\bullet)\ ,\ F(\bullet,y)$$

   The nodes of search trees are pairs formed by a term and a variable which is a direction in the term. The arcs are labeled by the possible values the direction can take and leaves are represented by the matched patterns.



2. For Berry's example: $G(A,A,x)$, $G(B,y,A)$, $G(z,B,B)$ the decomposition of $G(z,y,x)$ produces:

$$
\begin{array}{lll}
G(A,A,x) & \{G(z,y,x)|z\diamond\{A,B\}\ y\diamond\{B\}\} & \{G(z,y,x)|y\diamond\{A,B\}\ x\diamond\{A\}\} \\
G(B,y,A) & \{G(z,y,x)|z\diamond\{B\}\ y\diamond\{A,B\}\} & \{G(z,y,x)|y\diamond\{A\}\ x\diamond\{A,B\}\} \\
G(z,B,B) & \{G(z,y,x)|z\diamond\{A,B\}\ x\diamond\{B\}\} & \{G(z,y,x)|z\diamond\{A\}\ y\diamond\{B\}\ x\diamond\{A\}\} \\
 & \{G(z,y,x)|z\diamond\{A\}\ x\diamond\{A,B\}\} & \{G(z,y,x)|z\diamond\{B\}\ y\diamond\{A\}\ x\diamond\{B\}\}
\end{array}
$$

As the original patterns have no common instance, they all belong to the decomposition, and there is no direction to start the match.

3. In this example extracted from a CAML program, we try to match lists of Booleans (Nil represents the empty list, $x :: y$ is a list containing the element x followed by the list y).

$$(y :: True :: u)\ ,\ (False :: Nil)\ ,\ Nil$$

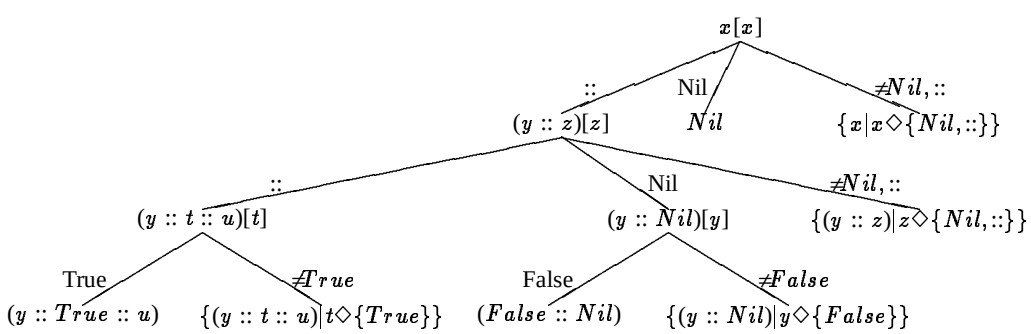
The decomposition of this example is:

$$
\begin{array}{lll}
(y :: True :: u) & Nil & \{(y :: z)|z\diamond\{Nil,::\}\} \\
\{x|x\diamond\{Nil,::\}\} & (False :: Nil) & \{(y :: t :: u)|t\diamond\{True\}\} \\
\{(y :: z)|y\diamond\{False\}\ z\diamond\{::\}\} & &
\end{array}
$$

the strict set is:

$$\bullet\ ,\ y :: \bullet\ ,\ \bullet :: Nil\ ,\ y :: \bullet :: u$$

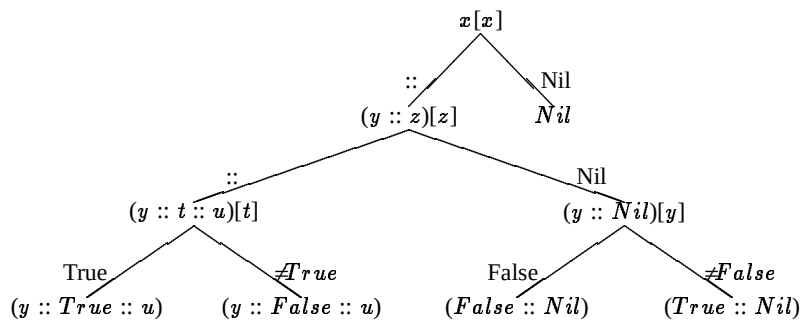
and the search tree is:



In the decomposition of this example, some of the patterns have the constraint $x\diamond\{Nil,::\}$. In a typed language, if Nil and :: are the only list constructors of lists, these patterns represent an empty set. Eliminating them (and assuming that $\neq True$ implies $False$ and that $\neq False$ implies $True$) the decomposition becomes:

$$(y :: True :: u)\ ,\ (False :: Nil)\ ,\ Nil\ ,\ (y :: False :: u)\ ,\ (True :: Nil)$$

which is the set of *minimal extended patterns* as defined in [6]. The search tree now becomes:



4. The sequentiality of a problem might depend on the signature of terms, for instance the decomposition of $F(x, y)$ by the patterns $F(A, A)$, $F(B, B)$ produces:

$$\begin{array}{lll} F(A, A) & \{F(x, y) | y \diamond \{A, B\}\} & \{F(x, y) | x \diamond \{A\}\ y \diamond \{B\}\} \\ F(B, B) & \{F(x, y) | x \diamond \{B\}\ y \diamond \{A\}\} & \{F(x, y) | x \diamond \{A, B\}\} \end{array}$$

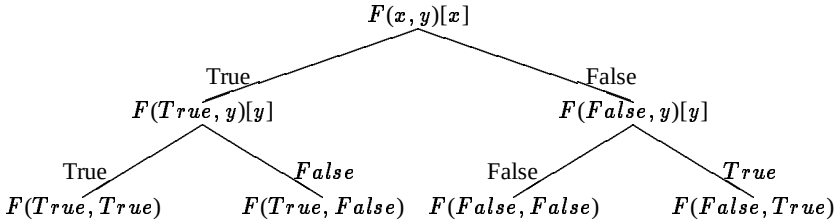
With the following strict set:

$$F(A, \bullet)\ ,\ F(\bullet, A)\ ,\ F(\bullet, \bullet)\ ,\ F(B, \bullet)\ ,\ F(\bullet, B)$$

This problem is not sequential because of the patterns $\{F(x, y) | y \diamond \{A, B\}\}$ and $\{F(x, y) | x \diamond \{A, B\}\}$. However, if the same match problem where given for a type that is defined with only two constants like the Booleans, those two patterns would represent empty sets and thus could be eliminated. In that case, the decomposition of $F(x, y)$ by the patterns $F(True, True)$, $F(False, False)$ produces:

$$F(True, True)\ ,\ F(False, False)\ ,\ F(True, False)\ ,\ F(False, True)$$

And the problem becomes sequential with the search tree:



6 Conclusion

Constrained terms are used to extend the sequentiality to ambiguous sets of patterns. The introduction of an explicit symbol $\bullet$ to represent non-terminating evaluations allows to use constraints for the partially evaluated terms.

The actual compilers for pattern matching use different techniques to improve the code generated for call by pattern matching, like the introduction of heuristics for finding directions, or the analysis of execution tests to improve most frequent cases. Both heuristics and execution tests analysis become unnecessary as our algorithm computes directions and produces an optimal search tree that includes only unavoidable tests.

The elements of the decomposition are exactly the leaves of the optimal search tree which depends inherently on the match problem. The order of complexity of the substitution and of the restriction is in $\mathcal{O}(l)$. For the decomposition it is $\mathcal{O}(m * l)$ and for the search of directions during the construction of a search tree it is $\mathcal{O}(m * l)$ where m is the number of patterns of the match and l is their average size.

The technique presented in this paper allows the implementation of optimal compilers for call by pattern matching in all the languages that support this feature, and encourages language designers to introduce it into new programming languages.

References

1. G. Berry. Séquentialité de l'évaluation formelle des lambda-expressions. In *Proc. 3rd International Colloquium on Programming*, Paris, March 1978. Dunod.

2. R. Burstall, D. MacQueen, and D. Sannella. Hope: An experimental applicative language. In *Lisp and Functional programming conference*, pages 136–143. ACM, 1980.

3. H. Comon. *Unification et disunification. Théorie et applications*. Thèse, Institut National Polytechnique de Grenoble, 1988.

4. R. Harper, R. Milner, and M. Tofte. The definition of Standard ML version 2. LFCS Report Series 88-62, University of Edinburgh, Department of Computer science, The King's Buildings, Edinburgh EH9 3JZ, Scotland, 1988.

5. G. Huet and J.-J. Lévy. Call by need computations in non ambiguous linear term rewriting systems. Rapport IRIA Laboria 359, INRIA, Domaine de Voluceau, Rocquencourt BP105, 78153 Le Chesnay Cedex. FRANCE, 1979.

6. A. Laville. *Evaluation paresseuse des filtrages avec priorité. Application au Langage ML*. Thèse, Université Paris 7, 1988.

7. A. Laville. Implementation of lazy pattern matching algorithms. In H. Ganzinger, editor, *ESOP'88*, pages 298–316. Lecture Notes in Computer Science 300, March 1988.

8. L. Puel. *Bons préordres sur les arbres associés à des ensembles inévitables et preuves de terminaison de systèmes de réécriture*. Thèse d'Etat, Université Paris 7, 1987.

9. L. Puel. Embedding with patterns and associated recursive path ordering. In N. Dershowitz, editor, *RTA*, pages 371–387. Lecture Notes in Computer Science 355, April 1989.

10. P. Weis et al. The CAML reference manual. Available through INRIA, Domaine de Voluceau, Rocquencourt BP105, 78153 Le Chesnay Cedex. FRANCE, January 1989.

PRL Research Reports

Report Number 1: *Incremental Computation of Planar Maps*. Michel Gangnet, Jean-Claude Hervé, Thierry Pudet, and Jean-Manuel Van Thong. May 1989.

Report Number 2: *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic*. Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé. April 1991.

Report Number 3: *Introduction to Programmable Active Memories*. Patrice Bertin, Didier Roncin, and Jean Vuillemin. June 1989.

Report Number 4: *Compiling Pattern Matching by Term Decomposition*. Laurence Puel and Ascánder Suárez. January 1990.

Report Number 5: *The WAM: A (Real) Tutorial*. Hassan Aït-Kaci. January 1990.