Comparison of priority rules in pattern matching and term rewriting

A. Laville

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 878

COMPARISON OF PRIORITY RULES IN PATTERN MATCHING AND TERM REWRITING

Alain LAVILLE

JUILLET 1988

Comparison of Priority Rules in Pattern Matching and Term Rewriting

Comparaison des règles de priorité pour les filtrages ambigus

Alain Laville

I.N.R.I.A. (Projet FORMEL) [1]
and Université de Reims[2]

[1]B.P. 105 78150 Le Chesnay CEDEX France (email: laville@margaux.inria.fr)
[2]B.P. 347 51062 Reims CEDEX France

Abstract :

Term Rewriting Systems as well as case definitions (function definitions, pattern matching, pretty printers ...) frequently use ambiguous cases. One generally adds a rule giving a way to choose the right case among all the ambiguous possibilities. We show in this report that in a useful kind of systems (constructors based, linear system) all the choice rules are essentially equivalent. Moreover the translation from one rule to another may be mechanically done. These results hold both when using lazy evaluation and when using eager evaluation.

Résumé :

Les systèmes de réécriture de termes ainsi que les définitions par cas (appels fonctionnels, "pretty-printers", ...) font un large usage de définitions ambiguës. En général on adjoint au système une règle de choix permettant de résoudre ces ambiguïtés. Ce rapport montre que dans un cas très fréquent (celui des systèmes à constructeurs avec motifs linéaires) toutes les règles de choix sont essentiellement équivalentes. Ce résultat est vrai aussi bien dans une stratégie d'appel par valeur que dans une stratégie d'appel par nécessité. Enfin on montre que la traduction d'une règle de choix vers une autre peut être faite de façon automatique.

Comparison of Priority Rules in Pattern Matching and Term Rewriting

Alain Laville
I.N.R.I.A. (Projet FORMEL) *
and Université de Reims†

Abstract

Term Rewriting Systems as well as case definitions (function definitions, pattern matching, pretty printers ...) frequently use ambiguous cases. One generally adds a rule giving a way to choose the right case among all the ambiguous possibilities. We show in this report that in a useful kind of systems (constructors based, linear system) all the choice rules are essentially equivalent. Moreover the translation from one rule to another may be mechanically done. These results hold both when using lazy evaluation and when using eager evaluation.

1 Introduction

Ambiguous Pattern Matching with priority is of growing use, as well in Term Rewriting Systems (Priority Rewrite Systems, see [1]), as in pure Pattern Matching (selection of cases in pretty printers, see for example [2], case-defined functions of programming languages, see for example [6] or [7], ...). However general priority rules are not too easy to deal with, as is shown in [1]. Particular rules have been more frequently studied and used, such as the two rules we call "textual priority" and "precision priority" (see below for precise definitions).

*B.P. 105 78150 Le Chesnay CEDEX France (email : laville@margaux.inria.fr, laville@inria.uucp)
†B.P. 347 51062 Reims CEDEX France

The choice of a priority to include in a given system seems to have been made without looking at the particular efficiency of the rule. In many cases the rule comes from the way the problem is stated. For example, when driving pretty printers, the precision priority rule is frequently used: it has the meaning that one wants to take advantage of all that is known of the value to print, in order to make it more readable. On the other hand, when defining functions by cases, the textual priority rule has the meaning of a classical nested "if" such as:

$$\begin{array}{rll} \text{if} & < \text{first case} > & \text{then} \ldots \\ \text{else if} & < \text{second case} > & \text{then} \ldots \\ & \vdots & \\ \text{else if} & < \text{last case} > & \text{then} \ldots \end{array}$$

We do not want to go into the debate about the advantages and disadvantages of each rule. Each of them has been advocated as the best one, without fully convincing argumentation. In fact each of the different rules that have been suggested seems to have its particular domain of use, where it is well suited. This fact led us to investigate the possibility of conciliating them in the same system.

Very often there exists some restriction on the kind of patterns one is allowed to use. The most frequent one is that the patterns must belong to what is known in the T.R.S. theory as "Constructors Based Systems". Roughly speaking this means that one only evaluates one function call in one single step (as opposed to nested function calls acting together to yield a result) or that one selects the case to apply using only a fully evaluated part of the scanned value.

Another frequent restriction is to forbid any variable to appear twice in the same pattern. Such patterns are called linear. This restriction means that, during the pattern matching process, patterns behave as "contexts", the variables only being holes to be filled. Of course variables retain their use of binders for parts of the matched value when evaluating the right hand sides of the rules.

The aim of this paper is to show that in the case where the patterns are made of constructors and linear, *all the priorities are essentially equivalent.* The result will hold as well when using a strict evaluation strategy as when using a lazy one. We shall moreover give a mechanical translation from any rule into the two well known ones. In the cases of

such patterns, we give the programmer the possibility of using the priority rule he judges to be the better one. The system will be able to translate it into the textual priority without modifying either the meaning of the program or its behaviour from the lazyness point of view.

This will allow the use with any priority rule of the already known results concerning these two rules. For example, since we preserve the "lazy" properties of the matching, we may in particular use the results of [4] and [5] to get safe evaluation strategies.

2 Definitions and Notations

We shall assume in this paper that we are given a set of *patterns*. These patterns are terms built from a set of *constructors* (including tuple operators) and a set of *variables*. These sets are assumed to be disjoint. We do not allow the same variable to appear twice in the same pattern. For precise definitions about trees, terms, occurrences and related notions we refer to [3], the notations and definitions of which we shall follow in this paper.

Patterns may be for example the left-hand sides of the rules of a Term Rewriting System, or the different cases of a function definition in such languages as DACTL, HOPE, MIRANDA or ML.

Remark 1 If we want to see a list of such patterns as left hand sides of a Term Rewriting System, we top all the patterns with an additional symbol which is not allowed to appear in any pattern (it represents for example the function of which the patterns are the different cases). Technically, a Term Rewriting System that uses such a set of terms as left hand sides of its rules is known as a "constructors based, linear" system.

Usually one says that a value *matches* a pattern if it is an instance of this pattern. We assume here that patterns are allowed to be *ambiguous* (this means that there may exist some overlapping between the patterns). From the definition of the patterns there is no possibility of superposition between a pattern and an inner part of another (or the same) one: One has here to think of patterns as topped by a function symbol (see preceding remark). Hence the only ambiguity is that some values may be instances of several patterns. In such a case several computations (or rewriting) may start from the same value. One way to deal with this feature is to ask the system to be confluent. Since

this is not very easy to prove, another way is to ask computations to be deterministic. In order to achieve this goal, we introduce a *Priority Rule*: this rule associates to each value one and only one of the patterns of which it is an instance. All along this paper we shall deal with the problem of finding which pattern is associated by a Priority Rule to a given value.

Definition 1 Given a set of patterns and a priority rule, we shall say that the value v *matches the pattern* e if v is an instance of e and the priority rule associates e to v.

With such a general definition of Priority it is rather difficult to handle pattern matching processes. Hence we shall restrict the definition of Priority in order to make it tractable.

Definition 2 The restrictions on the Priority Rules are the following ones:

First, we ask the choice of the pattern associated to a value v to only depend on the set of patterns of which v is an instance and not on the particular value v.

Hence we may define from the rule a (binary) relation among patterns: "to have highest priority". Of course this relation is not defined for any pair of patterns, since the rule does not have to choose one of them in all cases.

For the sake of consistency we ask this binary relation to be a (partial in the general case) ordering over the set of patterns. We shall denote this ordering between patterns by the symbol $\preceq$ ($p_1 \preceq p_2$ meaning that p_2 has highest priority than p_1). Moreover it is useful for this ordering to have some compatibility with the instanciation ordering on the terms. Thus we ask that if a pattern p_1 has greater priority than a pattern p_2 then p_2 is not an instance of p_1.

Remark 2 Asking the rule to define an ordering implies that if one has both $p_1 \preceq p_2$ and $p_2 \preceq p_1$ then p_1 and p_2 must be the same pattern.

All the priority rules that have been suggested define an ordering on the set of all the patterns. The compatibility with the instanciation ordering usually is not required. However, if this condition is not enforced, that only means that some patterns are useless: no value can match them. Hence one may fulfil the condition by removing these useless patterns. The matching is not modified (even in its lazy aspect since no partial value could match the removed patterns).

Example: The main two already considered priority rules are the following:

1. Give the set of patterns as an ordered list rather than an unordered set, and always choose the *first one* in the list of which the value to match is an instance. In the sequel we shall call this rule the *textual priority.*

2. Choose the *most defined* one of all the patterns of which the value to match is an instance (recall that a pattern p is more defined than a pattern p' if and only if p is an instance of p'). This rule will be called the *precision priority.*

   This priority rule may generate failure in the pattern matching process: for example if a value is an instance of only two patterns, none of which is more defined than the other. That is the reason why one generally adds a constraint on the set of patterns excluding such failure cases (see below a discussion of conditions related to such a constraint).

Notation: Let E be a set of patterns and $\mathcal{R}$ some priority rule associated to E. We shall denote $\mathcal{F}_{\mathcal{R}}(E)$ (or simply $\mathcal{F}_{\mathcal{R}}$ if there is no possible confusion) the pattern matching process that uses the set of patterns E together with the priority rule $\mathcal{R}$. We shall say that $\mathcal{F}_{\mathcal{R}}(E)$ associates the pattern e to the value v if $\mathcal{R}$ chooses e among the patterns of which v is an instance. We shall write in such a case: $\mathcal{F}_{\mathcal{R}}(E)(v) = e$. If the pattern matching process fails, we shall write: $\mathcal{F}_{\mathcal{R}}(E)(v) = \perp$.

We shall denote by $\mathcal{F}_p(E)$ the matching that uses the set of patterns E together with the precision priority, and $\mathcal{F}_t(\Pi)$ the matching that uses the ordered list of patterns Π together with the textual priority.

Definition 3 We shall say that the pattern matching $\mathcal{F}_{\mathcal{R}}(E)$ may *fail due to ambiguity* if one can find a term which is an instance of at least one of the patterns and for which the rule $\mathcal{R}$ is unable to choose which pattern is matched.

Remark 3 We address now the question of finding practical conditions to detect ambiguity failure cases. It is not too easy to characterize the sets of patterns E and the priority rules $\mathcal{R}$ such that the pattern matching $\mathcal{F}_{\mathcal{R}}(E)$ may fail due to ambiguity. The first idea is to express this condition as: *"Each set of compatible patterns has a $\preceq$ least upper bound which belongs to E"*. In most of the cases this condition is the good one to check and will be fulfiled by the set of patterns. Moreover it is in any case a sufficient condition to

avoid failures due to ambiguity. However it is not always necessary as shows the following example.

Assume the set of constructors is restricted to {true, false} and consider the set E of patterns: {(true,x,y), (x,false,y), (true,false,true), (true,false,false)}, together with the precision priority. The $\preceq$ l.u.b. (true,false,x) of (true,x,y) and (x,false,y) does not belong to E, but there is no possibility of failure due to ambiguity. Moreover, adding the pattern (true, false, x) to the set E simply adds a useless pattern, since no value may be an instance of it without being an instance of pattern with higher priority.

It is possible to give the following necessary and sufficient condition :

$\mathcal{F}_{\mathcal{R}}(E)$ never fails due to ambiguity if and only if for each *ground* (i.e. without variable) term M the set of all the patterns less than or equal to M (according to the instanciation ordering) has a greatest element for the $\preceq$ ordering. We use ground terms rather than terms with variables since values (when fully evaluated) always are ground terms.

This is not very easy to check. It implies that each set of compatible patterns has an $\preceq$ upper bound in the set E. Unfortunately the converse is obviously false. One may see it with the set of patterns {(true,x,y), (x,false,y), (true,false,true)} together with the precision priority. Each subset has an $\preceq$ upper bound, but the pattern matching process fails due to ambiguity when applied to the value (true,false,false).

Definition 4 We assume given a set of patterns E and a priority rule $\mathcal{R}$.

A *pattern matching algorithm* is an algorithm which given any value returns the pattern that value matches (if there exists one) and otherwise fails (it is an algorithm performing the evaluation of the function $\mathcal{F}_{\mathcal{R}}(E)$ on the set of values)

We assume that all the values we shall deal with are represented as trees (they are constructed terms). Moreover we assume that the way of accessing a part of the value is to follow a path from the root of the value (see precise definitions of these paths or "occurrences" and of trees, constructed terms, values ... in [5] or [3]). Hence the pattern matching algorithms we shall use will work in a *top-down way.* Moreover, as the priority rules are intended to ensure determinism of computations, the algorithms used will be *deterministic.*

A top-down, deterministic pattern matching algorithm $\mathcal{A}$ is said to be *lazy* if it never performs useless work when, given a value, it finds which pattern this value matches.

We shall denote $\mathcal{A}(M)$ the prefix of M which is scanned by $\mathcal{A}$ in order to find the pattern that M matches. The algorithm is lazy if this prefix is a minimal one.

Remark 4 With this definition of lazyness, we only ask successful pattern matching to be lazyly performed. One could ask failures to be lazyly detected too. This would complicate all the forthcoming reasonings. Moreover we assume that failure of the pattern matching process denotes a failure of the computation. Hence in the following we only ask success to be lazy.

Definition 5 A term t is *sufficient to decide the matching* $\mathcal{F}_{\mathcal{R}}(E)$ if and only if $\mathcal{F}_{\mathcal{R}}(E)$ associates the same pattern to every value which is an instance of t (and hence to t itself).

Using this notion, we may rephrase the definition of lazy pattern matching algorithm as: $\mathcal{A}$ is lazy if and only if for any value M no term strictly less than $\mathcal{A}(M)$ (for the instanciation ordering) is sufficient to decide the matching.

Notation: We shall use the following notations:

- The symbol $\leq$ will denote the instanciation ordering over the terms ($t \leq t'$ means that t' is an instance of t). We shall say that t' is more defined than t.
- $M \uparrow N$ means that M and N have a common upper bound for the instanciation ordering (and we shall say that M and N are compatible).
- $M \sharp N$ means that they don't have one (and we shall say that M and N are incompatible).
- $\vee$ and $\wedge$ will respectively denote the l.u.b. (when it exists) and the g.l.b. of two terms (always for the instanciation ordering).
- We recall that the symbol $\preceq$ will denote a priority ordering over the set of all the patterns ($p' \preceq p$ means that p has a higher priority than p')

Notation: As such sets will be of frequent use in the sequel, we shall denote by E_t the set $\{ e \in E \ ; \ e \leq t \ \}$.

In order to deal with lazyness in the pattern matching process, we have to introduce some more precise definitions. We shall use the set of truth values {tt, ff} in order to define predicates (i.e. functions with values in this set).

Definition 6 We assume given a set E of patterns (denoted by e_i for $i = 1,\ldots,n$) and a priority rule $\mathcal{R}$. For each $i \in \{1,\ldots,n\}$, the predicate $match_{e_i}$ is defined by $match_{e_i}(M) = \mathrm{tt}$ if and only if the following two conditionsare satisfied:

1. $e_i \leq M$
2. $\forall e \in E \quad ((e \uparrow M) \Rightarrow (e \preceq e_i))$

It is obvious to see that this is equivalent to say that M is sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$ and that the matching associates e_i to M, i.e. $\mathcal{F}_{\mathcal{R}}(E)(M) = e_i$.

We then define the predicate *match*:
$match(M) = \mathrm{tt}$ if and only if $match_{e_i}(M) = \mathrm{tt}$ for some $i \in \{1,\ldots,n\}$.

The meaning of these predicates is the following:
$match_e(M) = \mathrm{tt}$ iff M is sufficiently defined to know that every instance of M is an instance of e and not one of a pattern with higher priority than e. We may remark here that there exists at most one pattern e such that $match_e(M) = \mathrm{tt}$.
$match(M) = \mathrm{tt}$ iff M is sufficiently defined to decide (from M alone) which pattern will be matched by any instance of M. In other words this means that M is sufficient to decide $\mathcal{F}_{\mathcal{R}}(E)$ i.e. $\mathcal{F}_{\mathcal{R}}(E)(M)$ is not empty.

Definition 7 According to the notions defined in [4], we shall call *minimally extended pattern* any term p satisfying the following two conditions:

1. $match(p) = \mathrm{tt}$
2. If a term p' satisfies $p' < p$ then $match(p') = \mathrm{ff}$ ($<$ denotes here the strict ordering associated to $\leq$)

This means that p is minimal (for the instanciation ordering) among the terms which are sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$.

What are the reasons for introducing these *minimally extended patterns*? To answer this question, let us go back to a comparison of pattern matching definitions. Look for example at the following two definitions of the AND function over the booleans:

```
let AND = function
    true, true   -> true
    true, false  -> false
    false, true  -> false
    false, false -> false ;;

let AND = function
    true, true   -> true
    x, y         -> false ;;
```

where the second definition uses a priority rule giving the pattern `true, true` the highest priority.

If one only cares about the results these two definitions are equivalent: Any pattern matching algorithm associates the same result to any value (seen here as a ground term). Assume now we are using a lazy system. We are no more allowed to assimilate a value with a ground term: some parts may be unevaluated. Using the first definition of AND we have to completely evaluate the value prior to applying the function to it. On the other hand, using the second definition we may stop the evaluation of the argument given to the AND function as soon as we find a `false` as part of this argument. In such a system the two definitions of the AND function are not equivalent, it is the same for the two pattern matching processes.

The characterisation of a pattern matching, in a lazy system, is obtained by the *minimal* parts of the value that are needed to recognize which pattern will be matched. These minimal parts are exactly what we called minimally extended patterns (for a more detailed approach of these features see [4] and [5]). Hence two pattern matching systems may be said equivalent in a lazy system if and only if they have the same set of minimally extended patterns.

Definition 8 We shall say that a minimally extended pattern e is *associated to the initial pattern* p, if p has the highest priority among the set E_e of all the initial patterns of which e is an instance. This implies that E_e has a $\preceq$ maximal element, which of course is p. Moreover this implies that $match_e(p) = \mathrm{tt}$.

Lemma 1 *If a partial term t is instance of two distinct minimally extended patterns then these extended patterns are associated to the same initial pattern. As a consequence, if two minimally extended patterns are associated to distinct initial patterns, they are incompatible.*

Proof : It is an obvious consequence of our definitions: the two minimally extended patterns are sufficient to decide the matching, hence they are both associated with the initial pattern that t matches. ■

Proposition 1 *A partial term t is sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$ if and only if it satisfies the following three conditions:*

1. *There exists at least a pattern of which t is an instance.*
2. *The set E_t has a $\preceq$ maximal element, say e.*
3. *Among the patterns compatible with t, e is $\preceq$ maximal.*

Proof :

We shall first show that the condition is a sufficient one. We have to show that $\mathcal{F}_{\mathcal{R}}(E)$ associates the same pattern to every term more defined than t. Let t' be an instance of t, then from the first condition on t, t' is an instance of at least one pattern. All the patterns of which t' is an instance are compatible with t, hence e is the $\preceq$ greatest element of $E_{t'}$. Moreover every pattern compatible with t' is also compatible with t so that e is $\preceq$ maximal among the patterns compatible with t'. It is easy to prove from these three properties that $\mathcal{F}_{\mathcal{R}}(E)$ associates e to t'.

Conversely assume that t is sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$.

Assume moreover that t is not an instance of any pattern and let e be the pattern $\mathcal{F}_{\mathcal{R}}(E)$ associates to any value instance of t. Now t has to be compatible with e (the values are common instances). Since t is not an instance of e, one can find an occurrence u where there is a variable in t and a constructor A (not a variable) in e. Now one may build a value by instanciation of t with another symbol than A at occurrence u. This value cannot be an instance of e: $\mathcal{F}_{\mathcal{R}}(E)$ does not associates e to this value, which contradicts our assumptions. Hence t has to be an instance of e.

Since e is associated by $\mathcal{F}_{\mathcal{R}}(E)$ to any value v more defined than t, it is $\preceq$ greater than any pattern of which v is an instance. Hence it is greater than any pattern of which t is an instance (recall that v is an instance of t). This gives us the second property.

Now let e' be a pattern compatible with t. There exists a value v instance of both t and e'. By hypothesis $\mathcal{F}_{\mathcal{R}}(E)$ associates e to v. This implies that e is $\preceq$ greater than any pattern of which v is an instance. Thus we have $e' \preceq e$, giving us the third part of the result.

■

3 Equivalence Results

We shall show in this section that each pattern matching $\mathcal{F}_{\mathcal{R}}(E)$ may be simulated using a list ordering of the set of patterns E rather than the ordering $\preceq$ *provided that there is no failure due to ambiguity in* $\mathcal{F}_{\mathcal{R}}(E)$.

Theorem 1 *We assume given the set of patterns E with the priority rule $\mathcal{R}$. We totally order E into a list $\Pi = [e_1, \ldots, e_n]$ with the following recursive definitions:*

$$E_0 = E \text{ and } \forall i \in \{1, \ldots, n\} \begin{cases} e_i \text{ is some } \preceq \text{ maximal element of } E_{i-1} \\ E_i = E_{i-1} \setminus \{e_i\} \end{cases}$$

As previously defined we denote by $\mathcal{F}_t(\Pi)$ the pattern matching that uses the patterns of E with the priority defined by their ordering in Π.

Then for every (partial) term M if $\mathcal{F}_{\mathcal{R}}(E)$ associates to M the pattern e_i then $\mathcal{F}_t(\Pi)$ associates to M the same pattern.

Moreover, if $\mathcal{F}_{\mathcal{R}}(E)$ may not fail due to ambiguity, and $\mathcal{F}_t(\Pi)$ asssociates to M the pattern e_i then $\mathcal{F}_{\mathcal{R}}(E)$ associates to M the same pattern (in this case the two pattern matching processes are equivalent).

Proof : Assume that $\mathcal{F}_{\mathcal{R}}(E)$ associates to M the pattern e_i. This means that e_i is the $\preceq$ greatest element of the set $\{e \in E \ ; \ e \text{ is less defined than } M\}$. Hence M is more defined than at least one pattern (say e_i) which implies that the matching $\mathcal{F}_t(\Pi)$ does not fail on M. Denote by e_j the pattern associated to M by $\mathcal{F}_t(\Pi)$, e_j is less defined than M, hence it is smaller than e_i in the $\preceq$ ordering (recall that e_i is $\preceq$ maximal).

From the definition of $\mathcal{F}_t(\Pi)$, we know that j is the least subscript such that e_j is less defined than M. This implies that E_{j-1} contains all the patterns less defined than M,

particularly e_i and e_j. Since e_j is $\preceq$ maximal in E_{j-1}, e_i is not strictly greater (in the $\preceq$ ordering) than e_j. Thus these two patterns are in fact equal. Both $\mathcal{F}_t(\Pi)$ and $\mathcal{F}_{\mathcal{R}}(E)$ associate the same pattern to M.

Conversely assume that $\mathcal{F}_t(\Pi)$ associates to M the pattern e_i. From the first part of this proof we know that if $\mathcal{F}_{\mathcal{R}}(E)$ associates to M a pattern e_j, then e_j has to be the same pattern as e_i. Thus the second part of the theorem only can be false if $\mathcal{F}_{\mathcal{R}}(E)$ fails on M. Since there exists at least one pattern less defined than M (say e_i) this would be a failure due to ambiguity, which we precluded. Hence the two pattern matching processes are equivalent. ■

The preceding result is an equivalence of the two matching in some restricted sense. It does not deal with evaluation strategies. If one only is interested in strict evaluation (or "call-by-value") the theorem gives all what is needed. If one is interested in lazy evaluation (or "call-by-need") the result is no more sufficiently precise. In this case one may state the following result.

Theorem 2 *With the same definitions of E, $\mathcal{R}$ and Π as in theorem 1 assume that there is no failure due to ambiguity for the matching $\mathcal{F}_{\mathcal{R}}(E)$. Then*

- *either there exists a pattern matching algorithm which is lazy for both $\mathcal{F}_{\mathcal{R}}(E)$ and $\mathcal{F}_t(\Pi)$*
- *or none of these pattern matching processes may be done with a lazy pattern matching algorithm.*

Proof : Assume there exists a lazy algorithm $\mathcal{A}$ performing the matching $\mathcal{F}_{\mathcal{R}}(E)$ and let M be a value on which the matching succeeds and returns the pattern e. To apply $\mathcal{F}_{\mathcal{R}}(E)$ to M the algorithm looked at the prefix v of M. Since it is lazy this prefix is a minimal one. Hence if w is a prefix of M strictly less defined than v it must be not sufficient to decide the matching. This implies (see proposition 1) that either w is not an instance of e, or it is compatible with some pattern e' with higher priority than e. Furthermore, w may not be an instance of any pattern with higher priority than e.

We want to show that the same algorithm may be used to lazyly perform the matching $\mathcal{F}_t(\Pi)$. From the preceding theorem, we know that $\mathcal{F}_t(\Pi)$ associates e to M. Hence, it

suffices to show that v is still a minimal prefix for this new matching. Let w be a term strictly less defined than v. If w is not more defined than any pattern it obviously is not sufficient to decide the matching. Otherwise let e_j be the first pattern in the list Π which is less defined than w. We know that e_j does not have higher priority than e. One more time there are two cases to look at.

Either e_j is placed *after* e in the list Π. But then that means that w is not sufficient to decide the matching $\mathcal{F}_t(\Pi)$ since one does not know if the value of which w is a prefix matches e_j or not: the pattern e has higher textual priority than e_j and may not be excluded (e and w are compatible since they have a common instance v).

Or e_j is placed *before* e in the list Π. Since e_j does not have higher priority than e this implies that e_j and e are not comparable for the $\preceq$ ordering. On the other hand we know that e and e_j are compatible. Let M be a value instance of both e and e_j. These two patterns belong to E_M which must have an $\preceq$ maximal element, say e_k. As e_k has higher priority, it is placed in the list Π before both e and e_j. Moreover e_k is compatible with w (M is a common instance). This implies that w is not sufficient to decide whether M matches e_k or not (in the matching $\mathcal{F}_t(\Pi)$).

Conversely we want to show that if there exists an algorithm $\mathcal{A}$ lazyly performing the matching $\mathcal{F}_t(\Pi)$, this algorithm lazyly performs the matching $\mathcal{F}_{\mathcal{R}}(E)$ too.

Let M be a value on which the matching $\mathcal{F}_t(\Pi)$ succeeds, returning the pattern e. Let v be the prefix of the value M that is looked at by the lazy algorithm performing the matching $\mathcal{F}_t(\Pi)$ ($v = \mathcal{A}(M)$). This means that v is an instance of e and incompatible with every pattern placed before e in Π. Since any pattern with higher $\mathcal{R}$ priority than e is placed before it in Π, we see that v is sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$ and that this matching returns e.

It only remains to show that any strict prefix of v is not sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$. Let w be a term strictly less defined than v. If it is not more defined than at least one pattern it is not sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$.

Otherwise let e_i be the first pattern in Π less defined than w. Since w is not sufficient to decide the matching $\mathcal{F}_t(\Pi)$, there exists a pattern e_j, distinct of e_i and compatible with w which is placed in Π before e_i (remark that w cannot be an instance of e_j). This implies that e_i does not have higher priority than e_j.

If we assume $e_i \preceq e_j$, then from the prefix w we cannot infer that M matches (in $\mathcal{F}_{\mathcal{R}}(E)$) the pattern e_j since e_j is not less defined than w. Neither can we infer that M does not match e_j. This exclusion could only be gained if either w was incompatible with e_j (which is false), or w was an instance of a pattern with higher $\mathcal{R}$ priority than e_j. But such a pattern should have been placed in Π *before* e_j (and hence before e_i) so that it could not have w as an instance.

If we assume that e_i and e_j are not $\preceq$ comparable, since these two patterns are compatible, they must have an $\preceq$ common upperbound e_k in E in order to avoid failure due to ambiguity. Since e_k has higher priority than e_i it appears in Π before e_i and hence w is not an instance of e_k. So that we cannot infer from w that M matches e_k. For the same reason as in the preceding case, we cannot exclude from w that M matches e_k.

In all cases w is not sufficient to decide the matching $\mathcal{F}_{\mathcal{R}}(E)$, this means that v is a minimal prefix of M in order to perform the matching process. The algorithm $\mathcal{A}$ is lazy for this matching too. ■

The condition used in the preceding two theorems, that the matching $\mathcal{F}_{\mathcal{R}}(E)$ may not fail due to ambiguity is not a real restriction: The goal of using a priority rule in fact is to avoid such failures. Hence in all reasonable cases, we get from the preceding two theorems a way of simulating any priority rule using the textual priority. In particular this allows us to use the results of [4] and [5] to decide if there exists a lazy pattern matching algorithm and to devise one when it exists.

4 Using the precision priority

The results of the preceding section give a way to simulate any priority rule using textual priority. One may also be interested in simulating the textual priority using another priority rule. This has no sense without strong conditions on the priority rule one wants to use, which has to be defined over the whole set of partial terms. This is needed since one may have to generate patterns that do not belong to the initial list (see below theorem 3). Hence we shall only give here a way of simulating textual priority with precision priority, the only already considered rule which meets this constraint.

We shall use in this section some results on the minimally extended patterns associated to a matching $\mathcal{F}_i(\Pi)$ the proof of which one may find in [4].

Remark 5 Since the priority rule is here the precision one, the ordering $\preceq$ is the instanciation ordering. It is already denoted by $\leq$, hence we drop from now on the notation $\preceq$.

Definition 9 A set of terms is said to be *closed under least upper bound*, with respect to some ordering, if the least upper bound of any subset of E belongs to E as soon as such a l.u.b. exists.

Proposition 2 *Given a finite set E of terms there exists a minimal set of terms containing E and closed under least upper bound operation, with respect to the instanciation ordering. This minimal set is finite. It will be called the* closure *of E under least upper bound operation.*

Proof : Since E is finite and each term in E is finite too, the set of all the occurrences appearing in at least one element of E and the set of symbols used in at least one element of E are both finite. Thus in any l.u.b. of elements of E one only may find a finite set of symbols (those used in the elements of E). Let e be an upper bound of two elements of E e_1 and e_2. Assume one can find an occurrence u where in e there is a symbol other than a variable and such that (for $i = 1, \ldots, 2$):

- either u does not appear in e_i
- or there is a variable at occurrence u in e_i.

Then if we define e' as e where the subtree at u is replaced by a variable, e' is still an upper bound of e_1 and e_2. Thus the least upper bound operation may only create a finite set of terms. Taking all the occurrences and symbols used in E, and building all the possible terms from these sets, we get a set of terms containing E and closed under l.u.b.

On the other hand it is easy to see that the intersection of any family of sets closed under l.u.b. still is a set closed under l.u.b. This gives the closure of E by intersection of all the sets closed under l.u.b. and containing E. ■

Theorem 3 *Let Π be the ordered list of patterns $[p_1; \ldots; p_n]$, there exists a set $E = \{e_1, \ldots, e_q\}$ of partial terms satisfying the following properties:*

1. *$E = \bigcup_{i=1}^{n} E_i$, the sets E_i's being pairwise disjoint, and every element of E_i being an instance of the pattern p_i.*

2. *Every subset of E which has an upper bound (in the set of partial terms) has a least upper bound which belongs to E. The same result holds if one replaces E by any of the E_i's.*

3. *Given a value M, if the matching $\mathcal{F}_t(\Pi)$ fails on M, then the matching $\mathcal{F}_p(E)$ also fails on M. If using the matching $\mathcal{F}_t(\Pi)$, M matches the pattern p_i, then using the matching $\mathcal{F}_p(E)$, M matches one of the elements of E_i.*

4. *If there exists a lazy algorithm performing the matching $\mathcal{F}_t(\Pi)$, the terms sufficient to decide the matching are the same for $\mathcal{F}_t(\Pi)$ and $\mathcal{F}_p(E)$. Hence the same algorithm lazyly performs the matching $\mathcal{F}_p(E)$.*

Proof :

In order to build the sets E_i's, one generates all the minimally extended patterns associated to Π. We define P_i to be the set of all the minimally extended patterns which are instance of p_i. The sets P_i's are pairwise disjoint (see lemma 1). We then define E_i as the closure of P_i for the least upper bound operation. Of course E will simply be the union of the E_i's.

As any element of P_i is incompatible with any element of P_j (if $i \neq j$), this implies that the E_i's also are pairwise disjoint. We get here the first part of the theorem.

To get the second part let T be a subset of E and M be an upper bound of T. It is easy to see that the g.l.b. of all the upper bounds of T still is an upper bound for T. Of course it is the least one. Since the elements of T are compatible (they have a common upper bound) they belong to the same E_i. From the building of E_i they have a common least upper bound which belongs to E_i.

Assume that the matching $\mathcal{F}_t(\Pi)$ fails on the value M. This means that none of the p_i's is less defined than M. If M were an instance of an element e of E, e would belong to some E_i and hence would be an instance of p_i. This implies that M would be an instance of p_i, contradicting our assumption. As M can not be an instance of any element of E the matching $\mathcal{F}_p(E)$ fails on M.

Assume that the matching $\mathcal{F}_t(\Pi)$ associates to M the pattern p_i. This implies that M is an instance of some element of the set P_i which is a subset of E_i. As all the elements of E which are less defined than M have to be compatible, they must belong to the same

E_i. From the second part of the theorem we know that this set E_M of elements of E has a least upperbound e_{i_0} which belongs to E_i. By definition of the least upper bound, e_{i_0} has to be less defined than M and is in fact the maximal element of E_M. Thus the matching $\mathcal{F}_p(E)$ succeeds on M and associates to this term an element of E_i. This ends the proof of the third part of the theorem.

If the matching $\mathcal{F}_t(\Pi)$ may be done using a lazy algorithm, we know (see [4]) that the minimally extended patterns are pairwise incompatible. In such a case, the sets P_i and E_i are identical (for all i). There is no ambiguity between the patterns of $\mathcal{F}_p(E)$, this matching is the usual one using the set of all the minimally extended patterns. It is exactly the same matching as $\mathcal{F}_t(\Pi)$. ■

Remark 6 Although the third part of the theorem may be viewed as an equivalence result between $\mathcal{F}_p(E)$ and $\mathcal{F}_t(\Pi)$, the fourth one is not symetrical in the two matchings.

In fact the two matchings are only equivalent in a strict system, or in a lazy one if we forbid the use of patterns set leading to a matching without a lazy associated algorithm.

Example: We assume the set of constructors to be restricted to $\{true,\ false\}$ and we start with the list of patterns $\Pi = [(true,true,x)\ ;\ (x,y,true)]$. There is obviously no lazy pattern matching algorithm associated to $\mathcal{F}_t(\Pi)$. We shall try to simulate this matching using the precision priority rule. We have to define a set E of patterns such that $\mathcal{F}_p(E)$ has the same lazy behaviour as $\mathcal{F}_t(\Pi)$: we want the terms sufficient to decide the matching to be the same in both cases.

The set of minimally extended patterns associated to $\mathcal{F}_t(\Pi)$ is here

$$MEP = \{(true,true,x),\ (false,x,true),\ (x,false,true)\}.$$

If we try to define E as MEP, we get a case of ambiguity failure when applying the pattern matching process on the value $(false,false,true)$. Since such a failure may not occur in the matching $\mathcal{F}_t(\Pi)$ this is not a good choice for E.

If we put into E a pattern without variable, at least one of the minimally extended patterns associated to $\mathcal{F}_t(\Pi)$ is not sufficient to decide $\mathcal{F}_p(E)$. This implies that we have to put into E the pattern $(true,true,x)$. If we try to modify another of the minimally extended patterns, we cannot change the "$true$" part which is needed from the initial

patterns. We cannot replace the variable: this would give a pattern without variable, which we are not allowed to do as we have already seen. Replacing the "$false$" part would lead to an ambiguity with the first pattern.

In all cases either we allow some failure due to ambiguity, or we prevent some of the minimally extended patterns associated to $\mathcal{F}_i(\Pi)$ to be sufficient to decide $\mathcal{F}_p(E)$.

References

[1] J. Baeten, J. Bergstra, J. Klop, "Priority Rewrite Systems", Report CS-R8407, Center for Mathematics and Computer Science, Amsterdam ; also Proceedings of the 3^{rd} Conference on Rewriting Techniques and Applications, Bordeaux 1987, L.N.C.S. 256, pp 83-94, Springer Verlag

[2] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, "CENTAUR : The System", I.N.R.I.A. Research Report 777, December 1987.

[3] G. Huet, "Confluent Reductions : Abstract Properties and Applications to Term Rewriting Systems", J.A.C.M., Vol. 27, No. 4, October 1980, pp. 797-821

[4] A. Laville, "Lazy Pattern Matching in the ML Language", 7^{th} Conference on Foundations of Software Technology and Theoretical Computer Science, Pune (India), December 1987, L.N.C.S. 287, pp 400-419

[5] A. Laville, "Evaluation Paresseuse des Filtrages avec Priorité. Application au Langage ML", Thèse de Doctorat de l'Université Paris 7, February 1988.

[6] D. Turner, "Miranda a Non Strict Functional Language with Polymorphic Types", in J.P. Jouannaud ed. Functional Programming Languages and Computer Architecture, L.N.C.S. 201, Springer Verlag, 1985

[7] P. Weis et al., "The CAML Reference Manual", I.N.R.I.A., January 1988

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique