Compilation of Pattern Matching with Associative-Commutative Functions

E. Kounalis D. Lugiez *
CRIN-INRIA
Nancy - FRANCE

Abstract

This paper deals with the problem of compiling pattern-matching of associative-commutative function definitions in a functional-like language where the evaluation is performed by head-rewriting. We propose an effective algorithm (i.e which does not rely on some -unknown- procedure for computing complements of terms modulo AC-axioms) to compile the pattern-matching process. For this purpose, we introduce the concept of *pattern trees*. We also get for free a test of the completeness of such definitions (with respect to head-rewriting), provided some linearity condition is met. Our method will ensure an efficient pattern-matching process at evaluation time.

1 Introduction

Functional languages allow to define functions by a set of rules such as:

$$\begin{cases} 0 + y & \rightarrow y \\ x + s(y) & \rightarrow s(x + y) \end{cases}$$

together with a priority on this set of rules (here: apply first applicable rule first) to deal with ambiguous patterns (a pattern is the left-hand side of a rule). When an expression is to be evaluated, it is matched against the patterns (according to the given priority) and when the matching succeeds the evaluator returns the right-hand side expression (which can be evaluated again). However, for efficiency purpose, the above definition should be compiled into a piece of code which is more suited to machine like:

```
(x + y) =  if  x = 0          then  y
           elsif  y=s(y')     then  let y=s(y') in  s(x + y')
           else  no−match
```

*author's address: CRIN, BP 239, 54506 Vandoeuvre les Nancy Cedex, FRANCE. e-mail kounalis,lugiez@loria.crin.fr

Nowadays, many algorithms for compiling definition by pattern matching exist see [Aug85, PJ87, Sch88]. Unfortunately many functions arising in practice are defined with equations which cannot be oriented into rewrite rules as above. The most important case is that of Associative-Commutative functions (in short AC-functions), i.e functions f such that $f(x,y) = f(y,x)$ and $f(x,f(y,z)) = f(f(x,y),z)$ which are very common (the + operator, union operator in sets,...). For example, the definition of the + operator above should be extended by the declaration that + is AC (otherwise it does not define the usual + over the integers: the definition is not *complete* since no rule applies to +(s(x),0)). No present compilation algorithm for pattern-matching handles the case of AC-functions. Since Σ_3^0 formulae built on the equality predicate are undecidable when AC-functions are involved [Tre90], and since the quasi-reducibility property with AC-functions is also undecidable [KNRZ87], the problem appears to be of an intrinsic complexity.

We give a new algorithm to compile pattern-matching definitions into *if then else* expressions in a functional-like language using head-rewriting. Moreover, our algorithm provides a decision procedure for the completeness of *left-linear* definitions. We also have sufficient conditions for completeness in the general case.

The key concept in compiling functions defined through pattern-matching is that a term (with variables) is a representation of the (infinite) set of its ground-instances. Therefore, the main problem to deal with is "how do the ground instances of a given term behave with respect to a finite set of patterns?". The concept of *quasi-instance* that we introduce for solving this question, is similar to that of quasi-reducibility, studied in [JK86, Kou90]). Pattern trees (as defined in section 4) whose nodes characterize the behaviour of the ground instances of a term, will provide the simple tool to investigate this notion. In addition, they tell us which position(s) and which symbol(s) must be checked to know which rule to apply. Section 2 presents informally the method on a simple example, section 3 introduces the main notations and concepts on terms and pattern-matching definitions. Section 4 presents pattern trees and the most significant theorems, whereas section 5 describes the pattern matching algorithm.

2 An Example

In this section, we describe informally the problem and our solution on an example. Missing definitions can be found in section 3.

When defining the multiplication operator $*$ on the natural numbers, constructed from zero, denoted by 0, and the successor operator, denoted by s, provided with the sum operator +, one may write the following case definition:

$$\begin{cases} *(x,0) & \rightarrow 0 \\ *(x,s(0)) & \rightarrow x \\ *(s(x),y) & \rightarrow +(*(x,y),x) \end{cases}$$

together with the declaration that $*$ is Associative-Commutative. Since the patterns (a pattern is a left-hand side of a rule) are ambiguous (the term $*(s(0),0)$ is matched modulo AC by the three patterns $*(0,x), *(s(0),x), *(s(x),y)$ a priority rule is necessary and we choose the textual ordering which is more suitable in our case (this is the usual priority in

functional programming [1]). The requirement that the patterns are non-ambiguous would add an unnecessary burden to the programmer's task.

Two main issues must be addressed now: is the definition complete, and how to select a pattern?

The first question means that any ground term -i.e without variables- with $*$ as root-symbol, can be evaluated (using head-rewriting) to a term containing only 0 and s. That is the same as proving that any ground instance of $*(x, y)$ is a ground instance of some pattern. [2]

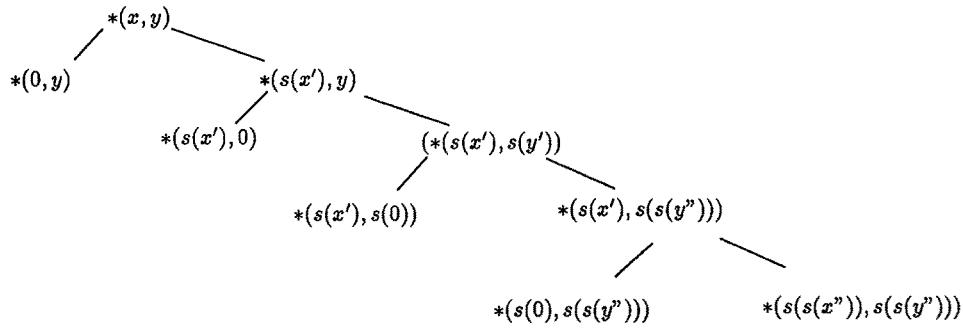
The second question (find the right pattern) is a key-one since pattern matching is a basic operation for evaluation. A naive answer would be: use a matching algorithm modulo AC, match the term to evaluate against the first pattern, if it fails match against the second one, and so on... Unfortunately, this is not realistic since pattern matching modulo AC is NP complete (see [BKN87]) and many repeated invocations of this algorithm at evaluation time will cause the process to be unefficient (even when no AC function are involved, pattern matching compilation algorithms have been designed to avoid redundant calculations [Aug85, PJ87, Sch88]). Let us detail this solution with the term $*(s(s(0)), s(s(0)))$.

- match $*(s(s(0)), s(s(0)))$ against $*(x, 0)$. Matching fails, as well as unification, therefore the first rule can be dropped. This step requires to solve the diophantine equation $x_1 + x_2 = x_3 + x_4$.
- match $*(s(s(0)), s(s(0)))$ against $*(x, s(0))$. Both matching and unification fail, therefore the second rule is discarded. Again the above diophantine equation must be solved and its four minimal solutions must be computed and combined.
- eventually the third rule matches and $*(s(s(0)), s(s(0)))$ is rewritten to $+(s(s(0)), *(s(0), s(s(0))))$. (other rewriting are possible since several matching substitutions exist)

Our approach gives a better solution since most of the computations modulo AC are rejected at compile-time. Firstly, the compiler computes a *minimal complete pattern tree* (see section 4 for definition) which is a tree whose nodes are instances of the term $*(x, y)$. The successors of a node are computed by instantiating a variable by all possible elementary constructions 0 and $s(z)$, z being a new variable (for simplicity we assume 0 and s as constructors, $+$ being completely defined, in section 5 , the same example is treated without the assumption that 0 and s are constructors). A minimal complete tree, computed by the compilation algorithm, is:

[1] the textual ordering is not always the more relevant one, see [Lav88] for a comparison of priorities

[2] Functional definitions often ends with a rule like $*(x, y) \rightarrow$ *undefined*, but this does not really answer the completeness issue



Since each leaf of the tree is a term matched by some pattern, the definition is complete (no case have been missed) which answers the first question. From this tree, the compiler will produce some code which is called whenever a term of the form $*(x,y)$ must be evaluated. The compiler will generate a piece of code which looks like[3]:

if x=0 **then** *apply-rule* 1
else let x =s(x') **in** **if** y=0 **then** *apply-rule* 1
else let y = s(y') **in** **if** x'= 0 **then** *apply-rule* 2
else let x' = s(x") **in** **if** y'=0 **then** *apply-rule* 2
else *apply-rule* 3

The evaluation of this code is obviously much more efficient than the painful naive approach. The compiler has replaced the initial set of patterns $\{*(x,0),*(x,s(0)),*(s(x),y)\}$ by the new one $\{*(0,y),*(s(x'),0),*(s(x'),s(0)),*(s(0),s(y')),*(s(s(x'')),s(s(y'')))\}$.

3 The Framework

3.1 Terms, Substitutions, Matching and Unification

We need some definitions on terms and substitutions. The term algebra $T_\Sigma(X)$ is constructed from a finite set Σ of function symbols and from a denumerable set X of variables (denoted by $x,y,z\ldots$). We *do not* require that Σ is partitioned into constructors and non-constructors. When such a partition exists, our algorithm can be simplified, but this paper deals with the general case. We will assume that some of the functions are associative and commutative. These functions are called AC-functions.

A *ground* term is a term without variables, a *linear* term is a term where a variable may occur only once. $Var(t)$ is the set of variables of a term t. The notation $\overline{x_i}$ denotes the vector $x_1,\ldots,x_m$. *Substitutions* are the morphisms on terms, the *domain* of a substitution σ is the set $Dom(\sigma)=\{x\in X; x\sigma\neq x\}$ where $x\sigma$ denotes the application of σ to x. A substitution will be described by $\{x_1\leftarrow t_1, x_2\leftarrow t_2,\ldots\}$ with $Dom(\sigma)=\{x_1,x_2,\ldots\}$.

The smallest equivalence relation generated by the equations $f(f(x,y),z)=f(x,f(y,z))$ and $f(x,y)=f(y,z)$ is denoted by $=_{AC}$. A term t is *AC-unifiable* with another term s if there exists a substitution σ such that $t\sigma=_{AC}s\sigma$. If $t'=_{AC}t\sigma$ with σ a substitution,

[3]because we must deal with *flattened* terms and not terms, the actual code is different, but has the same flavor

we say that t' is an *AC-instance* of t, and that t is an AC-match of t'. A substitution unifying t and t' (resp matching) is called a unifier of t and t' (resp. match).

The basic notion in our approach is that of **quasi-instance**: a term t is a quasi-instance of a set of terms $l_1, \ldots, l_n$ iff for each ground substitution σ there exists a substitution θ, an index i s.t. $t\sigma =_{AC} l_i\theta$.

For instance, the terms $+(0,+(s(0),y))$ and $+(s(0),z)$ are AC-unifiable with $\sigma = \{z \leftarrow +(0,y)\}$, the term $+(0,+(s(0),x))$ is an AC-instance of the term $+(x,y)$, with a matching substitution $\sigma = \{y \leftarrow +(0,s(0))\}$ and $+(0,x)$ is a quasi-instance of $\{+(0,0), +(0,+(y,z)),+(s(y),0)\}$ if $\Sigma = \{0,s,+\}$.

We recall that matching modulo AC as well as unifiability modulo AC are decidable, finitary i.e there exists a finite set -which can very large- of (most general) matching substitutions -resp unifiers-, but both problems are NP-complete [BKN87]. Therefore any computation with AC-axioms will be costly except if more specific cases are considered.

3.2 Flattened terms and Paths

An AC function can be seen as a function of arbitrary arity and we consider *flattened* terms, i.e terms such that no argument of an AC function f has f as top-symbol. For example, $flat(t)$, the flattened term corresponding to $t = +(+(a,b),c)$ is $+(a,b,c)$ if $+$ is declared to be AC. The permutative class of a flattened term t, denoted by $[t]$, is the set of flattened term s such that $t =_{AC} s$. For instance $[+(a,b,c)] = \{+(a,b,c),+(a,c,b), +(b,a,c),+(c,a,b),+(b,c,a),+(c,b,a)\}$. All previous definitions for terms hold for flattened terms. The number of arguments of a flattened term is defined by $arg(f(t_1,\ldots,t_n)) = n$. For instance, $arg(f(s(x),s(0))) = 2$, and $arg(0) = 0$. The depth of a term is defined by $depth(f(t_1,\ldots,t_n)) = 1 + Max_{i=1,\ldots,n}(depth(t_i))$, and $depth(x) = depth(c) = 1$ if x is a variable and c a constant.

The *successors* of a flattened term t (with respect to $x \in Var(t)$) are the element of $Suc(x,t) = \{s \ flattened\ term; s =_{AC} t\sigma),\ \sigma = \{x \leftarrow f(x_1,\ldots,x_n)\}, x_i\ fresh\ distinct\ variables\ where\ f\ ranges\ over\ \Sigma\}$. For example, if $\Sigma = \{0,s,+\}$ and $t = +(x,y)$ then $Suc(x,t) = \{+(0,y),+(s(x_1),y),+(x_2,x_3,y)\}$

One problem to solve is to travel through a flattened term (independently of what representant of the permutative class is given). For this purpose, we use sequences of symbols which we call *paths*, defined as follows:

- ϵ the empty sequence is a path,
- $f.c$ is a path if c is a path and f is an AC-function symbol,
- $f.i.c$ is a path if c is a path and f a function symbol which is not AC, of arity n with $n \geq i > 0$.

In general several different subterms can be reached following a path (i.e a sequence of symbols), for example following the path $+.s.1$ in $+(s(0),s(s(x)))$, one can reach either 0 or $s(x)$. We define $\|t\|_c$ the set of subterms of t reachable at c by:

- $\|t\|_\epsilon = \{t\}$,
- $\|t\|_{f.c}$= if $t = f(t_1, \ldots, t_n)$ where f is an AC-symbol then $\cup_i \|t_i\|_c$ else undefined,
- $\|t\|_{f.i.c}$= if $t = f(t_1, \ldots, t_n)$ where f is not an AC-symbol and where $n \geq i \geq 0$ then $\|t_i\|_c$ else undefined.

For example $\|+(s(0), s(s(x)))\|_{+.s.1} = \{0, s(x)\}$, $\|+(s(0), s(s(x)))\|_{+.s.1.s.1} = \{x\}$, and $\|+(s(0), s(s(x)))\|_{+.s.1.+}$ is undefined.

We define $path(t)$ to be the multiset of paths from the root to the leaves in t. For example $path(+(s(s(x)), s(y))) = \{+.s.1.s.1, +.s.1\}$.

3.3 Definition by Pattern Matching

Let $\mathcal{L}$ be a functional language. In $\mathcal{L}$, a term represents the set of its ground instances. Moreover, the evaluation is restricted to ground terms only, and the operational semantics of the language follows an *outermost reduction scheme*, not a *call by value* one. This approach is better tailored for lazy languages but this paper does not intend to discuss lasy-evaluation in presence of AC-function at full length. Anyway, this question appears to be a difficult one since the classical algorithms do not generalize fairly. For example, if the strategy of evaluation is "look at the first argument first", the evaluation of $f(\bot, 0)$ with f being AC and with a rule $f(x, 0) \rightarrow 0$, will give $\bot$, while one expect 0. Changing the strategy to "look at the second occurrence" will not clear the situation, since the same problem happens with $f(0, \bot)$. The problem is that the matching is not sequential, see [HL79, Lav87, PS90] for a definition of this notion. A possible solution is to have a set of unavoidable positions (and not a single one), which will provide some kind of minimal laziness [SR90]. In this paper, we stick to head-rewriting, without discussing the lazyness issue.

We suppose that $\mathcal{L}$ allows the definition of a function by pattern matching. A definition of a function f by pattern matching is a set of rules: $\begin{cases} l_1 & \rightarrow & exp_1 \\ l_2 & \rightarrow & exp_2 \\ \ldots & \ldots & \ldots \\ l_n & \rightarrow & exp_n \end{cases}$

where the l_i (also called patterns in the following) are terms with f as top-symbol and exp_i are expressions of the functional language $\mathcal{L}$. The definition is called left-linear when all the l_i are linear. To avoid junk when the same term matchs two (or more) patterns, one must give a priority on the set of rules. Our priority rule is the textual ordering, as usual in functional languages [PJ87]. A definition of an AC-function f by pattern matching is an usual definition by pattern matching augmented with the two equations:

$\begin{cases} f(x, y) = f(y, x) \\ f(x, f(y, z)) = f(f(x, y), z) \end{cases}$

and the semantics of the pattern matching is modified in the following way: matching is replaced by matching modulo AC.

4 Pattern trees

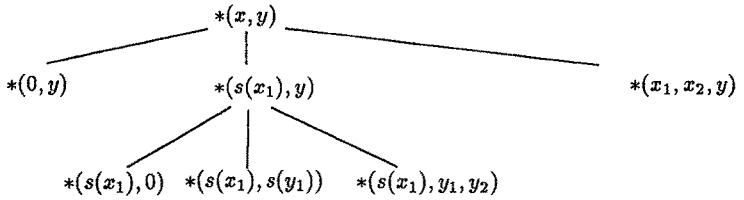
4.1 Complete Pattern Trees

The key notion of our approach is that of *pattern trees*. Given a term t and a set of terms $R = \{l_1, \ldots, l_m\}$ (the l_i are also called *patterns* in the following), a pattern tree will help us describing how the ground instances of t behave (modulo AC) with respect to the ground instances of the patterns. For simplicity we do not distinguish a node of a tree and its label.

Definition 1 *Given a flattened term t, a* pattern tree *associated to t is a tree T such that:*

- *the root of T is t.*
- *if s is an internal node n, the successor nodes of n are the terms of $Suc(x, s)$ with $x \in Var(s)$.*

Example A pattern tree of the term $t = *(x, y)$ is



Pattern trees enjoy the following fundamental property:

Proposition 1 *Let T be a pattern tree of a term t,*

- *each node is an instance of t*
- *the set of ground instances of t is equal to the set of ground instances of leaves, more precisely for any ground substitution θ there exist a leaf s, and a ground substitution σ such that $t\theta = s\sigma$.*

Pattern trees allow us to split the set of instances of t but we still do not know which are the useful ones. The notion of *extensible* node permits to choose which nodes must be expanded.

Definition 2 *Let t be a variable of t, we say that the flattened term t is extensible in x with respect to the flattened term t' iff:*

- *t and t' are AC-unifiable, and*
- *there exists a path c such that $x \in \|t\|_c$ and $\|t'\|_c$ is defined (i.e there is a path leading to x in t, which also exists in t'),*

- *if* $c = c'.f$ *with* f *an AC-symbol, there are some* $s = f(\ldots) \in \|t\|_{c'}$ *and some* $s' = f(\ldots) \in \|t'\|_c$ *such that* $x \in Var(s)$ *and* $arg(s) \leq arg(s')$.

One must remark that the path c' occurring in the third condition may be empty. Roughly speaking, this means that the position of x in t corresponds to either a position of a function symbol in t', or to the position of a variable argument of an AC-symbol in t' (this AC-symbol having less (or the same number of) arguments in t than in t'). For instance, $t = *(x, s(y))$ is extensible in x but not in y with respect to $t' = *(x', 0)$ (since $*.s.1$ cannot be the prefix of any path of t'), $t = *(x, y, z, w)$ is not extensible with respect to $t' = *(0, x')$ nor to $t'' = *(s(x'), y')$ in any variable x or y or z or w.

From now f is an AC-function and $R = \{l_1, \ldots, l_n\}$ is a list of flattened terms with f as top-symbol (the patterns). We classify the terms occurring in a pattern tree:

Definition 3 *Let* T *be a pattern tree of* $t = f(x, y)$ *with respect to* R, *a node* s *of* T *is said to be of:*

- type 1 *if* s *is an instance of some* l_i *modulo* AC *and is not* AC*-unifiable with any* l_j *for* $1 \leq j < i$.
- type 2 *if* s *is not* AC*-unifiable with* l_i *for* $1 \leq i \leq n$
- type 3 *if* s *is not a ground term,* s *is neither of type 1 nor 2, and* s *is not extensible in any variable with respect to* $\{l_1, \ldots, l_n\}$

A pattern tree is complete *for* R *iff each leaf has type 1,2 or 3. A complete pattern tree is minimal iff it does not contain a complete proper subtree.*

Example Let us consider the pattern tree given for $*(x, y)$. It is not complete for $R = \{l_1 = *(0, x),\ l_2 = *(s(0), y)\}$ since $*(s(x_1), s(y_1))$ in extensible in x_1 and y_1 (while $*(0, y)$, $*(s(x_1), 0)$ are instances of l_1, and $*(x_1, x_2, y)$ and $*(s(x_1), y_1, y_2)$ are of type 3).

A minimal complete pattern tree is obtained by extending first $*(s(x_1), s(y_1))$, getting $*(s(0), s(y_1))$ of type 1, $*(s(s(x_1)), s(y_1))$ and $*(s(*(x_3, x_4)), s(y_1))$ which are extensible in y_1. Extending these nodes yields $*(s(s(x_1)), s(0))$ and $*(s(*(x_3, x_4)), s(0))$ of type 1, $*(s(*(x_3, x_4)), s(*(y_2, y_3)))$, $*(s(*(x_3, x_4)),\ s(s(y_4)))$ of type 2.

The next point is to prove that complete pattern trees exist (therefore minimal complete trees).

Theorem 1 *Given a flattened term* $t = f(x, y)$ *and a set* $R = \{l_1, \ldots, l_n\}$ *of flattened terms, there exists a finite pattern tree of* t *complete for* R.

Sketch of the proof: when one extends a node, its sons have subterms either deeper or wider, therefore eventually one gets nodes of type 1 2 or 3 □

In the following, the complete patterns trees that we consider are minimal ones.

4.2 Quasi-instances and Pattern Trees

Now we extract the information provided by the leaves of a minimal complete pattern tree. We deal with types 1, 2 and 3 respectively.

Lemma 1 *Let $R = \{l_1, \ldots, l_n\}$, and let T be a pattern tree of $t = f(x, y)$ complete for R. Then a node of type 1 is a quasi-instance of R and a node of type 2 is not a quasi-instance of R.*

The proof is straightforward. Let us now deal with the more complicated situation: terms of type 3. We divide this class into two subclasses: type 3a and type 3b. Roughly speaking, terms of type 3a are the terms which are deep but not too wide, and a term of type 3b has a subterm wider than a corresponding subterm in some l_i.

Definition 4 *A node t of type 3 is of type 3a iff for each AC-function symbol g, for each $c = c'.g$ such that $\|t_{|c}\|$ is defined, for each l_i AC-unifiable with t, one has $Max\{arg(s), s = g(\ldots) \in \|t_{|c'}\|\} \leq Min\{arg(l), l = g(\ldots) \in \|l_{i|c'}\|\}$. Otherwise t is said to be of type 3b.*

An example of type 3a is given by $+(s(x), s(y))$ when $R = \{+(x, x)\}$.

Theorem 2 *Let T be a pattern tree of a term t complete with respect to R, then a node of type 3a is not a quasi-instance of $\{l_1, \ldots, l_n\}$.*

Proof. We show that a term of type 3a which is a quasi-instance of $l_1, \ldots, l_n$ must be an instance of some l_i, which contradicts the definition of type 3a.

The first point to notice is that, since there is a term of type 3a, there are terms of any depth: a term of type 3a has a depth greater or equal to 3, therefore there are at least two symbols of functions of arity greater than 0. Let f and g denotes these symbols, then one may consider terms like $f(\ldots g(\ldots, f(\ldots)))$, which have depth $n + 1$ where n is the number of alternances.

Let t be of type 3a, and $R' = \{l_i, i \in I\}$ be the set of elements of R AC-unifiable with t. We construct $\theta = \theta_1 \ldots \theta_n$ as follows: let $\{x_1, \ldots, x_n\}$ be the set of variables of t, and θ_i such that

- $depth(x_1\theta_1) > Max(depth(l_i)) + depth(t) + 1$
- $depth(x_i\theta_i) > depth(t\theta_1 \ldots \theta_{i-1})$ for $i > 1$.

Let l such that $t\theta =_{AC} l\rho$. Since t is of type 3a, for any variable y of l

- either $y\rho$ is equal to a ground subterm of t
- or $y\rho$ is the instantiation of st a subterm of t which contains a variable x_i
- or $y\rho = g(\ldots, sg, \ldots)$ with g AC and where sg is the instantiation of st, a subterm of t which contains a variable x_i.

Let y_n be the variable of l such that $y_n\rho$ contains $st_n\theta$ where st_n is a subterm of t containing x_n. Because of the condition on $x_n\theta$, y must be linear in l (remember that t is linear). Let Θ_1 be the product of the θ_j for the $x_j \notin Var(st_n)$ and let σ_n be the substitution such that $y_n \leftarrow st_n$. By construction one has $t\Theta_n = l\sigma_n$.

Now we repeat the process with $x_{n_1} \in t\Theta_n$ such that $depth(x_{n_1}\theta)$ is maximal and so on with $x_{n_2}, \ldots$ until all the variables of t have been treated. Therefore we end with a substitution σ such that $t = l\sigma$, yielding a contradiction (t is of type 3a and cannot be an instance of any $l \in R$). □

The difficult problem is to deal with a term of type 3b. The solution will be to test if some terms (which are instances of the given term) are AC-instances of the left-hand side of the rules. These terms will form a *test-set* $TS(t)$ of a term t of type 3b.

Definition 5 *$TS(t)$ is the set of flattened terms $s = flat(t\sigma)$ such that:*

- *s is not too deep, i.e $depth(s) \leq Max\{depth(l), l \in R\} + 1$*
- *no variable position in s can be a position in some l, more precisely for each $x \in Var(s)$, for each path c such that $x \in \|s\|_c$, then $\|l\|_c$ is undefined for all $l \in R$ unifiable with s.*
- *s is not too wide, i.e for each path $c.f$ in s where f is an AC-symbol, then*
  - *either $\|l\|_{c.f}$ is undefined for all $l \in R$ unifiable with s, in this case we require that $Max\{arg(u); u \in \|s\|_c$ and $u = f(\ldots)\} \leq 2$*
  - *or $\|l\|_{c.f}$ is defined for some $l \in R$ unifiable with s, and we require that $Max\{arg(u), u \in \|s\|_c$ and $u = f(\ldots)\} \leq Max\{arg(v), v = f(\ldots)$ and $v \in \|l\|_c$ when $\|l\|_{c.f}$ is defined$\}$ +1.*

From the definition, one sees that a test-set is finite. For instance, a test-set for $t = +(0, y, z)$ and $R = \{+(x, 0)\}$ is $TS(t) = \{+(0,0,0), +(0,0,s(z')), +(0,s(y'),s(z')), +(0,s(y'),0)\}$ (from which we may remove one of the two AC-equivalent terms $+(0,s(y'),0)$ and $+(0,0,s(z'))$).

Two theorems fix the case of type 3b. From now, we suppose that one can build terms of an arbitrary depth.[4]

Theorem 3 *If t of type 3b is a quasi-instance of R, then for each $s \in TS(t)$, there exists $l \in R$ and a substitution σ such that $t =_{AC} l\sigma$.*

Proof. The proof is obvious for the ground terms of $TS(t)$, since t is a quasi-instance of R. What remains to do is to deal with the non-ground term of $TS(t)$. Any element of $TS(t)$ is a quasi-instance of R since it is an instance of t. The proof is the same as in theorem 2 for type 3a: if a term t is a quasi-instance of R and if there is no variable x

[4] the case of a signature consisting of one AC-symbol and constants is easily dealt with since all terms are linear

and no $l \in R$ unifiable with t, such that $x \in \|t\|_c$ and such that $\|l\|_c$ is defined, then t must be an instance of some $l \in R$. □

The following example shows why it might be necessary to instantiate so much the initial term:
$t = +(x, y, z)$ is of type 3b and is a quasi-instance of R with $R = \{+(0, y), +(s(0), y), +(s(s(x)), y), +(s(f(x, y)), z)\}$ but the instance $s = +(s(x), y, z)$ of t (which is not in the test-set of t) is not an AC-instance of any rule.

The converse of the theorem requires a linearity condition to hold:

Theorem 4 *Let t be a term of type 3b, let R be a set of* linear *terms, if for all $s \in TS(t)$, there exists $l \in R$ and a substitution σ such that $s =_{AC} l\sigma$, then t is a quasi-instance of R.*

Proof. (sketch) For simplicity we assume that there is only one subterm of t which has more arguments than the corresponding subterm in R. Therefore one may write $t = C[f(s_1, \ldots, s_m)]$ where $f(s_1, \ldots, s_m)$ is this distinguished subterm and C denotes its context. Since the variables in the context do not play any role, we also suppose that this context does not contain variables. Let θ a ground substitution with $Var(t) \subseteq Dom(\theta)$. We will show by induction on $depth(\theta)$ that $t\theta = l\sigma$ for some l and σ.

Let $f(u_1, \ldots, u_m)$ be a ground term obtained from $flat(f(s_1\theta, \ldots, s_m\theta))$ by deleting arguments of AC-functions such that for each position c of an AC-function which is a position of some l in R, $Max(arg(s)|s \in \|t\|_c) \leq Min(arg(t)|t \in \|t\|_c)$. By construction, the term $t' = C[f(u_1, \ldots, u_m)]$ is in $TS(t)$ or is the instance of some term of $TS(t)$, therefore there exist some l and σ such that $t' = l\sigma$. Either σ is such that $x\sigma = C'[f(u_1, \ldots, u_m)]$ with $x \in Var(l)$ or $f(t_1, \ldots, t_p)\sigma =_{AC} f(u_1, \ldots, u_p)$ with $f(t_1, \ldots, t_p)$ a subterm of l with $p < m$.

In the first case, the variable x is linear since l is linear therefore σ' such that $x\sigma' = f(u_1, \ldots, u_m)$ and $y\sigma' = y\sigma$ for $y \neq x$ satisfies $t\theta = l\sigma'$.

In the second case, there is some (linear) variable x such that $x\sigma = f(u_1, \ldots, u_{i_k})$. Let σ' be such that $y\sigma' = y\sigma$ and $x\sigma' = f(u_{i_1}, \ldots, u_{i_k}, v_1, \ldots, v_k)$ where $\{v_1, \ldots, v_k\}$ is the set of arguments of $flat(f(s_1\theta, \ldots, s_m\theta))$ which have been eliminated when constructing $u_1, \ldots, u_m$. The process is repeated for the u_i not equal to some argument of $flat(f(s_1, \ldots, s_m)\theta)$, and finally we get a substitution σ_n such that $t\theta = l\sigma_n$ □

4.3 Refinements for Type 3b

When non-linear patterns are present the previous theorem does not apply to type 3b. We have necessary conditions for completeness, but no necessary and sufficient condition. We give a sufficient condition which is straightforward but useful in practice:

Proposition 2 *Let t be a term of type 3b, if for some variable $x \in Var(t)$, all the elements of $Suc(x, t)$ are AC-instance of the patterns, then t is a quasi-instance of R.*

The proof is straightforward. Unfortunately, this sufficient condition is not necessary, even for linear patterns as shown by the example following the proof of theorem 3. More complex conditions of the same flavor exist which are not given.

Another improvement of our method is to restrict the size of the test-set $TS(t)$ (which may be very large). We construct a smaller but equivalent set as follows:

Let $T_0(t), \ldots, T_n(t), \ldots$ be a sequence of sets defined by

- initialize $T_0(t)$ to $\{t\}$.
- construct $T_n(t)$ from $T_{n-1}(t)$:
  - each ground (flattened) term of $T_{n-1}(t)$ is in $T_n(t)$.
  - each (flattened) term of $T_{n-1}(t)$ which an AC-instance of some pattern, is in $T_n(t)$
  - for all term $s \in T_{n-1}(t)$ which is not ground or not an AC-instance of some pattern, all the elements of $Suc(x, s)$ (where x is some variable of s) which satisfy the condition of the definition of $TS(t)$ are in $T_n(t)$

This sequence is stationary from some n, and we take its last element (which is equivalent to $TS(t)$ with respect to the quasi-instance property) as the effective test-set.

5 Compilation of Pattern Matching

5.1 Pruning Pattern Trees

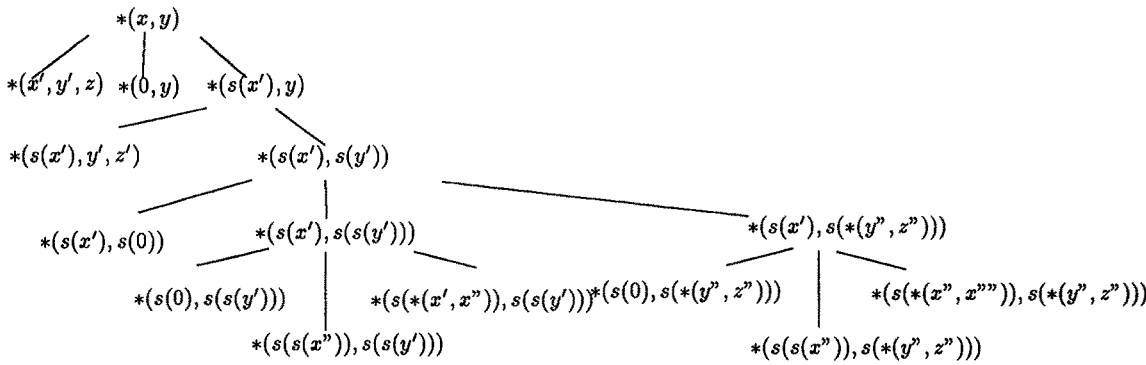
The reader may notice that pattern trees often contain redundant nodes: for example the node $f(x, y)$ produces the nodes $f(x, 0)$ and $f(x, s(y'))$ which produce the nodes $f(0, 0), f(s(x'), 0)$ and $f(0, 0), f(s(x'), s(y'))$), while the set $f(0, 0), f(0, s(z)), f(s(z_1), s(z_2))$ is sufficient. A solution is to detect most of the redundant nodes before generating them: we call two variables x and y *equivalent* in t when there exists a subterm $f(\ldots, x, \ldots, y, \ldots)$ in t with f an AC-function. If a linear term t is extensible in x its sons which are extensible are extensible in y. To avoid redundancy, if $\Sigma = \{c_0, \ldots, c_n\}$, if $t_i = t_{\{x \leftarrow c_i(\overline{z})\}}$ is extensible in y, it is sufficient to extend t in y with the substitutions $\{y \leftarrow c_j(z')\}$ for $i \leq j \leq n$. This prunes (roughly) half of the successors of t. Obviously this may be generalized to more than two variables and to similar situations.

For example, let $\Sigma = \{0, s, +, *\}$ and let the pattern definition be:

$$\begin{cases} *(x, 0) & \rightarrow 0 \\ *(x, s(0)) & \rightarrow x \\ *(s(x), y) & \rightarrow +(*(x, y), x) \end{cases}$$

We will assume that $+$ is completely defined with respect to 0 and s, but we will not assume that 0 and s are declared as constructors [5]. A pruned minimal complete pattern tree for $*$ is:

[5] therefore the pattern tree given is section 2 is no longer relevant



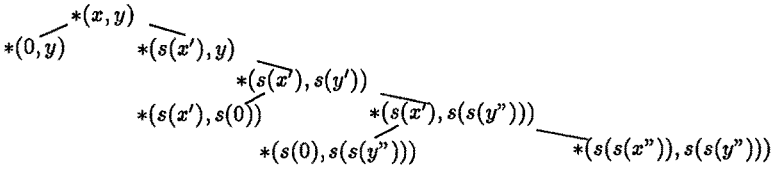
The theorem on terms of type 3b apply, for example the construction of $TS(*(x', y', z))$ gives $*(0, y, z)$ which is an instance of the first rule, $*(s(0), y, z)$ which is an instance of the second rule and $*(s(s(x'')), y, z)$ and $*(s(*(x'', x''')), y, z))$ which are instances of the third rule. Therefore $*(x, y, z)$ is a quasi-instance of the patterns, and the same property holds for the other terms of type 3b.

5.2 The Compilation Algorithm

This section provides the algorithm for the compilation of pattern matching. We recall that $\mathcal{L}$ is a language which evaluates ground term using head-rewriting. The compilation algorithm given in this section is relevant only for such a language. If we want to rewrite non-ground terms, the rules for terms of type 3 must be changed. The first part of the algorithm is described informally and a set of inference rule will describe more precisely the code generation.

- Compute a minimal complete pattern tree for $f(x, y)$
- Prune all nodes of type 3b which can be shown to be a quasi-instance of R and all nodes t such that the sub-tree rooted at t contains only leaves of type 1 or type 3b which are quasi-instance of R. If the pattern tree does not contain any node of type2 or 3a, and if all nodes of type 3b are proved to be quasi instance of R, then the definition of f is complete, and each node where f appears elsewhere than as root symbol can be discarded.

After this step, the pattern tree contains leaves of type 1,2,3a or 3b which cannot be shown to be quasi-instances of the left hand-sides of the rules. Since the definition of $*$ in the multiplication example is complete, the minimal pattern tree of the previous section is simplified into:



The compilation algorithm relies on a function *Compile* which has two arguments: t the functional scheme to compile (initially $f(x,y)$) and R the set of rules which defines it. The result is a function which has one ground term as argument and returns the code to be evaluated. Since it is easier to define a function by giving its value on some argument S, the inference rules of the compilation are defined for expressions like $Compile(t,R)[S]$ i.e the result of $Compile(t,R)$ applied to the dummy variable S. These expressions are conditional ones. When the variable S is instantiated by a ground term with f as top-symbol, the evaluation of $Compile(f(x,y),R)$ provides the same result than a straightforward evaluation of S using the rules of R. The reader must be aware that this algorithm is a general scheme which must be tailored for implementation (some hints are given in section 5.3).

Some definitions are required:

- *match-with*(l,S) (resp *unify-with*) is a function defined by *match-with*(l,S)=true iff $S =_{AC} l\sigma$ (resp. S and l are AC-unifiable).
- if t is a term with variable, $\tilde{t}$ denotes the scheme (or pattern) associated to t, obtained from t by replacing all the variables of t by the dummy symbol "?".
- *has-pattern*$(\tilde{t},S)$ with t a linear term and S a ground term, is true if and only if $S = t\sigma$ where σ is a ground substitution. One may ask why we introduce such a predicate similar to *match-with*: the answer is that they will be implemented in a different way.

The initialization sets $t = f(x,y)$ and $R = l_1 \ldots l_n$ the list of patterns. The compilation algorithm is given by the inference rules:

R_1 Compile(t,$\emptyset$)$[S]$ = *no-match*

R_2 Compile$(t,R)[S]$ = *no-match*
if t is of type 2.

R_3 Compile$(t,l.R)[S]$ = Compile(t,R)[S]
if t and l are not AC-unifiable.

R_4 Compile$(t,l.R)[S]$ = **if** *match*(l,S) **then** *apply-rule*$(l)[S]$
else Compile(t,R)
if t is of type 1, is AC-unifiable with l, but does not match l.

R_5 Compile$(t,l.R)[S]$ = *apply-rule*$(l)[S]$
if t is of type 1 and $t = l\sigma$

R_6 Compile$(t, R)[S]$ = **if** *has-pattern*$(t\tilde{\sigma_1}, S)$ **then** Compile$(t\sigma_1, R)[S]$
elsif *has-pattern*$(t\tilde{\sigma_2}, S)$ **then** Compile$(t\sigma_2, R)[S]$
...
else Compile$(t\sigma_n, R)[S]$

if t is extensible in x and the sons of t are the t_i with $t_i = t\sigma_i$ and $\sigma_i = \{x \leftarrow c_i(x_i)\}$

R_7 Compile$(t, R)[S]$= **if** *unify-with*(l_{i_1}, S) **then** *apply-rule*$(l_{i_1})[S]$
elsif *unify-with*(l_{i_2}, S) **then** *apply-rule*$(l_{i_2})[S]$
...
elsif *unify-with*(l_{i_p}, S) **then** *apply-rule*$(l_{i_p})[S]$
else *no-match*

if t is AC-unifiable with $l_{i_1}, \ldots, l_{i_p}$ (where the numbering i_1 to i_p is compatible with the priority) and t is of type 3a or else t is of type 3b but cannot be shown to be a quasi-instance of R

Let us see how this algorithm behaves on the multiplication example (for simplicity, *apply-rule*$(l_j)[S]$ is shortened into *apply-rulej*$[S]$). From the last pattern tree computed for $*$, the compilation algorithm returns the result[6]:

if *has-pattern*$(*(0,?), S)$ **then** *apply-rule*1$[S]$ **else**
if *has-pattern*$(*(s(?), s(?)), S)$ **then** **if** *has-pattern*$(*(s(?), s(0)), S)$ **then** *apply-rule*2$[S]$
else **if** *has-pattern*$(*(s(0), s(s(?))), S)$ **then** *apply-rule*2$[S]$
else *apply-rule*3$[S]$

The correction of the Compilation algorithm (rewrite a term using the compiled code yields the same result than using the definition of the function and the priority-rule) is a straightforward consequence of the theorems of section 4, and we state:

Theorem 5 *The compilation algorithm is correct.*

Remark: Our compilation algorithm works for flattened terms, and it is possible to design another algorithm working for terms (and not flattened terms).

5.3 Implementation Issues

A straightforward implementation of the compilation algorithm is likely to be unefficient. For example the *has-pattern* function would search through the same term many times with an increasing sequence of scheme such as $*(?,?), *(0,?), *(0, s(?))$ occurs in the ground term S. In the same way, pattern-matching and unification modulo AC should be implemented efficiently with respect to the data-structure chosen for representing terms. One solution of this problem is to use DAG (directed Acyclic Graph) for representing terms, and structure sharing, together with a total ordering on the set of function symbols, as in [GD88]. One may consider also the solutions suggested in [PB85]. A complete description of implementation sollution is out of the scope of the paper, we limit ourselves to show how a total ordering may help.

[6]The reader should compare with code generated without using our algorithm and not proving the completeness of $*$!

After a total ordering $\prec$ is chosen on Σ it is possible to represent a ground term t in a unique way as a binary search-tree. Let us identify such a tree with sequences of symbols separated by "/". These sequences are the sorted sequence of nodes of the tree occurring at the same depth and having the same father. Moreover all sequences corresponding to the same depth are concatenated (with respect to the multiset extension of $\prec$).

For example let $\Sigma = \{0, s, g, f\}$ where 0 is nullary, s is unary and both f and g are AC-symbols, with the ordering $0 \prec s \prec g \prec$. Then the term $S = f(g(s(0), 0), g(0, s(0), s(0)))$ is written: $/f/g\ g/0\ s/0\ 0\ s/0/0/0/$, and it is possible to reconstruct the term (or an AC-equivalent one) from this representation: f is the top-symbol of S, and $/g\ g/$ proves that it has two arguments, both with g as top-symbol. The first such argument has also two arguments with top-symbols 0 and s, and the second one has three arguments with top-symbols 0,s and s, and so on.

Scheme can be represented in the same way, and testing that a ground term S matches some scheme $\tilde{t}$ can be done linearly in the sum of the size of the representation of $\tilde{t}$ and S. Moreover when the pattern tree is generated in a suitable way (a sort of breadth-first construction), the *has-pattern* predicate can be implemented in an incremental way.

6 Conclusion

We have given a algorithm to compile the pattern-matching of AC-function definitions when the evaluation follows a head-rewriting scheme. Although the generated *if ... then ... else* expression can have an exponential size, practical examples usually have a reasonable size (when the definition is complete). Moreover our algorithm is also a decision method for the completeness of left-linear definitions. When the rules are not left-linear, we have necessary conditions for completeness. We have also set sufficient conditions for the non-linear case (for example, we can prove that the definition $+(x, 0) \rightarrow x, +(x, x) \rightarrow double(x), +(s(x), y) \rightarrow s(+(x, y))$ is complete). The main open question is to replace the linearity condition by a weaker one. Our method generalizes to associative functions (which includes the important case of function composition) and is useful also in Logic Programming.

Acknowledgements

We thanks S. Peyton-Jones for his comments on a previous version of this paper, and an anonymous referee for his remarks.

References

[Aug85] L Augustsson. Compiling pattern matching. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture*, volume 201, Nancy (France), 1985. Springer Verlag, Lecture Notes in Computer Science.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1 & 2):203–216, April 1987.

[GD88] B. Gramlich and J. Denzinger. Efficient ac-matching using constraint propagation. Technical Report SR-88-15, SEKI, Universite' de Kaiserslautern, RFA, 1988.

[HL79] G. Huet and J.J. Levy. Call by need computations in non-ambiguous linear term rewriting systems. Research report 359, INRIA, August 1979.

[JK86] J.P. Jouannaud and E. Kounalis. Proof by induction in equational theories without constructors. In *Proceedings 1st Symp. on Logic In Computer Science*, pages 358–366, Boston (USA), 1986.

[KNRZ87] D. Kapur, P. Narendran, D.J. Rosenkrantz, and H. Zhang. Sufficient-completeness, quasi-reducibility and their complexity. Technical report, State University of New York at Albany, 1987.

[Kou90] E. Kounalis. Testing for inductive-(co)-reducibility in rewrite system. In *15th Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages pp175–191. Springer-Verlag, 1990.

[Lav87] A. Laville. Lazy pattern matching in the ML language. In *Proceedings 7th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *Lecture Notes in Computer Science*, pages 400–419. Springer-Verlag, Lecture Notes in Computer Science, December 1987.

[Lav88] A. Laville. Comparison of priority rules in pattern matching and term rewriting. Technical report, INRIA, 1988.

[PB85] P. W. Purdom and C. A. Brown. Fast-many-to-one matching algorithm. In J. P. Jouannaud, editor, *Proc. 1st Conf. Rewriting Techniques and Applications*, pages 407–416. Springer-Verlag, Lecture Notes in Computer Science, 1985.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages.* Prentice Hall, 1987.

[PS90] L. Puel and A. Suarez. Compiling pattern matching by term decomposition. In *Proceedings ACM Conference on LISP and Functional Programming*, 1990.

[Sch88] Ph. Schnoebelen. Refined compilation of pattern matching for functional languages. *Science of Computer Programming*, 11:133–159, 1988.

[SR90] R.C. Sekar and I.V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. In *5th Symp. Logic in Computer Science*, pages 230–241. IEEE, 1990.

[Tre90] Ralf Treinen. A new method for undecidability proofs of first order theories. In K. V. Nori and C. E. Veni Madhavan, editors, *Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 48–62. Springer Lecture Notes in Computer Science, vol. 472, 1990.