A COMPILER FOR
CONDITIONAL TERM REWRITING SYSTEMS

Stéphane Kaplan
L.R.I. U.A. 410 of the C.N.R.S.
Bat. 490. Université des Sciences
F-91405 Orsay Cedex (France)
[e-mail : mcvax!inria!lri!kaplan]

Abstract :

In this paper, we present a *compiler* for conditional term rewriting systems. With respect to traditional interpreters, the gain in execution time that we obtain is of several orders of magnitude. We discuss several optimizations, among which a method to share code in the premises of the conditional rules, well-adapted to algebraic specifications.

INTRODUCTION

During the past decade, term rewriting systems (TRS) have known an increasing number of applications in several fields of computer science concerned with symbolic evaluation. To list but a few different utilizations, TRS are used in a significant number of theorem proving systems, to optimize Prolog-like interpreters where they can enhance the efficiency of the narrowing mechanism, as a prototyping tools in algebraic specification environments where they allow to compute the normal form of a given term, etc...

In all these applications, execution time has become an important factor. Until now and to our knowledge, term rewriting systems have only been executed in an *interpreted* mode (except for the basic specifications that may be directly implemented in an object language under some systems). In spite of a new generation of improved interpreters ([FGJM 85], [CGSA 85], [GCAS 85], [JD 86]), TRS have therefore been considered in general as inherently slow. In this paper, we present a *compiler* for such systems. This compiler produces Lisp code for each operator symbol of the specification, which is in turn compiled for the target machine by a classical Lisp compiler. Computing the normal form of a term then simply consists in evaluating (in the sense of Lisp) a canonical representation of it. The gain of efficiency is *of several orders of magnitude* with respect to the interpreted mode. The time needed to evaluate a given term becomes an acceptable function of its size. This allows to consider algebraic specifications as directly executable, and the whole formalism of the abstract data types as a programming language (cf. e.g. [FGJM 85]).

The compiler described in this paper has been integrated into the ASSPEGIQUE environment for the development of algebraic specifications (cf. [BC 85], [Assp 86]) ; this allows to directly execute specifications under this system.

For related works, the language HOPE ([BMS 80], [Darl 86]), that involves rewriting, has been compiled, though in a sense different from this paper. Emphasis is on the definition of abstract instruction sets and related architectures. Also, our treatment of the pattern-matching process has some similarities with optimizations of the unification step considered in logic programming language compilation (cf. e.g. [Warren 80], [Warren 83], [Cohen 85], [HS 86], [Stickel 86]). However, our compiler, though being still a prototype, turned out to run significantly faster than when compiling directly in Prolog (experiments performed with the Quintus Prolog compiler). The main reason is that we do not manipulate substitutions, but work directly with *functional calls*.

This article is organized as follows. Section 1 presents the formal concepts that underlie our method ; we prove there the correction of our compilation scheme. Section 2 describes the main features of the compiler for non-conditional rules ; some basic optimizations are presented and discussed. In section 3, we describe our treatment of conditional rules, while section 4 shows advanced features supported by the compiler. Several benchmarks are given and discussed in section 6. Lastly, section 7 presents further optimizations.

1. THE FORMAL BACKGROUND

In this section, we present the mathematical background of this article. We suppose that the reader has some knowledge about the programming language Lisp, and about terms and term rewriting systems (cf. [HO 80] for a survey of the definitions and fundamental results in the field). However, basic notions are recalled hereafter.

Σ stands for a finite set of ranked symbols. Σ^n is the set of the symbols of Σ with arity n. X stands for an enumerable set of variables. $T_{S,\Sigma}(X)$ is the set of the terms with variables built on Σ and X. We take for granted the notion of terms, occurrences in terms and substitutions. $t_{|\omega}$ is the subterm of t starting at occurrence ω, and $t[\omega \leftarrow u]$ is t where $t_{|\omega}$ is replaced by u.

A *term rewriting system* is a finite set R of couples (λ,ρ) of terms of $T_{S,\Sigma}(X)$ such that the variables of ρ are included in the variables of λ. The rewriting relation $\rightarrow$ is defined by $t \rightarrow t'$ *iff* there exists $(\lambda,\rho) \in R$, an occurrence ω of t and a substitution σ such that $t_{|\omega} = \lambda\sigma$ and $t'=t[\omega \leftarrow \rho\sigma]$. $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$. R is *finitely terminating* if there exists no infinite sequence $(t_i)_{i \geq 0}$ such that $\forall i\ t_i \rightarrow t_{i+1}$, and R is *confluent iff* $\forall t, t', t''$, if $t \rightarrow^* t'$ and $t \rightarrow^* t''$, then there exists a u such that $t' \rightarrow^* u$ and $t'' \rightarrow^* u$. In the following, all the TRS that we consider are assumed to be finitely terminating and confluent. Each term t thus admits a unique *normal form*, i.e. a term $\overline{t}$ such that $t \rightarrow^* \overline{t}$ and $\overline{t}$ is a normal form (i.e. there exist no t' such that $\overline{t} \rightarrow t'$). Lastly, the rules $\lambda \rightarrow \rho$ of R such that the root

symbol of λ is a given symbol 'f' are called the rules defining f.

The essential question addressed in this paper is how to efficiently *evaluate* a term, i.e. compute its normal form.

Conventions : *underlined* symbols correspond to functions associated to symbols of Σ. For instance, if $f \in \Sigma^2$ and t_1, t_2 are two terms of $T_{S,\Sigma}(X)$, then $\underline{f}(t_1,t_2)$ stands for the result of applying the function $\underline{f}$ (yet to be described) to t_1 and t_2, while $f(t_1,t_2)$ is the term of $T_{S,\Sigma}(X)$ with root symbol f and subterms t_1 and t_2. Also, to simplify our matter, we shall suppose that all the constants (i.e. the elements of Σ^0) are normal forms ; this is of course not compulsory in our actual compiler.

We now provide formal background to the principle of our compiler. We feel that this is useful since the formal correctness of evaluators is rarely proved in the literature, though being not necessarily obvious. To this effect, we define a function $\underline{eval} : T_{S,\Sigma}(X) \rightarrow T_{S,\Sigma}(X)$, and associate with each operator symbol $f \in \Sigma^n$ a function $\underline{f} : (T_{S,\Sigma}(X))^n \rightarrow T_{S,\Sigma}(X)$. They are defined, in a mutually recursive way, by the following abstract programs :

$\underline{eval}(t)$ =
  if $t = f(t_1, \ldots, t_n)$ **then** $\underline{f}(\underline{eval}(t_1), \ldots, \underline{eval}(t_n))$
  else ;; t must be a variable
    t

and :

$\underline{f}(t_1, \ldots, t_n)$ =
;; let $r_j : f(\vec{\mu_j}) \rightarrow \rho_j$, $1 \le j \le n$, be the rules in R defining f
  $[]_{1 \le j \le n}$ **if** there exists a substitution σ such that $(t_1, \ldots, t_n) = \vec{\mu_j}\sigma$ **then** $\underline{eval}(\rho_j\sigma)$
  else $f(t_1, \ldots, t_n)$

We use here a non-deterministic choice operator '[]'. If each test of the choice statement fails, then the 'else' statement is executed. For instance, consider the following TRS (S_1) :

x plus zero $\rightarrow$ x   x plus s(y) $\rightarrow$ s(x plus y)

The corresponding '$\underline{plus}$' function is :

$\underline{plus}(t_1, t_2)$ =
  [] **if** t_2 = zero **then** $\underline{eval}(t_1)$
  [] **if** t_2 = s(y) **then** $\underline{eval}$(s (plus(t_1, y)))
  else plus(t_1,t_2)

It should be noted that, hereabove, 'plus(t_1,t_2)' is the term with root symbol 'plus' and subterms t_1, t_2 ; this corresponds to the case where no rule could apply (i.e. t_2 is neither of the form 'zero' nor of the form 's(y)'). Also, 's' is defined by no rule, $\underline{s}$ is simply defined by $\underline{s}(t_1,t_2) = s(t_1,t_2)$; similarly, under our assumption that constants are irreducible, they are just represented by atoms.

We now have the following result of total correctness :

Theorem 1.1

For any $t \in T_{S,\Sigma}(X)$, any computation of $\underline{eval}(t)$ eventually terminates, producing the normal form of t.

Proof : The ordering

$$t > t' \quad \textit{iff} \quad t \rightarrow t' \text{ or } t' \text{ is a subterm of } t$$

is well-founded because $\rightarrow$ is finitely terminating. Let us denote by P(t) the property "any computation of t terminates, producing the normal form of t". We are going to show that P holds on $T_{S,\Sigma}(X)$ by noetherian induction on >. *Ad absurdum*, suppose that there exists a $t \in T_{S,\Sigma}(X)$ such that $\neg P(t)$. Since > is well-founded, it is possible to choose t extremal w.r.t. > such that $\neg P(t)$, i.e. if $t > t'$ then P(t') (cf. [Huet 77]). Let us then consider a 'faulty' computation of $\underline{\text{eval}}(t)$:

- We can suppose that t is not a variable, i.e. that $t = f(t_1,...,t_n)$. Since $t > t_i$, then $P(t_i)$ for any i. Thus, the computation of each $\underline{\text{eval}}(t_i)$ terminates, yielding $\overline{t_i}$.
- We now consider the next phase of the evaluation, i.e. the computation of $\underline{f}(\overline{t_1}, \ldots, \overline{t_n})$.
  - if the computation chooses a j and a substitution σ such that $(\overline{t_1},...,\overline{t_n}) = \vec{\mu}_j\sigma$, then $t \rightarrow \rho_j\sigma$ and $t > \rho_j\sigma$. Thus, any computation of $\underline{\text{eval}}(\rho_j\sigma)$ eventually stops producing the normal form of $\rho_j\sigma$, which is also the normal form of t. Thus, the corresponding computation of t terminates producing $\overline{t}$, which is contradictory ;
  - else, since the $\overline{t_i}$'s are normal forms, this means that $f(\overline{t_1},...,\overline{t_n})$ is a normal form, and thus the normal form of t. On the other hand, this is exactly what the procedure produces. We derive again a contradiction. ■

2. THE CODE GENERATION PROCESS

The compiler generates Lisp code according to the following ideas :

- the $\underline{\text{eval}}$ function of section 1 is implemented by the usual *eval* function of Lisp
- the code of each $\underline{f}$ function is translated into specific Lisp code.

Thus, the compiler generates in a first phase lambda-definitions for each $\underline{f}$. In a second phase, this code is compiled for the target machine by a usual Lisp compiler.

Terms have their natural Lisp representation : a term $f(t_1,...,t_n)$ is represented by $(f\ t_1 \ldots t_n)$; variables are implemented by atoms, and constants are implemented by by atoms bound to their name in Σ (still under the assumption that constants are irreducible). Since in Lisp, a user defined function (interpreted or compiled lambda) evaluates its arguments before evaluating its body, it is then easy to check that the implementation of $\underline{\text{eval}}$ is correct, w.r.t. its definition in section 1.

For each $\underline{f}$, the code generated by the compiler is a simple translation of its definition in section 1. *The search for a matching substitution σ is coded directly in the Lisp code of each* $\underline{f}$. During this pattern-matching step, any variable x of the lefthand side of the selected rule is locally bound to $x\sigma$. This allows to directly evaluate the righthand side of the rule as the result of the reduction. This method has shown to be much more efficient than a general purpose call of the form "match($\vec{\mu}_j$, $(t_1, \cdots, t_n)$)" (cf. also [Aug 85] where a similar process is described). A related idea is used to implement the unification

step when compiling logic programming languages (cf. [Warren 83], [HS 86]).

For instance, the code corresponding to the function plus of section 1 is :

```
(defun plus (*X1 *X2)
  (let ((x) (y))
      (cond
        ((and (setq x *X1) (eq 'zero *X2))  x)
        ((and (setq x *X1)
              (eq 's (care:car *X2))
              (setq y (cadr *X2)))          (s (plus x y)))
        (t                                 `(plus ,*X1 ,*X2)))))
```

Notes :

- The care:car function is a macro that checks whether its arguments is a cons ; if so, it returns its car, else it returns nil. As a consequence, if "(eq 's (care:car *X2))" evaluates to non-nil, it is known that *X2 admits a cadr.

- The rules defining f appear in the body of f in the same order as they are given in R. However, as opposed to the case of Prolog, the user is invited *not* to make use this information (for the systems are supposed to be confluent).

- In this paper, whenever readability is improved, we have preserved inside the bodies of the f's the variable names that occur in the rules of R (as in the previous example). This is optimized in the compiler that actually generates the following, more efficient though less readable, code :

```
(defun plus (*X1 *X2)
  (cond
    ((eq 'zero *X2)              *X1)
    ((eq 's    (care:car *X2))  (s (plus *X1 (cadr *X2))))
    (t                          `(plus ,*X1 ,*X2))))
```

- Also, as a technical point, we always refer in the paper to the 'eq' Lisp function in order to compare Lisp objects (for sake of clarity). The compiler also generates calls to the less efficient 'equal' function, when needed.

- *Non left-linear rules*, i.e. rules in the lefthand side of which a variable occurs more than once, are treated in a straightforward manner. Consider for instance the one-rule system :

$$f(x, a, x) \rightarrow x \qquad (S_2),$$

the corresponding piece of code would be :

```
((and (eq 'a *X2) (eq *X1 *X3))  *X1)
(t                              `(f ,*X1 ,*X2 ,*X3))
```

Let us consider a slightly more complex example (S_3) :

r_1 : $f(a) \rightarrow a$

r_2 : $f(f(f(x))) \rightarrow b(x)$

r_3 : $f(f(g(x))) \rightarrow c(x)$

The relevant piece of code generated for f is :

```
(let ((x nil) (*X1.1 nil)) (cond
  ((eq 'a *X1)                                                   a      )
  ((and (eq 'f (care:car *X1))    (setq *X1.1 (cadr *X1   ))
        (eq 'f (care:car *X1.1)) (setq x      (cadr *X1.1)))   (b    x))
  ((and (eq 'f (care:car *X1))    (setq *X1.1 (cadr *X1   ))
        (eq 'g (care:car *X1.1)) (setq x      (cadr *X1.1)))   (c    x))
  (t                                                           '(f ,*X1))))
```

The auxiliary variable *X1.1, generated at compile time, helps avoiding the whole computation of (cadadr *X1), remembering that (cadr *X1) is stored in *X1.1. This is particularly time-saving in the case of non-unary symbols. Another use of such variable in more advanced code optimization, allowing the factorization of the tests in the previous function body, is discussed in section 4.

3. CONDITIONAL TERM REWRITING SYSTEMS

In this section, we present the treatment of *conditional* rewrite rules by the compiler. Such kind of rules greatly extend the expressive power of classical term rewriting systems (though they do not extend their mathematical power. Cf. [BT 86]). Moreover, they arise very naturally when using term rewriting systems as the operational counterpart of algebraic specifications (cf. [Aff 81], [FGJM 85], [GK 86]).

By definition, a conditional term rewriting system is a finite set R of formulae :

$$r : u_1 = v_1 \;\&\; \dots \;\&\; u_n = v_n \;\Rightarrow\; \lambda \to \rho, \quad \text{with } Var(\lambda) \supseteq Var(\rho) \cup (\cup_{i=1}^{n} Var(u_i) \cup Var(v_i)).$$

The rewriting relation is defined (cf. e.g. [RZ 85], [Kaplan 84,86]) as the smallest relation $\to$ such that $t \to t'$ *iff* there exists a rule $r \in R$, an occurrence ω of t and a substitution σ such that

- $t_{|\omega} = \lambda\sigma$ and $t' = t[\omega \leftarrow \rho\sigma]$
- for any i, there exists a $\gamma_i \in T_{S,\Sigma}(X)$ such that $u_i\sigma \to^* \gamma_i$ and $v_i\sigma \to^* \gamma_i$.

$\to^*$ stands for the reflexive and transitive closure of $\to$.

Notions of confluence, termination and normal forms are defined as in section 1. Similarly, one can extend the definitions of the $\underline{f}$'s, simply checking recursively that for each i, the normal forms of u_i and v_i are equal. The generic form of the Lisp code of an $\underline{f}$ is :

```
(cond
  ;; treatment of rule r : u1 = v1 & ... & un = vn   =>   f(μ) -> ρ
  ((and
      (<does μ match the arguments of f ?>)        ;; this binds the variables of ρ
      (eq u1 v1) ... (eq un vn))                   ;; this checks for the premises
                                                     ρ)
  ;; treatment of the next rule, etc ...
```

During the evaluation of the (eq u_i v_i)'s, and possibly during the evaluation of ρ, a variable x is bound to its value $x\sigma$, σ being the matching substitution. It is thus the expressions $u_i\sigma$, $v_i\sigma$ and $\rho\sigma$ that are actually evaluated in the body of $\underline{f}$.

EXAMPLE : Consider the system (S_4) :

r_1 : $\qquad f(a) \rightarrow a$

$r_2 : f(e_1(x),a,e_1(x)) = g(e_2(x))$ & $h(i(e_2(x)),e_4(e_3(x))) = j(e_4(e_3(x))) \Rightarrow f(g(x)) \rightarrow R(e_1(x),e_2(x))$

$r_3 : k(e_4(x)) = c \qquad \Rightarrow f(g(x)) \rightarrow R(e_3(x),e_4(e_3(x)))$

The code associated to f would simply be :

```
(defun f (*X1) (let ((x))
  (cond
    ((eq *X1 'a)                                                    'a)
    ((and (eq 'g (care:car *X1)) (setq x (cadr *X1))
          (equal (f (e1 x) a (e1 x)) (g (e2 x)))
          (equal (h (i (e2 x)) (e4 (e3 x))) (j (e4 (e3 x)))))) (R (e1 x) (e2 x)))
    ((and (eq 'g (care:car *X1)) (setq x (cadr *X1))
          (eq    (k (e4 x)) 'c))                                    (R (e3 x) (e4 (e3 x))))
    (t                                                              `(f ,x)))))
```

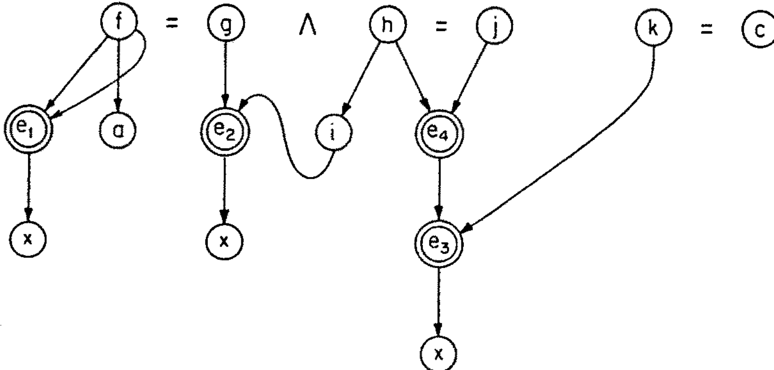
Note that in this example, the search of a matching substitution :
"(eq 'g (care:car *X1)) (setq x (cadr *X1)))" may be factorized, because the corresponding two rules happen to have the same lefthand side. Then, after such a factorization, we remark that some identical computations are performed several times. To improve on this aspect, we introduce auxiliary variables to store interesting partial results. This optimization is particularly important since rules of the form :

$P(x) = \text{True} \quad \Rightarrow f(x) \rightarrow \rho_1(x)$

$P(x) = \text{False} \quad \Rightarrow f(x) \rightarrow \rho_2(x)$

arise very frequently. Such rules constitute indeed the usual way to simulate if-then-else definitions of operators (direct if-then-else rewriting would use so-called "negative" rules, that have no well-defined semantics. Cf. [Kaplan 84]). In such a case, with the previous naive code generation scheme, P(x) would be evaluated twice at run-time.

In order to detect common sub-expressions in the premises, the compiler firstly searches for the rules that have a common lefthand side up to a variable renaming. It then incrementally creates the directed acyclic graph (DAG) of all the expressions occurring in the premises of these rules, in such a way that two common sub-expressions are pointing toward a same term. For instance, the DAG corresponding to rules r_2 and r_3 of example (S_4) is :



Notice that only sub-expressions that are not reduced to variables are shared.

When the DAG is completed, the sub-expressions shared by several terms correspond precisely to the nodes that have strictly more than one ancestor. A local variable *Yi is attached to each such subexpression. The code corresponding to the two last rules then becomes :

```
(let* ((*Y1 (e1 x)) (*Y2 (e2 x)) (*Y3 (e3 x)) (*Y4 (e4 *Y3)))
      (cond
        ((and (equal (f *Y1 'a *Y1)  (g *Y2))
              (equal (h (i *Y2) *Y4) (j *Y4)))      (R *Y1 *Y2))
        (     (eq    (k *Y3)         'c)            (R *Y3 *Y4))
        (t nil))))                                           ;; etc ...
```

A further enhancement is still possible : suppose that the first test "$f(e_1(x),a,e_1(x)) = g(e_2(x))$" failed. It was useless, and perhaps expensive, to evaluate *Y4 = (e4 *Y3). The value of *Y4 would actually be needed, in (R *Y3 *Y4), only if the test "$k(e_3(x)) = c$" did succeed. This led us to generate *demand-driven*, or *lazy*, evaluation of the *Yi's. The code actually generated by the compiler is finally :

```
(let ((*Y1) (*Y2) (*Y3) (*Y4))
     (cond
       ((and (equal (f (setq *Y1 (e1 x)) a *Y1)  (g (setq *Y2 (e2 x))))
             (equal (h (i *Y2) (setq *Y4 (e4 (setq *Y3 (e3 x))))) (j *Y4)))
          (R *Y1 *Y2))
       (     (eq (k (or *Y3 (setq *Y3 (e3 x))))  'c)
          (R *Y3 (or *Y4 (e4 *Y3))))
       (t  nil)))                                                     ;; etc ...
```

The reader is invited to check that expressions are evaluated, or bound, only if necessary. For instance :

- in the test "$k(e_3(x)) = c$", *Y3 is bound if and only if "$f(e_1(x),a,e_1(x)) = g(e_2(x))$" succeeded and "$h(i(e_2(x)),e_4(e_3(x))) = j(e_4(e_3(x)))$" failed. Thus, *Y3 may or may not be bound, which justifies the code "(or *Y3 (setq *Y3 (e3 x)))" ;
- in the evaluation of the righthand side $R(e_3(x),e_4(e_3(x)))$, *Y3 is certainly bound to $e_3(x)$, because of the previous evaluation of the expression "(or *Y3 (setq *Y3 ...))" ; this is actually why it was useful to bind *Y3. On the contrary, *Y4 may or may not be bound. Hence the code "(R *Y3 (or *Y4 (e4 *Y3)))". Of course, it is useless to bind here *Y4 to (e4 *Y3).

It is not difficult to determine which *Yi's are already bound or should be bound : at a given point, the variables that are surely bound are the ones that occur in the *first* premise of all the rules tested before the current one (for the first test "(eq u_1 v_1)" is systematically performed), and the ones occurring in the u_i's and the v_i's of the current rule that have already been checked (i.e. of index i smaller than the current one).

This code optimization has enabled us to greatly improve the performance of the compiler, specially in the case of "if-then-else"-like rules that arise very frequently in algebraic specifications.

4. OTHERS FEATURES OF THE COMPILER

4.1. Evaluation strategy

The user of term rewriting systems may wish to gain control over the evaluation strategy. Consider, for instance, the following specification (that simulates an "if-then-else" operator) (S_5) :

K true x y $\rightarrow$ x

K false x y $\rightarrow$ y

A good evaluation strategy for this operator is to evaluate its first argument, and then to apply a reduction at the top of the expression. This may be done for instance in the OBJ2 system ([FGJM 85]), where the user may stipulate "E-strategies" ; in this case, the operator would be assigned a "(1 0)" profile.

With our approach, we have been so far constrained to a "call-by-value" strategy, because of the underlying evaluation strategy of Lisp. However, it is possible to avoid the computation of a term in Lisp simply by quoting it. In our system, the user can ascribe to 'K' the profile '(t nil nil)', which means that :

- the first argument must be systematically evaluated before application of the body of the operator ;
- the two other arguments are evaluated only if necessary, that is if no reduction could be applied so far. This is a kind of *lazy* evaluation.

Notice that a profile '(t ... t)' corresponds to a call-by-value strategy, and a profile '(nil ... nil)' to a call-by-name strategy. To implement this feature, the internal form a of term 'K(b,x,y)' will be "(K b (quote x) (quote y))". So when the Lisp function K is called, it applies its body to its arguments after evaluation, i.e. to <(eval b),x,y>. The corresponding code is :

```
(defun K (*X1 *X2 *X3)
  (cond
  ((eq *X1 'true) (eval *X2)) ((eq *X1 'false) (eval *X3))
;; if the two previous tests have failed, this means that the normal form of
;; *X1, after reduction, is neither 'true' or 'false'. Evaluate *X2 and *X3,
;; and search for new reductions :
    ((progn (setq *X2 (eval *X2)) (setq *X3 (eval *X3)) nil)
    ((eq *X1 'true) *X2) ((eq *X1 'false) *X3)
    (t  `(K ,*X1 ,*X2 ,*X3))))
```

Actually, it is often the case (as happens here) that if the pattern matching step failed *before* evaluation, it will fail *after* evaluation ; this is because if the user estimates that some variables need not be evaluated, the pattern matching process is likely not to rely on their form. In this case, the second set of tests would simply be omitted. The code of K will thus actually be :

```
(defun K (*X1 *X2 *X3)
  (cond
  ((eq *X1 'true) (eval *X2)) ((eq *X1 'false) (eval *X3))
    ((progn (setq *X2 (eval *X2)) (setq *X3 (eval *X3)) nil)
    (t  `(K ,*X1 ,*X2 ,*X3)))))
```

As a last point, it should be noted that the quoting of the arguments is ensured dynamically whenever needed. For instance, with the previous K and the new rule :

$$K(K(b,b_1,b_2) , y , z) \rightarrow K(b , K(b_1,y,z) , K(b_2,y,z))$$

the corresponding code for the *righthand* side would be :

```
(K
        (K b  '(quote ,b1)  '(quote ,b2))
  '(quote ,(K b1 '(quote ,*X2) '(quote ,*X3))
  '(quote ,(K b2 '(quote ,*X2) '(quote ,*X3)))
```

This feature allows a reasonably large flexibility for the evaluation strategy. It also appears that a large number of operators are prone to benefit of such strategies. For instance, the operator 'plus' defined by (S_1) would gain in having profile '(nil t)' (cf. [FGJM 85] for a discussion).

Note : several other solutions in order to control the evaluation strategy have been discarded. For instance, *macros* could not be used because of recursively defined operators, *lexpr* are not defined in most Lisp dialects, etc...

4.2. Built-in Operators

Since each operator is implemented as a Lisp function (and each irreducible constant by a Lisp constant), it is easy to substitute the function generated by the compiler by a *built-in* Lisp function, that would implement things more efficiently. For instance, in order to implement basic arithmetics and to circumvent (S_1), one could declare :

```
(setq zero 0)   (defun s (*X1) (1+ *X1))   (defun plus (*X1 *X2) (+ *X1 *X2))
```

etc, ... Typically, in large scale applications, this is going to be done systematically for *basic* data types such as integers, booleans, lists, ... with their usual functionalities.

These operators do interact with the code produced by our compiler, since both categories are Lisp objects. For instance, the result of evaluating "(K true (s (plus (s zero) (s zero))) zero)" is "3". It should be noted that this result cannot match a term s(x), under this form. It is the responsibility of the user to take care that such situations do not occur. If so, several solutions might however be considered :

- introducing a parsing phase, that would transform "3" into s(s(s(zero)))
- introducing inverse (Lisp) functions for each symbol that receives a built-in implementation. Then pattern matching becomes possible again, by successive inversion of the symbols in the pattern.

This issue still has to be explored.

Built-in definitions and definitions generated by the compiler will then be "compiled" together by a Lisp compiler, resulting in still improved efficiency. Built-in definitions would even often be *macros* (cf. section 6 and appendix).

The results produced in this fashion are impressive, compared with an interpreted, no built-in mode. It becomes possible to compute expressions that would require an unrealistic evaluation time with

traditional interpreters (cf. section 5). The conclusion of the experiments we made is that, under the assumption that basic types receive built-in interpretation (which is done once for all), high-level, algebraic specifications become executable in a realistic way.

5. QUANTITATIVE EVALUATION OF THE COMPILER

In this section, we compare the performance of a classical interpreter (the evaluator of the ASSPEGIQUE environment [BC 85], [Assp 86]), and of the compiler described in this article.

To this effect, two sets of terms have been successively compared :

- terms of the form "factorial(n)", with the usual unary representation of the integers : n = s(s(...(s(0))...))
- terms of the form "sort([n,n-1,...,1,0])", where 'sort' is a specification of a sort by insertion in increasing order.

Both set of rules are given in appendix. The purpose of these examples is to test both for the non-conditional and for the conditional case. The benchmarks have been made on a VAX750 with operating system UNIX®4.2. Both the interpreter and the compiler are written in Franz Lisp Opus 38.79. The Lisp compiler used in order to compile the interpreter, and the f's produced by our compiler, is Liszt Opus 8.36. The results are as follows (CPU time is in seconds) :

factorial(n) :

	n = 1	n = 2	n = 3	n = 4
T_{interp}	0.65	1.88	6.37	36.85
T_{comp}	0.008	0.017	0.022	0.067
Ratio $\frac{T_{interp}}{T_{comp}}$	78.4	113.0	286.8	552.7

sort([n,n-1,...,1,0]) :

	n = 1	n = 2	n = 3	n = 4
T_{interp}	2.82	5.39	14.55	30.47
T_{comp}	0.024	0.030	0.042	0.067
Ratio $\frac{T_{interp}}{T_{comp}}$	115.2	179.7	349.2	457.0

These figures reveal an impressive gain obtained by the compiler. It also seems that the larger the term

and the number of rewriting steps, the larger the gain.

Being that the above comparison was relative, the compiler considered by itself leads to reasonably short computation times for both specifications. This was not so, to our knowledge, with any of the existing interpreters. Instead, because of their inherent slowness, it appears that TRS interpreters were mainly used as *prototyping tools* : a user could perform some experiments on small and not necessarily relevant terms, in order to check the adequacy of the TRS he wrote w.r.t. what he had in mind. It was by no mean question on *computing* with such systems. With a compiler, this whole picture is modified ; it becomes possible to compute expressions that would have required an unrealistic evaluation time with traditional interpreters. We believe that it becomes reasonable to use term rewriting systems as programming languages, and, consequently, to make algebraic specifications directly executable by rewriting.

This is even more obvious when we use evaluation with built-in operators as described in section 4. In the next benchmark, we give figures obtained with terms 'sort([n,n-1,...,1,0])', where :

- arithmetics (i.e. constants '0', 'T', 'F' and functions 's' and '≤') is implemented directly in Lisp ;
- what remains (i.e. the implementation of the lists, and the functions 'sort' and 'insert') is implemented by rewriting. Note that in sorting a list [n,n-1,...,1,0], a large amount of rewriting is still involved.

We obtained the following results :

	n = 8	n = 9	n = 10	n = 11
T_{comp}	0.11	0.14	0.18	0.24

Thus, in spite of the conditional rewriting involved in the previous benchmark, the execution time remains reasonable. Also, from a relative point of view, the evaluation of the previous terms is clearly out of scope of classical interpreters. This leads to the conclusion that conditional algebraic specifications may be considered as directly executable after compilation, provided a restricted number of basic types are directly implemented in Lisp (essentially the integers and the booleans). A compiler does allow to consider algebraic specifications as a realistic programming language, somehow as Prolog allows to view logic as a programming language.

6. OTHER OPTIMIZATIONS In this section, we describe some important optimizations not discussed so far. Some of them are already partially implemented in experimental versions of the compiler, while some others will be the subject of future researches.

- A first point is to re-consider the role of the partial variables *Xi.j.k... generated by the compiler (cf. section 2). As mentioned, they already serve, inside a given pattern matching test, to avoid useless

computations. However, they may also be used from a test to a next one. For example, with system (S_3), when evaluating the *third* test :

```
((and (eq 'f (care:car *X1))    (setq *X1.1 (cadr *X1  ))
      (eq 'g (care:car *X1.1)) (setq x     (cadr *X1.1)))   (c x)),
```

*X1.1 might already be bound to '(cadr *X1)' due to the first "and-test" (but this is not sure). A first improvement is therefore to replace "(setq *X1.1 (cadr *X1))"
by "(or *X1.1 (setq *X1.1 (cadr *X1)))". This optimization has been implemented on an experimental version of the compiler. It showed no noticeable improvement, partly because testing whether *X1 is nil is not significantly faster than accessing cadr's, on a classical non-tagged architecture.

In this case, it is possible to enhance the code still further : we notice that, actually, *X1.1 is bound by the first "and-test" if and only if *X1 is of the form 'f(z)', which the second test does check. Thus, this test could indeed be reduced to :

```
((and *X1.1
      (eq 'g (care:car *X1.1)) (setq x (cadr *X1.1))))    ;; etc ...
```

This situation seems to occur rather often in practice ; further research is needed to improve on this aspect of the code.

• It is possible to perform, during the compilation, a *flow analysis* of the reductions. A particular case is to detect *tail-recursive* rules, such as :

x plus zero → x

x plus s(y) → s(x) plus y .

We notice that the righthand sides of the rules occur in terminal position in the code generated by the compiler :

```
(defun plus (*X1 *X2)
  (cond                                 ;; ...
    (<does (*X1,*X2) match (x,s(y)) ?>  ;; this binds x and y
     (plus (s x) y))                    ;; ...
```

Thus, the tail recursion would be detected by a good Lisp compiler when compiling the previous definition of 'plus'. This is actually what happens in Franz Lisp. However, we feel that we should not rely on this, for portability reasons and because new methods to detect more complicated cases often come up, which a given Lisp compiler do not necessarily incorporate.

Other flow analysis methods may be considered. Several methods developed for classical programming languages often prove useful (cf. e.g. [Hecht 77], [MJ 81], [AU 77]). Among the works dealing with flow analysis of term rewriting programs, [Jones 86] considers the transformation of a certain class of TRS into context-free tree grammars, with application to partial evaluation and optimization. Also, [FGJM 85] discusses the fact that it is often possible to predict an interesting "evaluation profile" for some operators, and to foretell which rule is more likely to be applied after a given one (a success probability of 2/3 is announced, which is remarkable).

While "evaluation profiles" already exist in our context (section 4), one can ask what would be the meaning of one rule calling for another one, since our evaluation is not rule-driven. A possible answer is to generate a specific code $\underline{f}^{[r]}$ for each rule r defining f (i.e. with f as root symbol of the lefthand side). $\underline{f}^{[r]}$ is similar to $\underline{f}$, except that the test corresponding to the pattern-matching for rule r appears in first position in $\underline{f}^{[r]}$. This assigns higher priority to the application of a given rule. This point will be the subject of future research, from both a theoretical and a practical point of view.

• Consider the rules :

$x \times 0 \rightarrow 0 \qquad x \times s(y) \rightarrow x + (x \times y)$

The code associated to the '×' function is :

```
(defun × (*X1 *X2)
  (cond
    ((...) (...))
    ((...) (+ x (× x y)))
    (t (...))))
```

The call "(+ x (× x y))" will occur from within the external call '(× *X1 *X2)', which is actually not necessary. At run time, this results in useless and numerous pushes on the stack of the interpreter, that also have to be popped finally. Actually, one would like to have the call to "(× *X1 *X2)" *replaced* in the Lisp stack by the call to "(+ x (× x y))". This might be simulated by catches and throws in standard Lisp, but would certainly not enhance the performance !
Instead, we are experimenting with the LAL dialect of Lisp (developped by P. Amar), that allows for a special kind of 'funcall' that is not pushed on the stack (and needs of course not be popped). First benchmarks demonstrate an important gain using this method.

• A last point is that it is useful to simplify at compile time the *terms* appearing in the bodies of the $\underline{f}$'s, typically within the pieces of code corresponding to the righthand sides of the rules. For instance, since there is no rule defining the symbol 's', the identity $\overline{s(s(f(x,s(y))))} = s(s(\ \overline{f(x,s(y))}\ ))$ holds. This allows to replace an expression "((*<match ?>* (s (s (f x (s y)))))" by the more efficient form "(*<match ?>* '(s (s ,(f x '(s ,y))))". This is realized in our compiler by defining functions such as '$\underline{s}$' in the following fashion :

```
(defmacro s (*X1) ``(s ,,*X1))
```

The macro expansion of "(s (s (f x (s y))))" is then identical to " '(s (s ,(f x '(s ,y)))) ", as wanted. This optimization has already resulted in an important gain, specially when applying a Lisp compiler (that displaces macros) on the code produced by our compiler.

Further simplification of the terms appearing in the $\underline{f}$'s may be achieved if a prototype of an evaluator (typically, an interpreter) for the term rewriting system under consideration is available at compile time. This allows for instance to directly produce the normal forms of the righthand sides of the rules inside the code (this corresponds to the classical *constant folding* optimization). Thus, instead of generating the expression "((*<match ?>*) (factorial zero))", one would generate "((*<match ?>*) '(s zero))". Also, a

non-normalized righthand side such as "(factorial (s x))" would be replaced by "(times '(s ,x) (factorial x))"

Note that in order to avoid infinite loop during such compile-time normalizations (in the case of non-terminating TRS), the evaluation should be bounded to some given depth.

This feature will be implemented in future versions of our compiler, using as compile-time evaluator the interpreter (described in [BC 85], [Assp 86]) of the ASSPEGIQUE environment.

7. CONCLUSION

The experiments led so far with our compiler in the ASSPEGIQUE environment reveal an impressive gain with respect to traditional interpreters. The performances are still improved by the use of *built-in* operators. Some optimizations happened to greatly enhance the resulting gain. Among them, sharing code among the premises of conditional rules is particularly well-suited to the natural form of algebraic specifications.

So far, term rewriting systems executed in interpreted way have been used mainly as prototyping tools, with which it was hardly possible to *compute*. Compiling TRS tend to modify this situation, allowing to regard the high-level, mathematical language of algebraic specifications as a real programming language.

This work has been partially supported by the INRIA, the Agence de l'Informatique, and the METEOR Esprit Project. It has been realized while the author was visiting the Weizmann Institute (Rehovot - ISRAEL). Discussions with M.Bidoit, and J.-P.Jouannaud have been stimulating. I thank C.Choppy and F.Voisin for very helpful remarks and suggestions. The integration of the compiler to the ASSPEGIQUE systems has been performed with the help of F.Capy.

REFERENCES :

[Aff 81] R.Erickson, D.Thompson (eds.), *AFFIRM Reference Manual*, USC-ISI, Marina Del Rey (California), 1981.

[Assp 86] M.Bidoit, F.Capy, C.Choppy, N.Choquet, C.Gresse, S.Kaplan, F.Schlienger, F.Voisin, *ASSPRO : un environement de programmation interactif et intégré*, to appear in T.S.I., 1986.

[AU 77] J.Aho, D.Ullmann, *Principles of Compiler Design*, Addison Wesley Publishing Co., 1977.

[Aug 85] L. Augustsson, *Compiling Pattern Matching*, Proc. of the Conf. on Functional Programming Languages and Computer Architectures, LNCS 201 (1985).

[BT 86] J.Bergstra, J.Tucker, *Algebraic Specifications of Computable and Semi-Computable Data Types*, Report CS-R8619, CWI, Amsterdam (1986).

[BC 85] M.Bidoit, C.Choppy, *Asspegique : an Integrated Environment for Algebraic Specifications*, Proc. of the TAPSOFT'85 Conf., LNCS 186 (1985).

[BMS 80] R.Burstall, D.MacQueen, D.Sanella, *HOPE : an Experimental Applicative Language*, Conf. Record of the 1980 Lisp Conf., 1980.

[Cohen 85] J.Cohen, *Describing Prolog by its Interpretation and its Compilation*, CACM **28**, 12, September 1985.

[CGSA 85] D.Coleman, R.Gallimore, V.Stavridiou, F.Ali, *the Design of a Rewrite Rule Interpreter for UMIST OBJ*, UMIST Internal Report (Manchester), 1985.

[Darl 86] J.Darlington, *the Unification of Logic and Functional Languages*, in Logic Programming : Relations, Functions and Equations, DeGroot and Lindstrom Eds., Prentice Hall, 1986.

[FGJM 85] K.Futatsugi, J.Goguen, J.-P.Jouannaud, J.Meseguer, *Principles of OBJ2*, Proc. of the 12th ACM POPL Conf., New-Orleans, 1985.

[GCAS 85] R.Gallimore, D.Coleman, F.Ali, V.Stavridiou, *UMIST OBJ : A Language for Executable Specifications*, UMIST Internal Report (Manchester), 1985.

[GK 86] M.-C.Gaudel, S.Kaplan, *How to Build Meaningful Algebraic Specifications*, Esprit Meteor Project Report, University of Paris-South, 1986.

[Hecht 77] M.Hecht, *Flow Analysis of Computer Programs*, North-Holland Publishing Company, 1977.

[HO 80] G.Huet, D.C.Oppen, *Equations and Rewrite Rules : a Survey*, in Formal Languages : Perspective and Open Problems, R. Book Ed., Academic Press, 1980.

[HS 86] A.Houri, E.Shapiro, *A Sequential Abstract Machine for Flat Concurrent Prolog*, Technical Report CS86-20, Weizmann Institute (Israel), 1986.

[Huet 77] G.Huet, *Confluent Reductions : Abstract Properties and Applications to Term Rewriting Systems*, Proc. of the 18th FOCS Symposium, 1977.

[JD 86] A.Josephson, N.Dershowitz, *Efficient Implementations of Narrowing : the RITE Way*, Internal Report, University of Illinois, 1986.

[Jones 86] N.Jones, *Flow Analysis of Lazy Higher-Order Functional Programs*, Internal Report, University of Copenhagen, 1986.

[Kaplan 84] S.Kaplan, *Conditional Rewrite Rules*, in Theoretical Computer Science 33, 1984.

[Kaplan 86] S.Kaplan, *Simplifying Conditional Term Rewriting Systems*, Internal Report, Weizmann Institute (Israel), 1986.

[MJ 86] A.Mycroft, N.Jones, *A Relational Approach to Program Flow Analysis*, in *Programs as Data Objects*, LNCS 217, 1985.

[Stickel 86] M.Stickel, *A Prolog Technology Theorem Prover : Implementation by an Extended Prolog Compiler*, Proc. of the 8th CAD Conference, LNCS 230, 1986.

[RZ 85] J.-L.Rémy, H.Zhang, *Contextual Rewriting*, Proc. of the 1st Conf. on Rewriting Techniques and Applications, LNCS 202, 1985.

[Warren 80] D.Warren, *Logic Programming and Computer Writing*, in Software Practice and Experience **10**, 1980.

[Warren 83] D.Warren, *An Abstract Prolog Instruction Set*, A.I. Technical Report 309, SRI INternational, 1983.

APPENDIX

We present here the term rewriting systems used in the bench-marks of section 5.

• The first specification defines the integer with operators s, plus, times and factorial. This is a non-conditional specification.

x plus zero	→ x
x plus s(y)	→ s(x plus y)
x times zero	→ zero
x times s(y)	→ x plus (x times y)
factorial zero	→ s(zero)
factorial s(x)	→ s(x) times factorial(x)

• The second specification defines a sort by insertion. List are constructed via the constructors "empty : → list" and "add : integer×list → list".

sort(empty)	→ empty	
sort(add(x,l))	→ insert(x,l)	
insert(x,empty)	→ add(x,empty)	
$(x \leq y)$ = true	⇒	insert(x,add(y,l)) → add(x,add(y,l))
$(x \leq y)$ = false	⇒	insert(x,add(y,l)) → add(y,insert(x,l))
$0 \leq x$	→ true	
$s(x) \leq 0$	→ false	
$s(x) \leq s(y)$	→ $x \leq y$	

• In the last benchmark of section 5, the declarations of built-in operators are as follows :

```
(setq zero 0)
(defmacro s     (x)   `(1+ ,x))
(defmacro plus  (x y) `(+ ,x ,y))
(defmacro times (x y) `(* ,x ,y))
(defmacro ≤     (x y) `(or (eq ,x ,y) (lessp ,x ,y)))
```

The 'insert' and 'sort' functions are still implemented by rewriting.