Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

ARM abstract rewriting machine

J.F.Th. Kamperman and H.R. Walters

Computer Science/Department of Software Technology

CS-R9330 1993

Report CS-R9330
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

ARM
Abstract Rewriting Machine

J.F.Th.Kamperman (jasper@cwi.nl)
H.R. Walters (pum@cwi.nl)

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

Term rewriting is frequently used as implementation technique for algebraic specifications. In this paper we present the abstract term rewriting machine (ARM), which has an extremely compact instruction set and imposes no restrictions on the implemented TRSs. Apart from standard conditional term rewriting, associative lists are supported. ARM code is translated to (ANSI) C; the resulting execution speeds are good (on a sun4, an average of 80000 rewriting steps per second and a maximum of 416000 r/s were measured). Several benchmarks are shown, and related work is discussed in depth.

1991 CR Categories: D.3.4 `[Programming languages]`: Processors – *Code generation, Compilers, Interpreters, Optimization*, D.1.1 `[Programming Techniques]`: Applicative (Functional) Programming, D.1.6: Logic Programming.

1991 Mathematics Subject Classification: 68N20 `[Software]`: Compilers and generators, 68Q05: Models of Computation, 68Q42: Rewriting Systems and 68Q65: Algebraic specification.

Keywords & Phrases: abstract machine, term rewriting, benchmarking, algebraic specification, program generator, C.

Note: Partial support received from the European Communities under `ESPRIT` projects 5399 (Compiler Generation for Parallel Machines – `COMPARE`) and 2177 (Generation of Interactive Programming Environments II – `GIPE II`).

1 Introduction

Algebraic specifications can be used for the description and rapid prototyping of software systems when maintainability and reusability are key issues [BHK89, Hus88]. Term rewriting is frequently used as implementation technique for algebraic specifications. In this paper we present the abstract term rewriting machine (ARM), which has an extremely compact instruction set and imposes no restrictions on the implemented TRSs.

The two major problems of term rewriting implementations — low execution speed and reliance on supporting systems — are being approached in many different ways (a discussion of work related to ours appears in section 6). There are few ideas underlying ARM which do not appear elsewhere in some form or another. This is not to say that ARM was developed as a cocktail of existing ideas; most ideas occurred independently of the indicated references. Yet, our combination of factors results in a model which is elegant, concise and efficient. ARM has 4 instructions (6, when counting variants), without any additional built-in functionality. To our knowledge, the latter is a rare property.

ARM supports arbitrary TRSs, with, in addition, associative lists. There are no restricions (like left linearity, orthogonality) on the TRSs. ARM code is translated to portable (ANSI) C.

The resulting code performs in the order of 80000 rewr/sec for an average, non-trivial TRS. As will be argued in the sequel, this is fairly efficient, considering that few optimizations have been implemented in our system (yet).

In this article we will present the machine model of ARM, and describe its four instructions. In section 5, we will show some benchmarks, relating the exact timings of our implementation to timings of comparable Prolog, ML and C programs. Finally we will give an in-depth discussion of related research projects and we will formulate some conclusions.

2 ARM, the Abstract Rewriting Machine

Rewriting is an iterative process, which repeatedly attempts to recognize an instantiation of the left-hand side of a rule. Then, possibly after checking conditions (which involves a recursive invocation of the rewriting process), the established redex is replaced by an instantiation of the right-hand side.

In accordance, ARM performs a cyclic process in which it is guided by an ARM program. The programming language ARM has control structures for analyzing terms, recursively calling the rewrite process, and generating new terms. These control structures use and manipulate the machine's registers, stacks and heap. There are no instructions to manipulate these memories explicitly, nor can arithmetic or logic computations be performed, because ARM only concerns rewriting.

The cyclic process can be described as follows. Initially all function symbols in a subject term are pushed in pre-order on a stack (called the control stack), which for that moment may be regarded as the input stack. In general, no tree structures will be made by ARM for terms which have not yet been processed. Consequently, an actual tree representation is made only if a term is a normal form.

Each cycle a symbol is taken from the control stack and is interpreted as the outermost function symbol of a to-be-rewritten term. The immediate sub-terms of this term are found (that is, if the function symbol is not a constant) on a second stack, the argument stack . As required the sub-terms are analyzed further, until a rewrite rule matches. Then conditions are checked and the corresponding right-hand side is substituted. This is done by pushing the appropriate symbols (and complete subterms of terms on the argument stack) on the control stack.

If no rewrite rule matches, a normal form is constructed from the function symbol on the control stack and the required number of terms from the argument stack. Execution continues until the control stack is empty, at which time the argument stack holds the normal form of the input term. Below, we will discuss this algorithm in more detail.

2.1 Machine Model

ARM has two stacks and a heap. The ARM heap contains all tree data-structures. A single primitive exists for the creation of tree nodes. Destruction occurs implicitly by a garbage collector which removes inaccessible data.

The first stack is called the control stack, and it serves three purposes. First, it contains all function symbols still to be evaluated. Initially it contains precisely all function symbols in the subject tree, pushed onto it in a prefix order. Note that no tree parts have to be made for these function symbols, because they will probably be rewritten anyway.

The second of the two stacks is the argument or parameter stack. It contains the values (references to trees residing in the heap) of terms which have been normalized, and for which a parent function symbol still has to be processed (built-in innermost reduction strategy).

The second purpose of the control stack is to hold certain arguments. Two stacks as described above can not represent an arbitrary intermediate stage of the rewriting process. Consider, as an example, the intermediate result g(f(<a>),f(<b>)), where $< \cdots >$ indicates a reference to a tree which has already been normalized, rather than a function symbol, and in which the function symbols g, f and f have yet to be evaluated. Split in the middle, this term can not be distinguished from g(f(f(<a>)),<b>).

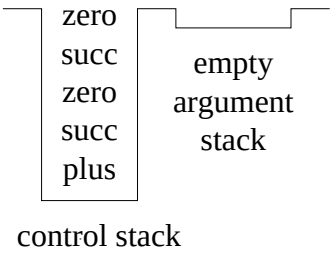
For this reason the control stack may also hold arguments, and when a rule is applied, the instantiated right-hand side is entirely pushed on this stack, again in prefix order. That is, it will contain the objects g, f, <a>, f and <b>, in that order. If, at the beginning of a cycle, ARM finds a tree reference (i.e. a normal form) on the control stack it is simply pushed onto the argument stack and the next cycle may start.

The third purpose of the control stack is to hold control information such as the return location after the evaluation of a condition, and some backtracking information for list matching. This will be discussed where appropriate, in the sequel.

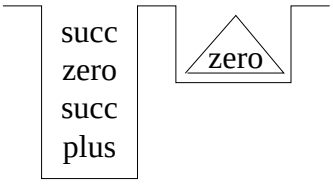
As an example, we will describe ARM's normalization of `plus(succ(zero),succ(zero))` in the TRS consisting of the rules:

```
plus(zero,Nat)        = Nat
plus(succ(Nat1),Nat2) = succ(plus(Nat1,Nat2))
```

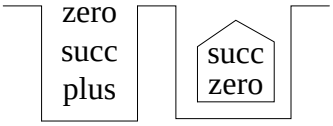
In the pictures below, the two stacks are shown. Initially, the control stack contains all function symbols in the subject term.



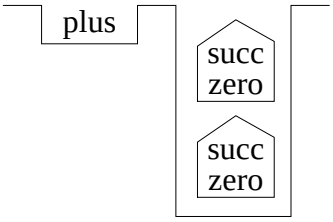
Step 1. No rule for `zero` exists, so the default action is taken: a tree node is built, and a reference to that node (symbolized by a triangle) is put on the argument stack.



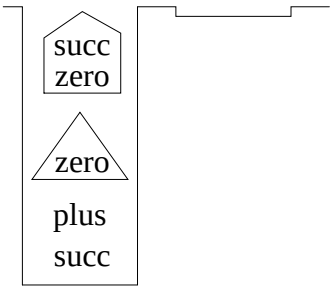
Step 2. Again no rules exist. Since `succ` is a unary function, a single argument is taken off the argument stack, and combined with `succ` into the tree `succ(zero)`. A reference to this tree is put back on the argument stack.



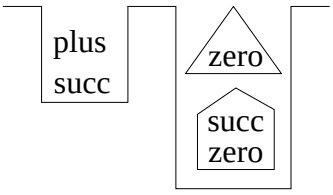
Step 3 & 4. Again, no rules exist for `zero` and `succ`, and the following situation results.



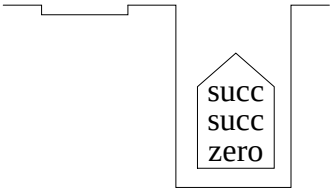
Step 5. Now, the control stack has `plus` on top, and the outermost symbol of the first item on the argument stack is `succ`. Hence rule 2 applies. After removing the arguments of `plus`, its right-hand side is traversed in pre-order, and all is pushed on the control stack.



Step 6. Observe the distinction between, e.g., `plus`, which is a function symbol, and the *tree* that contains `zero`. Note that the trees that were pushed on the control stack, will immediately be moved back to the argument stack. An (unimplemented) optimization is to directly push onto the argument stack these trees.



Step 7. Plus with `zero` as first argument results in the second argument being pushed on the argument stack, which is then combined with `succ` into the tree for `succ(succ(zero))`.



Step 8. As we can see, the input term is normalized as soon as the control stack is empty. At this time, the argument stack holds exactly one (reference to a) term. This is the required normal form.

2.2 Language

The ARM language is a very simple, structural language. It is mainly concerned with tree matching; in particular, tree matching automata [HO82a, Wal91] can easily be expressed in ARM.

The only non-standard feature is that of failures. Almost every ARM statement is conditional; it has a failure part and one or more success parts. If the condition is met, the corresponding success part is taken, otherwise the failure part is taken. This failure part is again an ARM statement. Failures in ARM are comparable though not equivalent to 'failure edges' in tree matching automata as described in [HO82a]. There, strong left sequentiality of the patterns allows reuse of work even if the current match fails entirely.

Ultimately an ARM construct may fail, by some check or verification failing to succeed, indicating that it defines no further options. In this case the failure part of the next enclosing language construct is executed (which may again fail). When an instruction finally 'falls through', the default failure action is taken: the term under consideration is in normal form, and a node is constructed to represent it. In any ARM statement, the failure part may be left out if no explicit failure action is defined.

To a limited degree, ARM has variables. These are used to provide temporary names for sub-terms of subject terms, but they can not be changed. The values of these variables reside on the argument stack. In order to avoid confusion with variables in the underlying term rewriting system, ARM variables will be called *locals*.

Finally, all function symbols defined in the signature of a term rewriting system, and the number and kind of their arguments need to be indicated in an ARM program. This is done implicitly in the outermost ARM instruction. No further declarations are needed.

2.2.1 The select statement

As discussed, the basic cycle of the ARM engine mainly concerns the recognition of a subject tree, the outermost function symbol of which is on the control stack, and the immediate subterms of which are on the argument stack. ARM provides the following two variants of the select statement to express this basic action:

select { *ocase*, *ocase*, ... }
select *source* **in** { *icase*, *icase*, ... } **on failure:** *stat*

where *ocase* looks like *fsym* (*loc*, *loc*, ...) : *stat* , and *icase* looks like *fsym* : *stat* . Here, *stat* is an ARM statement, *fsym* is a function symbol, *loc* is the name of a *local* subterm (i.e. a term residing on the argument stack), and *source* indicates a function symbol in one of the terms on the argument stack, and looks like:

$$loc\ [n_1\ \ldots\ \ n_k\ ]$$

where *loc* is a local and $[n_1\ \ldots\ \ n_k\ ]$ indicates a path in its value. For example, X[2,3] indicates the third sub-term of the second sub-term of the term referred to by X.

An entire ARM program consists of a single (outer) select statement, with each *ocase* clause providing symbolic names for the immediate sub-trees of the subject tree (residing on the argument stack). The outer select statement does not have a failure clause, because failure to process an outermost function symbol evokes the default action: creation of a node, taking its arguments from the argument stack. As mentioned, the failure part of an inner select statement should be left out if no explicit failure action is defined for that statement.

As an example, consider the following term rewriting system:

$$\begin{aligned} \mathtt{f(g(X,a),Y)} &= \ldots \\ \mathtt{f(a,X)} &= \ldots \\ \mathtt{g(b,X)} &= \ldots \\ \mathtt{g(X,b)} &= \ldots \end{aligned}$$

This term rewriting system is represented by the following ARM program.

```
select  {
  f(P,Q): select P[] in {
            g: select P[2] in {
                 a: ...
               }
            a: ...
          }
  g(P, Q): select P[] in {
             b: ....
           } on failure:
           select Q[] in {
             b: ....
           }
        }
```

2.2.2 The check statement

Using the select statement, it is possible to recognize instances of fixed, statically known terms, but it is not possible to compare two statically unknown terms (e.g. two sub-terms of a subject term, as in or(X,X)). For this purpose ARM provides the check statement. It compares two arbitrary terms, and looks like:

check *mterm1* # *mterm2* { *stat* } **on failure:** *stat*;

where # is either = or ≠, and where *mterm* is a so-called meta-term. A meta-term is a term in which no variables may occur, but in which *source* indications may be used.

As an example, consider the condition f(X) = g(Y), where the locals corresponding to the variables X and Y are XX and YY. This condition is checked by the arm statement:

check f(XX[]) = g(YY[]) { .... } on failure: ....

2.2.3 The let statement

The let statement is used to implement conditions in which new variables are introduced. It appears as

let *loc* = *mterm* { *stat* }

The statement associates a new name with a given meta-term. The term is normalized, and its value is left on the argument stack. It is accessed by using the name of the local. This local should not already have been defined in the enclosing statement.

2.2.4 The for statement

In ASF+SDF, lists are a predefined primitive. Any argument of a function may be declared to be a (possibly empty, or non-empty) list of some parameter sort, and variables of such list-sorts may be introduced. During rewriting, these variables will match sub-lists of the argument list. A complication is that list-matching is non-deterministic. Whereas ordinary matching either succeeds or fails, and if it succeeds it does so for one specific substitution of the occurring variables, a list-expression could match a subject term in more than one way. As an example, consider the list-expression 'X, I, Y, I, Z', where I is a variable of some arbitrary sort S, and where X, Y and Z are (possibly empty) lists of S. When this expression is matched against the list 'to, be, or, not, to, be', two subtitutions exist for which the expression matches the subject: X='', I='to', Y='be, or, not' and Z='be', and X='to', I='be', Y='or, not, to' and Z=''.

Our implementation has to make sure that all possible subdivisions of a list have been tried, before a rule can be rejected as non-applicable. This is done by maintaining backtracking information. That is, if during matching of a left-hand side, or any condition, a list expression is encountered, backtrack information is maintained which allows the implementation to remember which alternatives have been checked. Only when all alternatives have been exhausted should the entire match fail.

ARM has a single construct for list matching: the for statement. This statement has one of the following two appearances:

for *source* = [loc_1 n_1; loc_2 n_2; ... *] { *stat* } **on failure:** *stat*
for *source* = [loc_1 n_1; loc_2 n_2; ... !] { *stat* } **on failure:** *stat*

For each sub-list in the corresponding list expression a new local loc_i is declared. The numbers n_i indicate the minimal lengths of the corresponding sub-lists. The symbol * indicates

that the last sublist is allowed to stretch above its minimal length, whereas `!` indicates that the minimal length of the last sublist is also the maximal length.

Each local corresponds to a 'stretching' sub-list (possibly with the exception of the last). All singleton elements are considered to be part of the sub-list terminated by the next list-variable. It is up to the enclosed statement to retrieve these singleton elements from the associated local, and to check them further, if necessary.

Let us consider some examples. Below are two list expressions, each with a corresponding ARM statement. In the list expressions, we assume that `P` and `Q` are variables ranging over non-empty lists, and that X, Y and Z are possibly empty. The locals `PP`, `QQ`, `XX`, `YY` and `ZZ` correspond to the listpatterns `P`, '`s(s(N)),Q`', `X`, '`i,Y`' and '`i,Z`', respectively.

```
P, s(s(N)), Q                    for L = [PP 1; QQ 2 *]  {
                                     select QQ[] {
                                            s: ....


X, i, Y, i, Z                    for L = [XX 0; YY 1; ZZ 1 *] {
                                     check YY[1] = ZZ[1] { ....
```

The for statement is executed by ARM as follows. For each list variable in the clause an entry is put on the argument stack, which is initialized to the suitable sub-list of the subject list (taking the space used by explicit singleton elements, as indicated, into account). For example, given the for statement

```
for L = [XX 0; YY 1; ZZ 1 *] {...  }
```

and the subject list `[4, 3, 2, 1, 2]`, the entries for `XX`, `YY` and `ZZ` are initialized to the empty sub-list before `4`, the sub-list containing `4`, and the sub-list '`3, 2, 1, 2`'. Now matching can be started. If failure is indicated whilst matching the expression in the for statement, or at any point in the body of the statement (this includes possible nested conditions), then a special 'failure' action is taken: the sub-list variables are changed to reflect the next possible division of the subject list (in our example `YY` becomes '`4,3`', and `ZZ` becomes '`2,1,2`'). Then the body of the statement is started anew. Only after all possible list divisions have been examined and rejected, the failure part of the for statement is taken.

The above described procedure has a worst-case exponential complexity in the size of the list expression. Methods exist by which this theoretical complexity is reduced to polynomial proportions [Eke92a]. In practice, however, this exponential behavior rarely occurs. In common cases, the simplicity of our matching algorithm outweighs the overhead incurred by more sophisticated algorithms.

2.2.5 The proceed statement

When an instance of a rewrite rule has been recognized, and all its conditions have been validated, its right-hand side must be 'instantiated'. As mentioned earlier, this is done by placing function symbols and already normalized sub-terms on the control stack. After instantiation, all locals on the argument stack are discarded. This is done with the proceed statement, which looks like:

proceed *mterm*

The proceed statement does not have a failure part; applying a rule can not fail, and in term rewriting there is no backtracking over the application of rules. A summary of all arm statements is given in the

table below:

A summary of ARM statements

Outer select.	`select` {*ocase,...* }
Inner select. *Source* indicates subterm	`select` *source* `in` {*icase, ...* } `on failure:`*stat*
Equality and inequality of terms	`check` *m1* `#` *m2* {*stat*} `on failure:`*stat*
'Open ended' list matching	`for` *source* `= [`loc_1 n_1`; ... *]` {*stat*} `on failure:`*stat*
List matching with fixed size last sublist	`for` *source* `= [`loc_1 n_1`; ... !]` {*stat*} `on failure:`*stat*
Construction of a new term	`proceed` *mterm*

3 Compilation of TRSs into ARM programs

Translating rewrite rules to ARM instructions is a fairly straightforward process. In [WK], we describe the compiler `ASF2C` which transforms TRSs into ARM, and subsequently into a C program. `ASF2C` is an executable algebraic specification, written in the ASF+SDF formalism [BHK89]. `ASF2C` was successfully used to compile itself.

Here, we will only explain the process by showing an example TRS together with the ARM program generated for it. In general, an ARM program can only be constructed using all rules in a TRS. We will therefore assume that this example is a complete specification, defining ordered sets of integers. In this specification, `gt(Nat1,Nat2)` only reduces to `z()` if `Nat2` is strictly larger than `Nat1`. In the equations for sets {}, `X,Y` and `Z` are list variables. Double elements in sets are removed by the first equation, and all elements are sorted by the second equation.

```
gt(z,s(X)) = z()
gt(s(X),s(Y)) = gt(X,Y)

{X i Y i Z} = {X i Y Z}

gt(j,i) = z() ==>
  {X i Y j Z} = {X j Y i Z}
```

```
select{
  z(): fail,
  s(X1): fail,
  gt(X1,X2):
    select X2[] in {
      s: select X1[] in {
           z: proceed z(),
           s: proceed gt(X1[1],X2[1])
         }},
  set(L):
    for L = [L1 0; L2 1; L3 1 *] {
      check L2[1] = L3[1] {
        proceed set([L1(1-),L2(2-),L3(1-)])
      }} on failure:
      for L = [L1 0; L2 1; L3 1 *] {
        check gt(L3[1],L2[1]) = z() {
          proceed set([L1(1-),L3[1],L2(2-),
                       L2[1],L3(2-)])
        }}}
```

In the arm part of this example, it is assumed that the abstract name of a set is `set`, and lists are indicated by rectangular brackets. The notation `L(n-)` is used to indicate that a sublist starting at `n` should be taken from the list `L`. Note that the same list pattern occurs twice in the ARM program above, conform what is currently generated by `ASF2C`. An optimizing compiler could combine these two for loops.

There is no check which verifies if a list indeed occurs as the argument of the set function symbol. Failure would indicate a type error rather than a matching error. In ARM, no type-checking is done, since all type errors can be detected statically. That is, if the program is correct, and the input term is correct, then all results are also correct.

4 C-Implementation

When a TRS is translated into an ARM program, there are basically two options. The first one is to feed it into an ARM interpreter, the second one is to compile it into object-code. A disadvantage of the first option is that it introduces an additional layer of interpretation, which decreases speed. A disadvantage of the second option, however, is the fact that compilation into object code takes additional time. In this section we will discuss the second approach. We will not go into much detail, merely sketching how the various constructs are implemented. Note that this C code is never meant to be generated by hand.

The choice of C as a target language is inspired by portability considerations. It is likely that a hardware specific approach (e.g. ARM to 68K or SPARC) would yield better results. The main objection to C is the inaccessibility to the location of control[1]. That is, it is impossible to store and later retrieve the current location (this behaviour could be mimiced with function calls, using function pointers, but that feature comes at a substantial price in efficiency). Our solution is to store location tokens (integers), and to use these in a jump table.

Before we start an ARM-specific discussion, we would like to make notes on the compilation time and the code size. The code for one specification contains only one big function. Whereas this removes the overhead for function calls completely, there is a potential danger resulting from the non-linear complexity of the optimizations done in C compilers. Currently, the C compilation phase is the bottleneck in `ASF2C`, but this is even the case with all optimizations switched off. The code size is negatively influenced by the use of macros for all basic actions in ARM. However, as it is put in [WB90]:

> The usage of macros instead of subroutines produces more lengthy code but the resulting program is much more efficient. All overhead of subroutine calls is avoided, and instruction-prefetch queues and program caches of modern processors can be used in an optimal way.

The main body of a program generated by `ASF2C` is an infinite loop, which is entered after initializing the control stack with (the function symbols of) an input term, and which is only exited when that stack is exhausted. The resulting normal form is the single item remaining on the argument stack.

The basic mode of control is the C `continue` statement, which restarts ARM by re-starting this loop. Furthermore, failure cases simply follow success cases (which appear inside C control structures such as `if` or `switch`). In this manner, control automatically reaches failure cases as long as no explicit `continue` is performed. The very last action in the loop, which is the most general failure case, is the construction of a tree node using the supplied function symbol.

Function symbols are odd integers, whereas addresses in the heap are always even. This is a well-known technique to implement type tags without consuming additional space. The first action in the outer loop is to check whether the top element of the control stack is a function symbol or a heap value. In the latter case it is immediately placed on the argument stack, and the loop is restarted.

The `select` statement can be translated directly to a `switch` statement. The failure part of the `select` is not placed in the default section, but immediately after the `switch` statement, for it should be executed, for example, if a condition fails. When a branch of the `switch` is succesful, control will not reach the end of the `switch`, because `continue` is used.

The `check` statement is similarly translated to an `if` statement. Again the failure part appears after the `if` rather than in the `else` part.

[1] In gcc there is a non-standard operation to obtain the address of a label, which we have not used for portability reasons

The `for` statement is not translated to the C `for`, because control may exit, and later re-enter this loop, which is undefined in C. Labels and `goto`'s are used for this purpose. After pushing a slot with proper initialization on the argument stack for each list variable, a label appears. After the code corresponding to the `for` body, but before its failure part, code is generated which goes through all possible list divisions, each time updating the list items on the argument stack and jumping to the label mentioned above. After this code, we find the code corresponding to the failure part of the for statement itself.

The `proceed` statement pushes function symbols and subterms corresponding to its argument on the control stack and then enters the next cycle using C's `continue`.

The `check` statement pushes a token for the current location on the control stack, followed by the special operator (equal or inequal) and the symbols and subterms corresponding to its arguments. Then the next cycle is started using C's `continue`. When the special operator is encountered, (in)equality of its arguments is tested, and control returns to the indicated location, where the failure or the success branch is taken, depending on the result of the test.

The code for the `let` instruction is similar. In a `let`, only the side without new variables is pushed on the control stack. On return, the result is on top of the argument stack, ready to be investigated by the matching code generated from the other side of the condition.

5 ARM on the testbench

The reader might expect here a comparison with the speeds reported for interpreters of TRSs, or with those reported for compilers developed in various research projects which are similar to ours [WB90, Ken90]. Considering the fact that the ASF+SDF interpreter is already one of the fastest interpreters known to us [Eke92b], and the fact that compiled code runs roughly fifty times as fast as interpreted code, we will not pay attention to *interpreted* systems.

For two other reasons, we will also refrain from systematically comparing similar *compilers* directly. The first reason is that a lot of time is involved in obtaining versions of those systems that run in our environment, installing them, and getting to know the formalisms. Trying to compare the systems on paper is an even more hopeless task. In some cases, just the number of reductions per second was given, without the rewrite system that was tested. In other cases, the rewrite systems were so simple, that only a few machine instructions were left after optimization. Such tests tell more about the hardware they run on, than about the speed that can be expected for an average rewrite system.

The second reason is that many of the other systems focus on specific optimizations, whereas `ASF2C` does a very straightforward job. Rough estimates indicate that some optimizations (including generation of machine-specific code) will yield speeds comparable to those reported by others. Not withstanding these observations we have tried to evaluate figures and considerations in various publications. A discussion is presented in section 6.

Instead of the comparison above, we have attempted to make a fair comparison between `ASF2C` and commercial or generally valued compilers for several other languages; C, ML, and Prolog. The specifications and programs we used, are ftp-able from the directory `pub/gipe` of the site `cwi.nl`. We would like to encourage every implementor of a system similar to ours to run these specifications, and report on their findings. For *lazy* functional languages, an extensive set of benchmarks was studied in [Lan93].

Naturally, we have chosen a number of problems from domains where executable algebraic specifications are an appropriate way to find solutions. We stress that the results we found are only meaningful within this domain. Particularly with respect to the language C, we have to remark that it is not well suited for the kind of problems that are solved using algebraic specifications. Because of its widespread use and portability, it *is* however a probable candidate

for the final implementation of the specifications. ML and Prolog were chosen because they are most likely to be considered as direct alternatives to writing executable algebraic specifications. The versions of the compilers used are: GNU cc (gcc) version 2.2.2 without any optimization flags on, BIMProlog 3.1[2] , and Standard ML of New Jersey version 75 of 11 november 1991 which generates native code for our sparcstation. The C-code generated by ASF2C was compiled by gcc 2.2.2 also without any optimization flags on.

5.1 Criteria for language comparison

There are a number of caveats in trying to compare such immensely different programming languages. Firstly, all languages considered have predefined abstract datatypes such as lists and arrays, which are treated in a special way to improve efficiency. It is difficult to find exact equivalents of these features in all compared languages. Moreover, the speed of a program that makes extensive use of one of these special datatypes is a bad measure for the overall efficiency of the programming language under consideration. For these reasons, we have refrained from using other than basic datatypes while constructing the benchmarks. It should be noted that we do not even view integers as basic datatypes. In fact only symbolic constants and tuples or structs are used in the benchmark programs.

A second caveat is expressive power. Even if only basic datatypes are used, most problems occurring in the benchmarks have a 'natural' solution in the languages considered. E.g. list reversal is naturally solved by a recursive function in ML or a recursive predicate in Prolog, but in C one would choose a while loop. More involved is the non-trivial use of unification or nondeterminism in Prolog to solve certain problems. It is hard to say how well these language-specific styles of programming scale with program size, therefore we have chosen for a uniform, functional, programming style in all languages considered.

A third caveat is the runtime behaviour of the programs. In the executables that perform garbage collection (ML and Prolog), we have taken care not to measure garbage collection time. In the C programs we allocate one private block of memory to eliminate the well-known overhead incurred by the use of `malloc`. Measurements of the system time do not give much information and tend to show large variations. Therefore, we have only measured cputime.

Finally, the kind of program could influence the relative speeds of the executables; every compiler is expected to excel in a different area. We have distinguished seven factors that could (more or less independently) lead to different relative speeds.

- *The number of rewrite rules.* This is a measure of the program size. It is expected that not all languages behave the same under an increase of program size.
- *The number of function symbols.* Especially for highly optimizing pattern-matching languages, this factor is expected to have influence.
- *Average arity of functions.* Effects of the average arity are expected to correlate with the effects of pattern size, amount of recursion and the overlap of patterns.
- *Condition depth.* This is the average number of conditions per rewrite rule.
- *Overlap of patterns.* The degree of overlap determines the time to find the correct match.
- *Amount of recursion.* This is expected to differentiate between functional and imperative languages.

[2] One of the fastest WAM-based commercially available Prolog compilers

Table 1: Aspects of the benchmarks

aspect/ benchmark	number of rules	number of symbols	arity	condition depth	overlap	recursion	pattern size
nrev	4	6	1	0	–	+	2
nats	87	12	0.33	0	–	–	2.9
arm	27	32	1.41	0.22	+/–	+	2.92

- *Pattern size.* A measure of the efficiency of the pattern matching. We counted the number of function symbols in the left-hand sides of rules and conditions.

Ideally, we could have sought seven problems that independently measure these factors and also satisfy the constraint that only basic datatypes must be used. Apart from the fact that this proves to be very difficult, writing and testing the 35 resulting programs would have taken up too much time. Not knowing if these factors have differing relative importance doesn't encourage one to start the implementation effort. Therefore, we have chosen to implement only three problems, spanning the extremes for all factors, but not independently; naive reversal of lists (nrev), addition on a decimal representation of integers (nats), and an interpreter for our intermediate language ARM (arm).

Whereas these three problems give a good impression of the *average* speed to be expected of `ASF2C`, we also felt the need for a specification that would show the highest speed possible, comparable to the nfib problem for implementations of functional programming languages. Due to the fact that machine arithmetic is unavailable in ARM, it is impossible to write a specification that results in a huge number of the simplest rewrite steps thinkable. Therefore, in the benchmark max, we embedded an equation causing such a simple rewrite step in a more complex TRS, and measured the difference in execution speed of this TRS, and a TRS without the simple equation. This results in a maximum of 416000 simple rewrite steps per second.

5.2 The benchmarks

In table 1, we give an overview of the relevance of the abovementioned aspects for the three problems. Most of the scores have been obtained simply by counting in the ASF+SDF specifications. The scores that show +'s and –'s only give relative judgements.

The arity, condition depth and pattern size are only *static* averages. Though computing a runtime average would probably yield slightly different figures, close inspection of the programs shows that the relative order of the benchmarks with respect to these aspects would be the same. Therefore we have not bothered to measure the runtime averages.

The specifications in ASF+SDF were always the shortest and easiest to write. The Prolog and ML versions took a little bit more effort; the Prolog benchmark programs look less elegant than their ASF counterparts because of the extra argument that must be added to pass the result of a function. The ML versions are a little bit more complicated because non-free constructors (occuring in the Nats benchmark) must be translated using both a defined and a constructor function. Only in the C versions we had the feeling to be bothered with an unnecessary amount of low-level detail. The ASF+SDF specifications can be found in appendix A.

5.3 The Measurements

In table 2, we give the results of our measurements. The machine used was a SparcStation 1+ (sun4/65 with 12MB internal memory) running SunOS 4.1.1. All measurements were performed three times with inputs causing running times sufficiently large to show variation only in the last digit. If a loop was needed to obtain a longer running time, the same loop was run with a dummy task and the resulting time subtracted. The task given to nrev was to reverse a list of length 128 for 64 times, arm was given an ARM program to compute 7!, and nats 16384 times added 66666 to 66666 (this number was chosen to cause many carries during addition). The

Table 2: Measurements (cputime, seconds)

Benchmark	BIMProlog 3.1	NJsml 75 11/11/91	gcc 2.2.2	ASF2C
nrev	1.70	1.29	4.62	2.57
nats	6.53	1.88	6.81	4.50
arm	10.45	1.90	–	6.81

first thing to be observed is the last position of C[3] in the nrev and nats benchmarks. The arm benchmark has not been implemented in C, because we did not expect enough difference in its execution time to justify the implementation effort.

Bearing in mind that the overhead involved when using `malloc()` has already been eliminated by using explicit memory management, we conclude that C is a bad alternative for direct implementation of algorithms for symbolic computation. To put it slightly differently; using C for the final implementation of a specification in one of the other languages can only be beneficial if knowledge of the specification can be used to find representations that are much more efficient in C.

In the measurements for Prolog, it can be seen that the expressive power provided by Prolog's unification and backtracking comes at a price; only in the case of `nrev` it can easily be inferred that no backtracking information needs to be kept. In the Prolog code for the arm interpreter, we have even inserted 'green' cuts to make explicit its deterministic character, otherwise the code takes an order of magnitude more time and runs out of available memory. Just like the replacement of `malloc` in the C benchmark, the insertion of green cuts is a relatively cheap transformation that can be performed using only superficial meta knowledge of the specification.

The ML compiler proved to be two to four times as fast as `ASF2C`. Given the fact that we have not implemented any optimizations such as generation of machine specific code[4], tail recursion elimination, partial evaluation, common subexpression elimination, use of constructor information or even the prevention of unnecessary movements to and from the stack, we feel that the results of `ASF2C` are satisfying and *very* promising.

6 Related work

We are certainly not the first to remove a layer of interpretation from a term rewriting implementation. Most of the ideas in our paper occur in some form in earlier work. Therefore, we

[3] The reader may wonder how it is possible that the C program generated by ASF2C is faster than the handwritten C program. This is because the generated program has an entirely different structure; it consists of one single function, allowing it to keep much information in registers, and it has its own stack and heap management.

[4] According to the folk-lore, a factor of 2-3 can be gained by this technique.

will only claim an original *combination* of these ideas into a model that is very simple, fast, powerful and nonrestrictive. This is not to say that we developed `ASF2C` as a cocktail of existing ideas; a more faithful representation is to say that we sometimes reinvented variants of existing approaches. With this in mind, we will discuss some compilers for formalisms similar to ours.

6.1 Compilers mapping functions one to one

In the first class of compilers we will consider, one target-language function is generated per defined function in the TRS. Compared to our approach, strong points of this approach are:

- A relatively easy interface with other software, because compiled functions can be accessed as a library.
- The implicit call stack in the target language eliminates the need for an explicit control stack.
- Separate compilation of modules or functions is relatively easy. Incremental development only requires relinking.

There are also disadvantages:

- No fine tuning of control. E.g, elimination of general tail recursion is impossible.
- The function call mechanism in the target language is more general, and consequently slower, than what is needed in the generated programs.
- The memory management of the target language implementation caters for more than is actually needed. For example, garbage collection of LISP has to cope with cyclic structures, which do not occur in TRSs.

Compilation of CTRSs into LISP programs is described in [Kap87] and [Dik89]. As in all other compilers mentioned in this section, no built-in associative lists are provided, but otherwise the formalism compiled has the same expressive power as ours. Reliance on features of LISP, especially the features used to implement the optimisations, makes this approach less portable than ours. It is difficult to compare the timings given with ours.

In [Heu88], a compiler similar to Kaplan's is described. Matching sequences that can be shown to always fail (by using the fact that subterms are always in normal form under an innermost reduction strategy) are eliminated. However, this optimization is only computable for unconditional TRSs.

In [GHM88], compilation of CTRSs into PASCAL programs is described. The formalism is much more restrictive than ours; overlapping patterns, non-linearity, and non-free constructors are not supported. From the timings on a sun3, we estimate that the code would run at 20000 r/s on our machine.

In [Gar90], compilation of the data part of LOTOS specifications into libraries of C functions is discussed. No non-free constructors are allowed, but a transformation to eliminate them is given.

In [SG90], the compilation of OPAL to C functions is described. A reference counting (RC) scheme is used for garbage collection. At compile time, the overhead of RC is decreased by dataflow analysis, and at runtime, updates on a node with zero RC do not create garbage by reusing the node. The formalism is both stronger and weaker than ours. Higher order functions are supported, but overlapping patterns with backtracking over conditions lead to a duplication of code. The speed was measured on sun4/75 (32MB) hardware, slightly more powerfull than

our machine. The only benchmark that can be compared well is the Tree benchmark; form a complete binary tree, flatten it, and reverse the resulting list 4 times. For a tree of depth 13, this takes .57 seconds without RC optimizations, and .143 seconds with the RC optimizations. Note that the 'reverse' function (most frequently called in the example) profits maximally from the RC optimization; no space is consumed at all. Our code takes .77 seconds for Tree.

In [Sch88], rewrite rules are translated into an abstraction of imperative programming languages; 'if-then-else' structures. The algorithm is claimed to be very similar to [Aug85], but it is optimized with respect to the domain over which the function should be compiled. An interesting point is that the algorithm may give more efficient code when non-free constructors are allowed.

6.2 Compilers mapping to an abstract machine

The idea to use an abstract machine as an intermediate level for compilation is widespread in the area of functional programming languages [Ken90, FW87, CCM85] and logic programming languages (many commercial implementations of Prolog are based on WAM, the Warren Abstract Machine [AK91]). However, most of these machines are an order of magnitude more complicated than ARM. In the case of the WAM, this is because unification and general backtracking available in Prolog are much more difficult to implement than matching and the limited form of backtracking that occurs in ASF+SDF. In the case of functional languages, higher order functions are the complicating factor. For all machines, we will give the number of run-time parameters (program counter, stack pointers and pointers to other spaces) and the number of (parameterized) instructions. ARM has four runtime parameters, and four (two of which have two variants) instructions.

In [WB90], compilation of the data part of LOTOS into code for LATERM (Lazy Abstract TErm Rewriting Machine, implemented by macro's in 68020 assembler) is described. A speed of 100000 r/s on a sun3/160 is claimed, but it is not clear which TRS was used for this measurement. The formalism is much more restrictive (TRSs must be unconditional, stable, and left linear). LATERM is more complicated than ARM; it has variable bindings, 3 reduction strategies, 6 run-time parameters and 22 instructions.

In [KI89], attention is focussed on the efficient implementation of functions on recursively defined datastructures without function activation records. To cater for this, two extra stacks are needed. The abstract machine has 6 parameters and 12 instructions. The TRS may not contain non-free constructors, patterns must be less than two deep, and in each defined function, all constructor cases must appear.

In [Str89, SSD91], CTRSs in the class of *forward branching* (FB) *programs* are compiled to EM code. The class FB is less restrictive than strong left sequentiality [HO82b], but still a subset of strongly sequential programs [HL79]. EM has 6 instructions and an unbounded number of registers. The matching automaton is extended by taking righthand sides into account; several rewrite steps are combined into one matching cycle. From the explicit handling of registers, it appears that the abstraction level of EM is a bit lower than ARM.

Finally, we will consider some abstract machines used for the implementation of (higher order) functional programming languages. In most of these machines, a functional framework is adhered to as long as possible, whereas in our approach, the only intermediate level (ARM) is already imperative. Landin's SECD machine [Jon87] is the first abstract machine for functional languages and a source of inspiration for many others. Here, we will only discuss some modern descendants.

As [Jon87] notes, all implementations of functional languages make use of built-in functions to implement pattern matching, though this is not a theoretical necessity. In an attempt to

compare the numbers of instructions available in the several abstract machines, it should be kept in mind that ARM needs no built-in functions at all; the four instructions described in section 2 form a complete set.

The simplest architecture for functional languages is the Three Instruction Machine (TIM), described in [FW87], which has three (parameterized) instructions and three run-time parameters. The operational behaviour of TIM shows a remarkable similarity to ARMS's, apart from the use of frames, dictated by the higher order features, and the absence of a second stack which could be viewed as an optimization in ARM.

Another class of machines is based on Turner's SK machine [Tur79], which implements the lambda calculus using combinatorial logic. An example is the Categorial Abstract Machine (CAM), described in [CCM85]. It has 8 instructions and 3 runtime parameters, and it is claimed that compiling CAM code to machine-specific code yields faster code than an equivalent C program.

Finally, the ABC machine [Ken90], used in the implementation of Clean [vENPS90], based on graph rewriting, has three stacks, a descriptor store, a program counter, a program store, 50 instructions for rewriting, and a still larger number to handle basic values. No non-free constructors are allowed. Impressive speeds are claimed for machine specific implementations on stock hardware. On a sun3/280, 495000 function calls per second were measured. A direct comparison with our results is hard, because the benchmark (nfib) uses machine arithmetic in an esential way. Of our benchmarks, max (416000 r/s on a sun4) most closely corresponds to nfib.

7 Conclusions and future work

In this paper, we have presented an abstract machine for the implementation of conditional term rewriting systems with associative lists. We do not claim fundamentally new ideas, but we do claim a combination of several ideas into a model that is simple, fast, powerful and nonrestrictive. Summarizing section 6, we note that simplicity is reflected in the complete lack of predefined functions, the small number of both instructions and run-time parameters and the ease of both the compilation from ASF+SDF into ARM and from ARM into C. The speed shows from the benchmarks. Direct handling of associative lists, and the handling of overlapping patterns by failure parts make ARM a powerful machine. The fact that non-free constructors, (negative) conditions, new variables in conditions, non-left-linearity, overlapping and deep patterns are allowed, only adds to this power.
From a software engineering point of view, we conclude from the benchmarks, that

- The specifications in ASF+SDF were the shortest and most readable. Of course, we are not the most objective judges of the readability of ASF+SDF specifications.
- ML produces the fastest code, but it is to be expected that future optimizations will narrow the gap between ML and ASF2C.
- Using C for a straightforward implementation of a specification does not bring efficiency. Only if meta knowledge is used to find efficient representations, an efficiency gain may be expected.
- If a specification can be easily written using the expressive power available in ASF+SDF or ML, there is no reason to use Prolog with its more powerful features.

In the future, we will add some optimizations to the current implementation of `ASF2C`.

- The simplest optimization was already mentioned: decreasing the vacuous traffic of normal forms from the control stack to the argument stack.
- This can be generalized to the immediate construction of terms if (parts of) the right hand side of a rewrite rule are known to be in normal form.
- Then there is a notion of *matchless rewriting*; deducing from a right hand side what matching steps will be taken in the following cycle, and immediately jumping to the corresponding position in the arm program.
- In the C-version of the arm program, some other small optimizations concerning the use of registers can be done.
- Lastly, there is the possibility of machine-specific code generation.

Another path of research is concerned with reducing the compilation time. Currently, the main bottleneck is the speed of the C-compiler. A cure for this problem is interpretation, instead of compilation, of ARM. The speed of compiled specifications will decrease as a result of this, but flexibility will increase significantly. To overcome the decrease in speed, it could be worthwile to investigate the combination of interpreted code for incremental changes and compiled code for the stable part of a specification.

A The specifications used for benchmarking

A.1 Nrev

```
[a1] append(nil,X) = X
[a2] append(cons(X,Y),Z)
     = cons(X,append(Y,Z))
```

```
[n1] nrev(nil) = nil
[n2] nrev(cons(X,Y))
     = append(nrev(Y),cons(X,nil))
```

A.2 Nats

In the Nats benchmark, the equations labeled `z-i` and `d-j` define addition on digits, and the equations `n-k` define addition on composite numbers (composed by the function `nat`). The left argument of `nat` must be interpreted as being multiplied with one power of 10 more than the right argument, the equation `c-2` removes leading zero's, and the equation `c-1` takes care that the right argument of a `nat` term in normal form always contains only one digit. Thus `plus(nat(nat(1,5),6),nat(3,5))` yields as normal form `nat(nat(1,9),1)`, easily interpreted as representing 191. A more detailed analysis of a similar specification for binary numbers, including a proof of termination, can be found in [Wal91].

```
[z-1] plus(0,X) = X
[z-2] plus(X,0) = X

[d-1] plus(1,1) = 2
[d-2] plus(2,1) = 3
  .
  .
[d-81] plus(9,9) = nat(1,8)
```

```
[n-1] plus(nat(B,S),X)
      = nat(B,plus(S,X))
[n-2] plus(X,nat(B,S))
      = nat(B,plus(S,X))

[c-1] nat(B1,nat(B2,S))
      = nat(plus(B1,B2),S)
[c-2] nat(0,S) = S
```

A.3 A μARM interpreter in ASF+SDF

The specification presented in this section serves a dual purpose. Firstly, it is a non-trivial specification that involves considerable amounts of matching during execution. Secondly, it gives a formal specification of the operational semantics of a somewhat restricted version of ARM. μARM supports only functions with arity 0,1 or 2 (higher arities can be dealt with by compounding multiple arguments into tuples or lists), and no `for` construct is supported. μARM will be used in an experiment where specifications with associative lists are transformed into a specification with only `cons` lists. Nevertheless, μARM shows almost all interesting aspects of ARM, especially the failure handling.

```
%%  Module Terms
imports Layout
exports
  sorts FSYM MTERM
  context-free syntax
    ap0( FSYM )                    -> MTERM
    ap1( FSYM, MTERM )             -> MTERM
    ap2( FSYM, MTERM, MTERM )      -> MTERM
    void                           -> MTERM
    "0"                            -> FSYM
    "s"                            -> FSYM
    "+"                            -> FSYM
    "*"                            -> FSYM
    "!"                            -> FSYM
  variables
    Fsym[0-9']*                    -> FSYM
    Mterm[0-9']*                   -> MTERM
```

In the module Terms, terms with a function symbol and 0, 1 or 2 children are represented using the functions `ap0, ap1` and `ap2` (cf. the meta-terms on page **??**). The function `void` is used only where a placeholder for a tree is needed. For the sake of readability, a fixed number of suggestive function symbols is provided in the benchmark (`!, *, +, s` and `0`). A more realistic specification would provide an infinite supply of function symbols.

```
%%  Sources
imports Terms
exports
  sorts INDEX LOC PATH
  context-free syntax
    funof( MTERM )                 -> FSYM
    zero                           -> INDEX
    one                            -> INDEX
    two                            -> INDEX
    here                           -> PATH
    down( INDEX, PATH )            -> PATH
    retrieve( PATH, MTERM )        -> MTERM
    loc( INDEX, PATH )             -> LOC
    at( LOC )                      -> MTERM
  variables
    Path[0-9']*                    -> PATH
    Loc[0-9']*                     -> LOC
    Index[0-9']*                   -> INDEX
equations
[Funof-0] funof(ap0(Fsym)) = Fsym
[Funof-1] funof(ap1(Fsym,Mterm)) = Fsym
```

```
[Funof-2] funof(ap2(Fsym,Mterm,Mterm')) = Fsym


[Retrieve-here] retrieve(here,Mterm) = Mterm
[Retrieve-one1] retrieve(down(one,Path),ap1(Fsym,Mterm)) = retrieve(Path,Mterm)
[Retrieve-one2] retrieve(down(one,Path),ap2(Fsym,Mterm,Mterm')) = retrieve(Path,Mterm)
[Retrieve-two2] retrieve(down(two,Path),ap2(Fsym,Mterm,Mterm')) = retrieve(Path,Mterm')
```

In the module *Sources*, access to the components of terms (possibly living on the argument stack) is defined. The function symbol of a term can be obtained with the function `funof`.

An `INDEX` is used for three purposes. Firstly, an index is used to indicate the arity of a term to be created by a `create` or a `proceed` instruction. Secondly, an index is used in a `PATH` to indicate which child should be taken to find a particular subterm. The function `retrieve` actually retrieves such a subterm. Thirdly, an index is used to indicate a position on the argumentstack in the function `loc`. With `at`, a subterm of a term on the argumentstack can be obtained. Note that sources are added to the sort `MTERM`, thus extending the sort to exactly what is needed in the proceed statement.

```
%% Arm
imports
  Layout Terms Sources
exports
    sorts
        VALUE STACK CASE STAT ARM
    context-free syntax
        fsym( FSYM )                                  -> VALUE
        tree( MTERM )                                 -> VALUE
        mt                                            -> STACK
        psh( VALUE, STACK )                           -> STACK
        select( LOC, STAT )                           -> STAT
        cases( CASE, STAT )                           -> STAT
        case( FSYM, STAT )                            -> CASE
        proceed( INDEX, MTERM )                       -> STAT
        create( INDEX, FSYM )                         -> STAT
        arm( STACK, STACK, STAT )                     -> ARM
        eval( ARM )                                   -> ARM
        arm_case( FSYM, STAT, STACK, STACK, STAT )  -> ARM
        arm_execute( STAT, STACK, STACK, STAT )     -> ARM
        arm_process( MTERM, MTERM, MTERM, STACK )   -> STACK
        nfof( ARM )                                   -> MTERM
      variables
        Val[0-9']*                                    -> VALUE
        Stack[0-9']*                                  -> STACK
        [MA]S[']*                                     -> STACK
        Case[0-9']*                                   -> CASE
        Stat[0-9']*                                   -> STAT
        Arm[0-9']*                                    -> ARM
equations
[Eval-tree]
  eval(arm(psh(tree(Mterm),MS),AS,Stat)) = eval(arm(MS,psh(tree(Mterm),AS),Stat))
[Eval-fsym]
   eval(arm(psh(fsym(Fsym),MS),AS,Stat)) = eval(arm_case(Fsym,Stat,MS,AS,Stat))


[Case-this]
```

```
  arm_case(Fsym,cases(case(Fsym,Stat),Stat1),MS,AS,Stat0)
  = arm_execute(Stat,MS,AS,Stat0)
[Case-further]
  Fsym != Fsym' ==>
    arm_case(Fsym,cases(case(Fsym',Stat),Stat1),MS,AS,Stat0)
    = arm_case(Fsym,Stat1,MS,AS,Stat0)
[default-Case]
  arm_case(Fsym,Stat,MS,AS,Stat0)
  = arm_execute(Stat,MS,AS,Stat0)


[Select-1]
  retrieve(Path,Mterm1) = Mterm,
  funof(Mterm) = Fsym ==>
    arm_execute(select(loc(one,Path),Stat),MS,psh(tree(Mterm1),AS),Stat0)
    = arm_case(Fsym,Stat,MS,psh(tree(Mterm1),AS),Stat0)
[Select-2]
  retrieve(Path,Mterm2) = Mterm,
  funof(Mterm) = Fsym ==>
    arm_execute(select(loc(two,Path),Stat),MS,
                psh(tree(Mterm1),psh(tree(Mterm2),AS)),Stat0)
    = arm_case(Fsym,Stat,MS,psh(tree(Mterm1),psh(tree(Mterm2),AS)),Stat0)


[Proceed-0]
  arm_execute(proceed(zero, Mterm),MS,AS,Stat0)
  = arm(arm_process(Mterm,void,void,MS),AS,Stat0)
[Proceed-1]
  arm_execute(proceed(one, Mterm),MS,psh(tree(Mterm1),AS),Stat0)
  = arm(arm_process(Mterm,Mterm1,void,MS),AS,Stat0)
[Proceed-2]
  arm_execute(proceed(two, Mterm),MS,psh(tree(Mterm1),psh(tree(Mterm2),AS)),Stat0)
  = arm(arm_process(Mterm,Mterm1,Mterm2,MS),AS,Stat0)


[Create-0]
  arm_execute(create(zero, Fsym),MS,AS,Stat0)
  = arm(MS,psh(tree(ap0(Fsym)),AS),Stat0)
[Create-1]
  arm_execute(create(one, Fsym),MS,psh(tree(Mterm),AS),Stat0)
  = arm(MS,psh(tree(ap1(Fsym,Mterm)),AS),Stat0)
[Create-2]
  arm_execute(create(two, Fsym),MS,psh(tree(Mterm1),psh(tree(Mterm2),AS)),Stat0)
  = arm(MS,psh(tree(ap2(Fsym,Mterm1,Mterm2)),AS),Stat0)


[Process-0]
  arm_process(ap0(Fsym),Mterm1,Mterm2,MS)
  = psh(fsym(Fsym),MS)
[Process-1]
  arm_process(ap1(Fsym,Mterm),Mterm1,Mterm2,MS)
  = arm_process(Mterm,Mterm1,Mterm2,psh(fsym(Fsym),MS))
[Process-2]
  arm_process(Mterm,Mterm1,Mterm2,psh(fsym(Fsym),MS)) = MS' ==>
    arm_process(ap2(Fsym,Mterm,Mterm'),Mterm1,Mterm2,MS)
    = arm_process(Mterm',Mterm1,Mterm2,MS')
[Process-arg1]
```

```
  arm_process(at(loc(one,Path)),Mterm1,Mterm2,MS)
  = psh(tree(retrieve(Path,Mterm1)),MS)
[Process-arg2]
  arm_process(at(loc(two,Path)),Mterm1,Mterm2,MS)
  = psh(tree(retrieve(Path,Mterm2)),MS)


[Normal-Form]
  nfof(arm(MS,psh(tree(Mterm),AS),Stat)) = Mterm
```

The module *Arm* starts with the definition of the values that can be found on the control and argument stacks; function symbols and (references to) trees. A `STACK` is either empty (`mt`) or contains a value pushed on another stack (`psh`).

Arm continues with the constructor functions that define ARM programs. A `select` contains a list of cases constructed with the function `cases`. A `proceed` has an extra `INDEX` argument that indicates the arity of the function in the current case of the outer `select`, apart from the meta-term that must be proceeded. The `create` statement creates a node with the required arity and function symbol. The function `arm` represents an ARM machine together with the program it is running. The extra argument in `proceed`, and the `create` function are not present in ordinary ARM. In μARM, they are needed, because ocasionally, when a proceed or a construction of a normal form takes place, it is otherwise not known to the interpreter how many arguments must be popped off the argument stack.

The evaluation of an ARM program is defined by the function `eval`. If a (reference to a) term is found on the control stack, it is moved to the argument stack. If a function symbol is found, the correct case in the outer `select` is found by `arm_case`, to be evaluated by `arm_execute`. Otherwise, a normal form has been reached (the control stack is empty), or an error has occurred. In this case, eval just returns the state of the ARM machine.

`arm_execute` executes the statement in its first argument, using the control and argument stacks in its second and third argument. In the fourth argument, the statement to be executed on failure is passed. The most complicated statement is the `proceed` statement, which leads to the processing (by `arm_process`) of a meta-term, a recipe for an update on the control stack. Finally, `nfof` extracts the computed normal form from the argument stack.

References

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine. A Tutorial Reconstruction.* The MIT Press, 1991.

[Aug85] Lennart Augustsson. Compiling pattern matching. In J.P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, 1985.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 1985.

[Dik89] Casper H.S. Dik. A fast implementation of the algebraic specification formalism. Master's thesis, Faculty of Mathematics and Computer Science, University of Amsterdam, February 1989.

[Eke92a] S.M. Eker. Associative matching for linear terms. Report CS-R9224, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[Eke92b] S.M. Eker. A comparison of obj3 and asf+sdf. Report CS-R9223, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45. Springer-Verlag, 1987.

[Gar90] Hubert Garavel. Compilation of lotos abstract data types. In S.T. Vuong, editor, *Formal Description Techniques, II*, pages 147–162. Elsevier Science Publishers B.V. (North-Holland), 1990. IFIP, 1990.

[GHM88] A. Geser, H. Hussmann, and A. Mück. A compiler for a class of conditional term rewriting systems. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 84–90. Springer-Verlag, 1988.

[Heu88] Thierry Heuillard. Compiling conditional rewriting systems. In S. Kaplan and J.P. Jouannaud, editors, *Proceedins of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 111–128. Springer-Verlag, 1988.

[HL79] G. Huet and J.-J. Lévy. Call by need computations in non-ambiguous linear term rewriting systems. Rapports de Recherche 359, INRIA, 1979. To appear as: Computations in Orthogonal Rewriting Systems, Part I and II, in J.L. Lassez and G. Plotkin, editors, *Computational Logic, essays in honour of Alan Robinson*, MIT Press, 1991.

[HO82a] C.M. Hoffmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[HO82b] C.M. Hoffmann and M.J. O'Donnell. Programming with equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112, 1982.

[Hus88] H. Hussmann. The Passau RAP system: rapid prototyping for algebraic specifications. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 264–265. Springer-Verlag, 1988.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Kap87] S. Kaplan. A compiler for conditional term rewriting systems. In P. Lescanne, editor, *Proceedings of the First International Conference on Rewriting Techniques*, volume 256 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 1987.

[Ken90] R. Kennaway. The specificity rule for lazy pattern-matching in ambiguous term rewrite systems. In N. Jones, editor, *ESOP '90 - Proceedings of the Third European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 256–270. Springer-Verlag, 1990.

[KI89] H. Klaeren and K. Indermark. Efficient implementation of an algebraic specification language. In M. Wirsing and J.A. Bergstra, editors, *Proceedings of the METEOR workshop on Algebraic Methods: Theory, Tools and Applications. Passau 87*, volume 394 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[Lan93] Koendert Gustaaf Langendoen. *Graph Reduction on Shared-Memory Multiprocessors*. PhD thesis, University of Amsterdam, 1993.

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

[SG90] Wolfram Schulte and Wolfgang Grieskamp. Generating efficient portable code for a strict applicative language. In *Phoenix Seminar and Workshop on Declarative Programming, Hohritt (Sasbachwalden, Germany)*, Lecture Notes in Computer Science. Springer-Verlag, 1990. to appear.

[SSD91] David Sherman, Robert Strandh, and Irène Durand. Optimization of equational programs using partial evaluation. *ACM SIGPLAN Notices*, 26(9):72–82, september 1991.

[Str89] Robert Strandh. Classes of equational programs that compile into efficient machine code. In M. Dershowitz, editor, *Rewriting Techniques and Applications, third international conference*, Lecture Notes in Computer Science, pages 449–461. 1989.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[vENPS90] Marko van Eekelen, Eric Nocker, Rinus Plasmeijer, and Sjaak Smetsers. Concurrent clean. Technical Report 90-20, University of Nijmegen, November 1990. Version 0.6.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[WB90] Dietmar Wolz and Paul Boehm. Compilation of lotos data type specifications. In E. Brinksma, G. Scollo, and C.A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 187–202. Elsevier Science Publishers B.V. (North-Holland), 1990. IFIP, 1990.

[WK] H.R. Walters and J.F.Th. Kamperman. A self-fulfilling prophecy, design and implementation of a compiler for algebraic specifications. to appear.