Introducing TRIM

Tech Report TR01-283

Lutz Hamel
Dept. of Computer Science and Statistics
University of Rhode Island, Kingston, R.I. 02881, USA
email: hamel@cs.uri.edu

July 25, 2001

Abstract

We present TRIM, an abstract machine capable of executing a subset of the algebraic specification language OBJ3. A specifically designed compiler translates the order sorted conditional equations of an OBJ3 specification into TRIM code. The fact that TRIM supports order sorted term rewriting natively differentiates it from many other abstract term rewriting machines. To obtain more efficient abstract machine code we discuss and include two optimizers in our compiler system; a peep-hole optimizer and a rule-base optimizer. The whole TRIM system is seamlessly integrated into the OBJ3 environment. Compiling OBJ3 specifications provides roughly an order of magnitude speed-up over interpreted specifications.

Contents

1 Introduction

Here we present TRIM, an abstract machine capable of executing a subset of the algebraic specification language OBJ3 [9]. The subset of OBJ3 was chosen in such a way that it reflects all the semantic components of the language but omits some of the syntactic constructs present in the current version of OBJ3. A specifically designed compiler translates the order sorted conditional equations of an OBJ3 specification into the abstract term rewriting machine code – the TRIM code.

To obtain more efficient target code we include two optimizers in our compiler system; a peep-hole optimizer and a rule-base optimizer. Peep-hole optimizers improve the efficiency of programs by successively applying behavior preserving optimization rules to the original program. The rule-base optimizer prevents unnecessary attempts of equation application to an input term for which one knows the application will fail. It accomplishes this by splitting the set of equations in an OBJ3 specification into partitions such that, if one equation application in a particular partition fails, then none of the other equations in the same partition are tried.

The compiler, optimizer, and the abstract term rewriting machine have been seamlessly integrated into the OBJ3 environment. The implementation in C++ comprises about 15,000 lines of code. The system provides a speedup of about one order of magnitude over the interpreted OBJ3 equations.

In this paper we assume familiarity with OBJ3 specifications and term rewriting systems. For further discussion of either please refer to the OBJ3 User Manual [9]. The integrated OBJ3/TRIM system can be downloaded from `https://www.kindsoftware.com`.

2 The TRIM Abstract Machine

Abstract machines[1] have proved successful in the implementation of very high level logic and functional programming languages; in particular, we have the G-machine [14] for functional programming languages and the WAM [4, 22] for Prolog. Here we develop an abstract machine appropriate for the implementation of algebraic specification languages.

Our abstract machine is the term rewriting machine TRIM[2] first sketched in the paper [12] and fully developed (including a formal correctness proof) in

[1] Here we have an unfortunate clash of terminology between the term abstract machines as developed in [18] where the term denotes objects with hidden states and visible behavior and the term abstract machine as deployed by language implementors representing abstract instruction sets to be used for the implementation of very high level languages.

[2] TRIM stands for **T**erm **R**ewrite **I**nstruction **M**achine; this is different from the original meaning which was: Tiny Rewrite Instruction Machine. The name change is due to the fact that order sorted and conditional rewriting has added a number of features to the original instruction set so that it is no longer plausible to talk about a *tiny* machine. However, we feel that the acronym **TRIM** is still appropriate, since the native support of order sorted and

the thesis [10]. It is a stack based machine and natively supports order sorted conditional rewriting. This native support of order sorted rewriting is unusual in the sense that most other abstract rewriting machines use some encoding scheme to support order sorted rewriting at the abstract machine code level rather than supporting it as a built-in feature of the machine itself.

As with most abstract machines, the TRIM instruction set reflects the computational characteristics of the source language which it is meant to implement. In this case where the source language is algebraic specifications, we find instructions for primitives such as matching an operator symbol to an input term or binding a term to a variable. However, compared to the semantics of the source language, the semantics of our abstract machine is defined in terms of much more concrete data structures and therefore may be implemented more efficiently on conventional computer architectures.

2.1 The Abstract Machine Instruction Set

TRIM is an abstract term rewriting machine. To convey the flavor of its syntax, let us take a look at how a term rewriting rule could be implemented on this abstract machine.

Example 1 Consider the rewrite rule

```
(var X:Int)  eq X * 1 => X
```

One way to implement this rule on TRIM is[3]:

```
                    ...

                   MATCH * 2 ;
                   JUMPF L5 ;
                   BIND X [ Int ] ;
                   JUMPF L5 ;
                   MATCH 1 0 ;
                   JUMPF L5 ;
                   KILLOP ;
                   GET X ;
                   APPLY ;
L5 :
                    ...
```

The generated code attempts to match the input term by doing a prefix, depth-first traversal of the left-hand side term of the given rewrite rule. In this case, it attempts to first match the `*` operator to the input term, then it tries to bind the variable `X` observing its sort `Int`, and lastly it attempts to match the constant operator `1`. Should a match or binding fail, then control is passed to

conditional rewriting results in tight (or rather trim) abstract machine code.

[3]For simplicity's sake we ignore all kinds of start-up, initialization, and run-down code.

signature	::=	*sort_decl_list sort_rel_list op_decl_list var_decl_list*
sort_decl_list	::=	`sort` *sort_name* . *sort_decl_list*
	|	ϵ
sort_rel_list	::=	`subsort` *sort_name* `<` *sort_name* . *sort_rel_list*
	|	ϵ
op_decl_list	::=	`op` *op_name* : *sort_list* `->` *sort_name* . *op_decl_list*
	|	ϵ
var_decl_list	::=	`var` *var_name* : *sort_name* . *var_decl_list*
	|	ϵ
sort_list	::=	*sort_name sort_list*
	|	ϵ
sort_name	::=	*identifier*
op_name	::=	*identifier*
var_name	::=	*identifier*

Figure 1: A context-free grammar for order sorted signatures.

the code following the rule with the `JUMPF` instruction (short for "jump if false"). Once the input term is matched it is removed with the `KILLOP` instruction and the replacement term is computed. Here this is simply a retrieval of the term bound to the variable `X`. Finally, the replacement term is inserted into the input term via the `APPLY` instruction. □

Let us take a look at the concrete syntax of the TRIM instruction set. A context-free grammar for the syntax appears in Fig. 2. A couple of general remarks; the non-terminal *integer* denotes the predefined class of integers, the non-terminal *identifier* denotes the class of all character strings, and ϵ denotes the empty production. From the grammar we see that a TRIM module consists of a *signature* and a *program*. The productions for the non-terminal *signature* are given in the grammar in Fig. 1; i.e., the grammar for the signature part of our source language. The signature part of a TRIM module is identical to the signature declaration in an OBJ3 module. TRIM programs consist of a sequence of, possibly labeled, instructions. Furthermore, there are four classes of instructions:

init_instr – instructions for book keeping and general maintenance of the internal machine state,

compute_instr – instructions that actually manipulate terms,

mode_instr – instructions which allow the direct manipulation and interrogation of machine states,

jump_instr – instruction which transfer control from one point in a program to another.

We assume that the labels in jump instructions refer to labeled instructions appearing in the program under consideration. In general, it is the responsibility

module	::=	*signature program*
program	::=	*labeled_instr* ; *program* | ϵ
labeled_instr	::=	*label* : *instr* | *instr*
instr	::=	*init_instr* | *compute_instr* | *mode_instr* | *jump_instr*
init_instr	::=	`LINK`
	|	`APPLY`
	|	`LOAD`
	|	`EXIT`
	|	`ABORT`
	|	`RESTORE`
	|	`SAVEOP`
	|	`KILLOP`
	|	`PUSHFRAME`
	|	`POPFRAME`
compute_instr	::=	`MATCH` *op_name integer*
	|	`BIND` *var_name* `[`*sort_name*`]`
	|	`GET` x
	|	`BUILD` *op_name sort_name integer*
	|	`NOP`
mode_instr	::=	`SET` *mode*
	|	`IS` *mode*
	|	`PEEK` *op_name sort_name integer*
jump_instr	::=	`JUMPT` *label*
	|	`JUMPF` *label*
	|	`JUMP` *label*
	|	`CALL` *label*
	|	`RETURN`
mode	::=	`NORMAL`
	|	`MATCHED`
	|	`REDUCED`
	|	`FAILURE`
	|	`DONE`
	|	`NFORM`
label	::=	*identifier*

Figure 2: A context-free grammar for TRIM.

of the assembler to check that this assumption holds.

2.2 The Basic Operational Model of the Abstract Machine

TRIM is a stack based machine and each of the instructions manipulate a fairly complex machine state. A machine state is a tuple with the following elements:

EK – eval stack – contains the term to be reduced.

OPK – operand stack – contains rewritten operands to be used in the reduction of non-nullary operators.

WK – work stack – a scratch area for the machine.

E – variable binding environment.

R – state register – holds states such as *matched*, *reduced*, or *failure*.

B – 'is' register – a boolean flag which some of the instructions set.

RA – 'return address' stack – used in *call*/*return* instructions.

SO – sort order – contains the sort relation declared in the signature.

OT – operator table – contains the operators declared in the signature.

The most unusual aspect of the machine state is the presence of the two tables; the sort order and the operator table. These are necessary in order to be able to maintain the appropriate sort information in terms.

Let us sketch what happens during the execution of a TRIM program. More precisely, let us take a closer look how TRIM effects reductions. In order to gain some insight into the workings of the machine let us consider a simple term rewriting system, namely, the Peano numbers with addition:

```
sort Nat .
op 0 : -> Nat .
op s : Nat -> Nat .
op plus : Nat Nat -> Nat .
var M : Nat .
var N : Nat .
eq plus(M,0) => M .
eq plus(M,s(N)) => s(plus(M,N)) .
```

This is essentially the same rewriting system considered before with the infix addition operation replaced by a prefix `plus` operation. The TRIM code for this term rewriting system looks as follows:

```
--- Generated by TRIMOPT Rev 1.3 -- Sat Nov 25 17:57:35 1995
--- /usr/tmp/cca00633.trm => PEANO.trm

op plus : Nat Nat  -> Nat
op s : Nat  -> Nat
op 0 :  -> Nat

--- prologue code
```

```
BEGINMOD :
        IS NFORM ;
        JUMPF L1 ;
        SAVEOP ;
        JUMP BEGINMOD ;

--- eq plus(M,0) => M .
L1 :
        LINK ;
        RESTORE ;
        MATCH plus 2 ;
        JUMPF L11 ;
        BIND M [ Nat ] ;
        JUMPF L5 ;
        MATCH 0 0 ;
        JUMPF L5 ;
        KILLOP ;
        GET M ;
        APPLY ;
        JUMP L11 ;

--- eq plus(M,s(N)) => s(plus(M,N)) .
L5 :
        RESTORE ;
        MATCH plus 2 ;
        JUMPF L11 ;
        BIND M [ Nat ] ;
        JUMPF L11 ;
        MATCH s 1 ;
        JUMPF L11 ;
        BIND N [ Nat ] ;
        JUMPF L11 ;
        KILLOP ;
        GET M ;
        GET N ;
        BUILD plus Nat 2 ;
        BUILD s Nat 1 ;
        APPLY ;

--- epilogue code
L11 :
        IS DONE ;
        JUMPT L12 ;
        IS FAILURE ;
        JUMPF BEGINMOD ;
        SAVEOP ;
        JUMP BEGINMOD ;
```

```
L12 :
        RESTORE ;
        RETURN ;
```

The TRIM program begins with the signature declaration. Following that, the translated code for the first and the second rewrite rules appears between labels `L1` and `L5`, and between labels `L5` and `L11`, respectively. Besides this, we have some control code which allows the machine to loop until the term has reached an irreducible form.

In order to study this process closer, let the term `plus(s(0),s(0))` be an input term to our program. Now, since the detail and tedium of stepping through the actual TRIM code and observing the changes in the machine state would bury all the interesting aspects of the rewriting process, we will look at it more abstractly. First of all, we ignore everything in the machine state except the *eval stack* and the *operand stack*. Furthermore, we pretend that we are executing the rewrite rules directly. This allows us to talk more about the conceptual ideas behind the operational model rather than every concrete detail of the actual execution of the translated rewrite rules. The following illustrates the basic steps involved in rewriting a term. Below, the left stack represents the eval stack and the right stack the operand stack of the machine state. At start-up time the eval stack is initialized by pushing the function symbols of the input term in prefix order. The operand stack is initialized to the empty stack. The rewriting process proceeds as follows:

Step 1:

0		There are no rules to rewrite the function symbol `0`; push unchanged onto the operand stack.
s		
0		
s		
plus		

Step 2:

s	0	There are no rules to rewrite the function symbol `s`; push onto the operand stack taking into account that `s` is a unary function symbol; thus, we build a term using the terms which are on the operand stack as arguments and push the completed term onto the operand stack.
0		
s		
plus		

Step 3:

0	s(0)	Similar to Step 1, there are no rules to rewrite the function symbol `0`; push unchanged onto the operand stack.
s		
plus		

Step 4:

s	0	Similar to Step 2, there are no rules to rewrite the function symbol `s`; push onto the operand stack taking into account that `s` is a unary function symbol; thus we build the a term using the terms which are on the operand stack as arguments.
plus	s(0)	

Step 5:

plus	s(0)	If you consider that the two terms on the operand stack are the subterms of the `plus` operation symbol on the eval stack, then the second rule applies. The function symbols of the rewritten term are then pushed in prefix order onto the eval stack.
	s(0)	

Step 6:

0		
s		
0		Function symbol `0` cannot be rewritten.
plus		
s		

Step 7:

s	0	
0		
plus		Function symbol `s` cannot be rewritten.
s		

Step 8:

0	s(0)	
plus		Function symbol `0` cannot be rewritten.
s		

Step 9:

plus	0	
s	s(0)	Apply the first rule.

Step 10:

0		
s		Function symbol `0` cannot be rewritten.
s		

Step 11:

s	0	
s		Function symbol `s` cannot be rewritten.

Step 12:

s	s(0)	Function symbol `s` cannot be rewritten.

Step 13:

	s(s(0))	The result term is constructed on the operand stack.

The basic functionality outlined here is very similar to the workings of the abstract rewriting machine developed at CWI in Amsterdam [15]. However, in addition to the basic term rewriting summarized above, our machine also offers native support for order sorted and conditional term rewriting making it a convenient platform for the implementation of order sorted conditional term rewriting systems and consequently also for algebraic specification languages.

3 The Translation Scheme

The translation scheme for our TRIM compiler is given in Appendix A. The scheme is given in an OBJ3-like specification format. Briefly, `phiSyn` is the translation function that takes syntactic constructs of the source language to the syntax of our abstract term rewriting machine. Variables are defined over the appropriate elements of the source syntax. We are not only translating the modules and lists of equations, but also signature declarations.

Here is the translation scheme for a single, unconditional equation:

```
eq phiSyn(LHS => RHS ) =
      phiSyn(LHS) ;
      IS FAILURE ;
      JUMPT get.label.push('EQLABEL) ;
      KILLOP ;
      phiSyn(RHS) ;
      APPLY ;
    get.label.pop('EQLABEL) : NOP .
```

First we generate the match code for the left side of the equation (rewrite rule). Then we generate the test code. If we have a failure we jump to the end of the equation, otherwise we remove the input term and build the appropriate right side term and apply it. For more detail please refer to the appendix.

4 Optimizing the Code

The code generated by our compiler is quite inefficient. This is due to the fact that we simply map each syntactic source language construct into an appropriate target language construct without taking any context into account. Such a simple minded translation scheme leads to many repeated instructions and an inefficient flow of control in the target code. It is well known that the efficiency of the target code generated in this construct-by-construct manner can be dramatically improved by applying local transformation rules to the code [1, 2]. This is usually accomplished by sliding a window over the program code and allowing a set of transformation rules to rewrite the code within this window. This method of optimization is called "peep-hole optimization" and derives its name from the fact that we treat the window like a peep-hole on the code in which we look for optimization opportunities.

Another optimization which we consider here is rule-base optimization. This is an optimization particularly geared towards term rewriting systems. Consider a term rewriting system and an appropriate input term, it should be clear that we should only try to match rewrite rules with left sides which are similar to a particular subterm of the input term. Thus, we may partition the rewrite rules according to their "left side similarities" and when looking for matches we only search for rewrite rules in the appropriate partition cutting down the search space. The rewrite rule partitions are called *rule-bases* and the process of constructing them is called the *rule-base optimization*.

4.1 What is Peep-Hole Optimization?

Assume that we have a programming language where programs are simply sequences of instructions. Furthermore, assume we have a set of code rewrite rules. Now, given a program, peep-hole optimization is accomplished by sliding a window over the program instruction sequence and allowing the rewrite rules to rewrite the instructions within the window. It is typical for this approach that the application of one optimization rule to the code creates new opportunities for other optimization rules in the rule set; thus, this process has an iterative flavor and multiple passes of sliding the window over the instruction sequence should be made until no further rule applications can be identified. A sketch of this process is given in Fig. 3. Here, the sequence of small rectangles represents the program instruction sequence. The set of optimization rules is represented by the set of rules in the big square brackets. The square sliding across the code represents the peep-hole code window. The implementation of the iterative nature is accomplished by conceptually attaching a flag to the peep-hole. This flag is initially set to false. However, every time an optimization rule fires, the value of the flag is set to true. Once the peep-hole reaches the end of the code, the flag is examined. If it is true, then the peep-hole is reset to the beginning of the code, the flag is reinitialized to false, and the whole process is repeated. Should the flag be false once the peep-hole reaches the end of the code, the optimization process terminates.

In general we do not require the window to be contiguous or even of fixed size. We assume that it always is "just the right size" in order to find matches for the optimization rules within the code. Furthermore, the optimization rules should be *reductive* in character in order to be considered optimizing transformations. By this we mean, that they should reduce some aspect of the resources used by the program, such as memory, registers, machine cycles, *etc.*

One of the simplest measures of the resources consumed by a program is the number of instructions in a program. However, this is not always adequate. For example, consider a machine that has both a multiplication and a shift-left instruction. Here, a multiplication by two can be represented by both, the straight forward multiplication instruction or a shift-left-by-one instruction. It is clear that the general multiplication instruction is much more complicated than a simple shift-left, but a simple instruction count on the program will not reveal the difference in efficiency. In cases like this a much more revealing "cost"

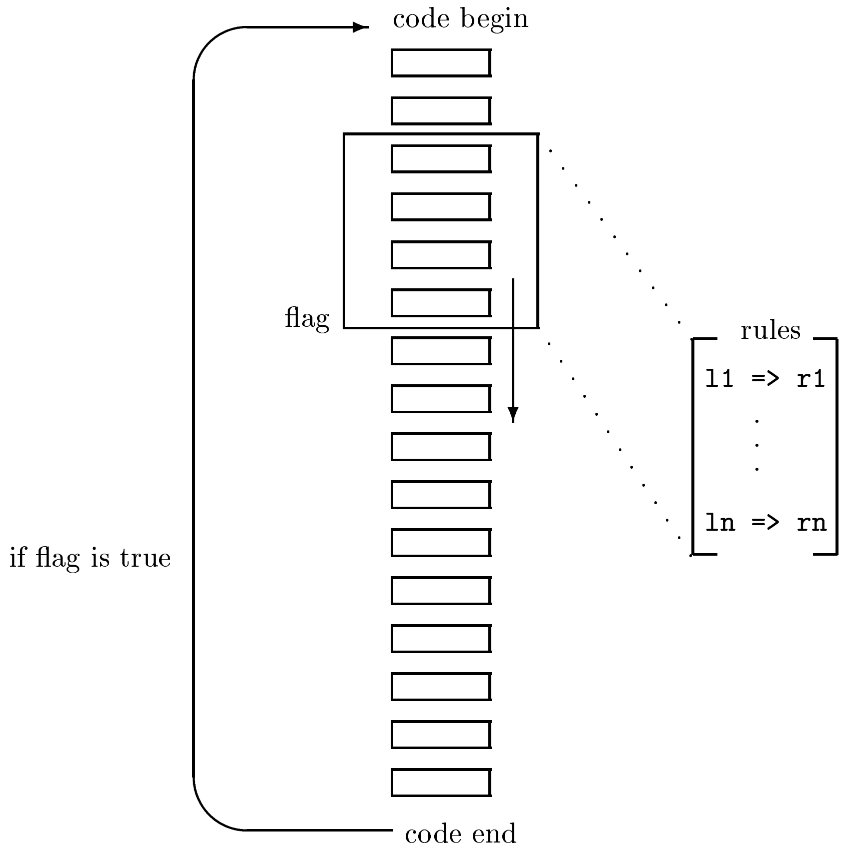

Figure 3: A peep-hole optimizer.

$$
\left.\begin{array}{ll}
 & \texttt{IS FAILURE} \\
 & \texttt{JUMPT}\ \mathit{l1} \\
 & \mathit{instrs1} \\
\mathit{l1}: & \\
 & \texttt{IS FAILURE} \\
 & \texttt{JUMPT}\ \mathit{l2} \\
 & \mathit{instrs2} \\
\mathit{l2}: & \\
 & \mathit{instrs3}
\end{array}\right\} \Rightarrow \left\{\begin{array}{ll}
 & \texttt{IS FAILURE} \\
 & \texttt{JUMPT}\ \mathit{l2} \\
 & \mathit{instrs1} \\
\mathit{l1}: & \\
 & \texttt{IS FAILURE} \\
 & \texttt{JUMPT}\ \mathit{l2} \\
 & \mathit{instrs2} \\
\mathit{l2}: & \\
 & \mathit{instrs3}
\end{array}\right.
$$

Figure 4: The jump retargeting optimization rule.

function needs to be devised; possibly taking machine cycles into account.

From this, it is clear that the reductive character of an optimization rule can be somewhat elusive. On the other hand, in practical peep-hole optimizers, it is usually immediately clear what the optimization rules need to look like and that they are in fact reductive. We therefore assume that the optimization rules which we consider here are reductive.

4.2 A Peep-Hole Optimizer for TRIM

The peep-hole optimizer for our compiler has about half a dozen rewrite rules. Let us take a look at some of the optimization rules in the TRIM peep-hole optimizer. For example, the rule in Fig. 4 attempts to eliminate repeated, unnecessary `IS FAILURE` tests by retargeting the appropriate jump destinations. Here, the instruction `JUMPT L` (short for "jump if true") transfers control to some label `L` if the previous test instruction yields a true in the machine 'is' register. The names *instrs1*, *instrs2*, and *instrs3* represent variables quantified over sequences of instructions, whereas the names *l1* and *l2* are variables quantified over program labels. It should be evident that the right side of the rule represents more efficient code, since on encountering the first `IS FAILURE` test, control is directly passed to the code following label *l2* instead of first jumping to label *l1* and then transferring control to label *l2*.

Let us take a look at another rule. The rule in Fig. 5 removes redundant tests from the source code. As in the previous figure, names in *italic* represent variables. The property which this rule exploits is the fact that the machine state can never be in a failure state immediately after a `RESTORE` instruction. Thus, the `IS FAILURE` test is redundant and we may remove it without changing the behavior of the program. It should be clear that the program resulting from applying this rule is more efficient by the mere fact that it has less instructions.

Let us look at one more rule. The rule in Fig. 6 makes use of the fact

$$\left.\begin{array}{l}\texttt{RESTORE}\\ \texttt{IS FAILURE}\\ \texttt{JUMPT}\ l\\ \textit{instrs}\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\texttt{RESTORE}\\ \textit{instrs}\end{array}\right.$$

Figure 5: Eliminating redundant tests.

$$\left.\begin{array}{l}\texttt{MATCH}\ op\ \ ar\\ \texttt{IS FAILURE}\\ \texttt{JUMPT}\ l\\ \textit{instrs}\end{array}\right\} \Rightarrow \left\{\begin{array}{l}\texttt{MATCH}\ op\ \ ar\\ \texttt{JUMPF}\ l\\ \textit{instrs}\end{array}\right.$$

Figure 6: Rewriting a failure test into a more efficient test.

that the `MATCH` instruction not only generates a failure state if the match is unsuccessful, but also generates the appropriate boolean value in the machine 'is' register. This boolean value may be used in test-and-jump instructions such as `JUMPT` and `JUMPF`. Thus, instead of first explicitly testing for failure we may immediately use the boolean value generated by the match instruction.

4.3 The Rule-Base Optimizer

Consider the following scenario: Given some rewriting system R and an appropriate term t, it should be immediately clear that it is very inefficient to attempt to apply every single rule in R to some subterm of t, let us call it t', in order to reduce t. It would be much more efficient to try to apply only rules in R which look similar to subterm t'. Thus, cutting down on failed match attempts and therefore increasing the rewriting efficiency.

One way to achieve this is by factoring the rules according to their "similarities"; in particular, the similarities of their left sides. This induces partitions, also called *rule bases*, on the set of rules of a rewriting system. Now, should the application of a rule in a particular partition fail due to the fact that its left side is dissimilar to the input term, none of the other rules in this particular partition are tried, since they also would fail for the same reason.

Lets make this a little bit more concrete. Consider the following rewriting system:

```
sort Nat .
op 0 : -> Nat .
op s : Nat -> Nat .
```

```
op _ + _ : Nat Nat -> Nat .
vars M N : Nat .
eq M + 0 => M .
eq M + s(N) => s(M + N) .
```

Given the input term `s(s(0))`, any attempt to apply the rewrite rules to this term will fail. In order to keep these attempts to a minimum we may factor the rules according to their similarities. In this case, both rules share the `+`-sign inducing a single partition on the set of rules. Therefore, if a rule application fails due to the fact that we could not match the `+`-sign to the input term, then there is no need to check the other rule in the partition. Given the input term above, this reduces the attempted matches from six to three, which is a substantial saving.

This process of partioning the set of rules is called *rule-base optimization* and in our compiler it proceeds in two distinct phases. The first phase is the partitioning of the set of rules of a given rewriting system. Here, the rules are partitioned and grouped together according to their top-level left-hand operators. That is, two rules are in the same partition if they share the same top-level left-hand operator. This is done in the compiler front-end before the rules are translated into machine code. The front-end treats the set of rules as a list of rules and rules in the same partition are grouped together into a contiguous sublist. This regrouping of the rewrite rules is legal, since the language of term rewriting systems we are considering here does not put any ordering assumptions on the set of rewrite rules.

Once the rules have been translated into abstract machine code, the second phase of the rule-base optimization is accomplished as part of the peep-hole optimization. Here we try to eliminate redundant rule application attempts within the same partition. Fig. 7 shows the relevant peep-hole optimization rule. The optimization rule makes use of the fact that the top-level left-hand operator of a rewrite rule is the first thing to be matched by the generated code. Should this match fail and the following rewrite rule attempts to match the same operator, i.e., the following source program rewrite rule is in the same partition as the first rewrite rule, then the peep-hole optimizer rewrites the code sequence in such a way, as to skip the second rewrite rule application attempt altogether. This brings about the desired effect: if the first rewrite rule in the sublist of rewrite rules forming a partition fails due to the fact that the discriminating operator could not be matched, none of the other rules in the same partition are tried. Note, because the peep-hole optimizer is iterative, the optimization rule in Fig. 7 is sufficient to optimize rule bases with more than two rules.

5 System Implementation

We take a brief look at the actual implementations of the above components and their integration into the OBJ3 environment. We compare the efficiency of

```
rule1 :                                  rule1 :
          RESTORE                                  RESTORE
          MATCH op ar                              MATCH op ar
          JUMPF rule2                              JUMPF next-rule
          rule1-code                               rule1-code
rule2 :                       =>         rule2 :
          RESTORE                                  RESTORE
          MATCH op ar                              MATCH op ar
          JUMPF next-rule                          JUMPF next-rule
          rule2-code                               rule2-code
next-rule :                              next-rule :
          rest-of–code                             rest-of-code
```

Figure 7: Eliminating unnecessary match attempts in a rule-base.

our implementation with the OBJ3 interpreter itself and an industrial strength equational programming system called UCG-E.

5.1 Implementing the Components

The complete compiler package for OBJ3 consists of four components:

- compiler,
- optimizer,
- assembler,
- TRIM runtime library.

These components are implemented in C++ and comprise about 15,000 lines of code. One interesting aspect of the compiler package which has not been mentioned is the assembler. The assembler maps TRIM instructions into C++ code making use of the translation scheme in Fig. 8. This scheme is quite powerful, since there is no need for actual TRIM instruction interpretation at runtime; TRIM instructions are directly mapped into C++ code and then in turn are translated into native machine code by the appropriate C++ compiler. This means that given a particular OBJ specification module, the compiler package will generate an actual, highly optimized native machine executable for that module.

The more traditional way of looking at this in terms of compiler technology is, that the TRIM machine language is an intermediate representation during the compilation of OBJ3 modules into native machine code providing a stopgap

Assume that for each TRIM instruction, *INSTR*, there exists a C++ function, *instr*, appropriately implementing its semantics:

$$INSTR\ op1\ op2 \ \Rightarrow\ state' = instr(op1, op2, state),$$

where *state* and *state'* are the abstract machine states before and after the execution of the instruction, respectively. Then, given a list of TRIM instructions we may compose the corresponding C++ functions to obtain an implementation of the list:

$$\left.\begin{array}{l} INSTR\ op1\ op2; \\ INSTR'\ op1'\ op2'; \\ INSTR''\ op1''\ op2''; \end{array}\right\} \Rightarrow \left\{\begin{array}{l} state1 = instr(op1, op2, state0); \\ state2 = instr'(op1', op2', state1); \\ state3 = instr''(op1'', op2'', state2); \end{array}\right.$$

Figure 8: The TRIM assembler translation scheme.

between the abstraction levels of the very-high-level OBJ3 specification language and the extremely concrete native machine code.

Our choice of C++ as the implementation language had a direct impact on the overall system architecture. Consider the fact that the OBJ system is written in Lisp and C which does not lend itself for the direct integration of our compiler components written in C++. To understand this we have to briefly consider C++. The C++ language provides the programmer with an object-oriented programming paradigm, thus, it provides high-level programming concepts such as classes and class instantiations, i.e., objects. Because these concepts do not exist in Lisp or C and because C++ compilers rename function calls in order to provide typesafe linkage it is very difficult to incorporate code written in C++ into programs written in other languages[4]. To circumvent this integration problem we decided to run our compiler components as subprocesses to the actual OBJ system. The OBJ system communicates with these subprocesses over UNIX pipes in a specially designed Term Description Language (TDL).

5.2 System Integration

Fig. 9 depicts a very high-level view of the TRIM/OBJ3 system architecture disregarding the fact that the actual compiler consists of various components and runs as a subprocess to OBJ. One can easily see from this diagram that the compiler takes advantage of the OBJ3 module system. The user readable output is also generated by a shared component which we call here the pretty

[4]Since procedural or functional programming may be considered a subset of the C++ object-oriented programming paradigm (after all, in C++ objects communicate via function calls) and one is able to turn off the C++ 'name mangling' facility for external program units, one can easily import code written in either Lisp or C into C++ programs.

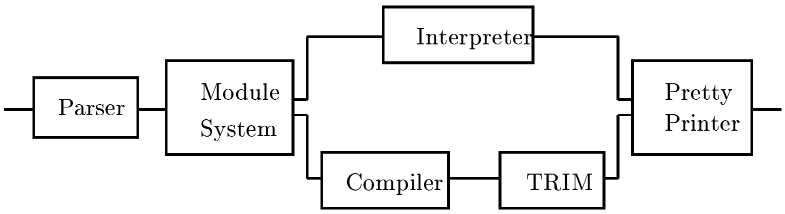

Figure 9: Integration of the TRIM compiler package into OBJ3.

printer. Both the interpreter and the TRIM compiler package coexist within the OBJ3 environment.

Two new commands have been added to the OBJ3 command line interface:

```
run [verbose|keep] <term> .
compile [verbose|noopt|keep] [<module-expression>] .
```

The `compile` command allows the user to compile any module currently loaded into the environment. The `run` command works analogously to the OBJ `reduce` command: given a term, it will attempt to reduce the term using the equations in the appropriate module as rewrite rules. The main difference being, that `run` evokes a compiled version of the module in order to reduce the term. The argument `keep` forces the compiler subsystem to keep any intermediary files it generates. The other arguments are self-explanatory. More information on these and other OBJ3 commands can be retrieved from the command line interface by typing "?" at the command line prompt.

It is never necessary to leave the OBJ3 environment to accomplish any of the tasks necessary to construct an appropriate user environment. All the tasks may be interleaved and repeated in an arbitrary manner. To conclude this section, the following is an example session in the OBJ3 system with the integrated TRIM compiler:

```
client48.comlab% trimobj
                     \|||||||||||||||||/
                   --- Welcome to OBJ3 ---
                     /|||||||||||||||||\
         OBJ3 version 2.03.1 (TRIM) built: 1995 Mar 3 Fri 13:56:40
            Copyright 1988,1989,1991 SRI International
                   1995 Aug 25 Fri 17:00:55
OBJ> in peano
==========================================
set include BOOL off
```

```
==========================================
obj PEANO
OBJ> show PEANO .
obj PEANO is
  sort Nat .
  op _ + _ : Nat Nat -> Nat [strat (2 0 1)] .
  op s : Nat -> Nat .
  op 0 : -> Nat .
  var M : Nat .
  var N : Nat .
  eq M + 0 = M .
  eq M + s(N) = s(M + N) .
endo
OBJ> ***
OBJ> *** use the interpreter
OBJ> ***
OBJ> open PEANO .
OBJ> reduce s(0) + s(0) .
reduce in PEANO : s(0) + s(0)
rewrites: 2
result Nat: s(s(0))
OBJ> close
OBJ> ***
OBJ> *** compile the PEANO module
OBJ> ***
OBJ> compile PEANO .
Warning: operator attributes ignored.
OBJ> ***
OBJ> *** run the PEANO executable
OBJ> ***
OBJ> open PEANO .
OBJ> run verbose s(0) + s(0) .
run PEANO: s(0) + s(0)
PEANO -r  -v  < /tmp/OBJ1871.wr > /tmp/OBJ1871.rd
TRIM Rev 1.2a -- (c) Copyright 1995, Lutz H. Hamel
Program : 'PEANO'
Reductions : 2
CPU Time : 0.016666 sec.
Reductions/sec. : 120.004800
result Nat: s(s(0))
OBJ> close
OBJ> q
Bye.
client48.comlab%
```

Having started a version of OBJ3 which incorporates the interface to the TRIM compiler, the first thing we do is to read in a file with contains a module of interest. In this case, this is a specification of the Peano numbers. The command `in peano` searches for a file '`peano.obj`' in the user's local directory. Once found, the file is read into OBJ3. Next, we list the Peano module just for completeness sake. This is of course not a necessary step when working with the system. One interesting observation here is, that OBJ3 attaches a default reduction strategy attribute to the `+`-operator. Due to this, we will find that the TRIM compiler issues a warning at compile time that it ignores all operator attributes. In the next step we use the OBJ3 interpreter for the simple computation: `s(0) + s(0)`. On the following line the `compile PEANO` command produces a machine executable for the Peano module. Once this is accomplished, we are free to use this executable to effect computations. In this case we `run` the executable on the same input term as the OBJ3 interpreter and observe that we obtain the same result. Note that, since the computation only consists of two rewrites, the actual performance of the abstract machine is quite low due to image startup time and inter-process communication overhead.

5.3 Evaluating the Performance

In order to measure the efficiency of our compiled TRIM abstract machine code we ran the following benchmark program on a SUN SPARC Station 2.

```
obj FIB is
  sort Nat .
  op _ + _ : Nat Nat -> Nat .
  op s : Nat -> Nat .
  op 0 : -> Nat .
  op fib : Nat -> Nat .
  vars M N : Nat .
  eq M + 0 = M .
  eq M + s(N) = s(M + N) .
  eq fib(0) = 0 .
  eq fib(s(0)) = s(0) .
  eq fib(s(s(N))) = fib(N) + fib(s(N)) .
endo
```

The program computes the Fibonacci number of a given natural. The natural numbers are represented in the usual Peano form, e.g., the natural 3 is represented by `s(s(s(0)))`. To get a better feel for the efficiency we compared the performance of the compiled TRIM code with the performance of the OBJ interpreter itself, with the performance of the functional programming language Orwell [5] and with the efficiency of an industrial strength equational programming system called UCG-E [11]. The UCG-E system also compiles an equational specification into C++ code. However, UCG-E only supports single sorted equational specifications. Thus, the comparison might seem somewhat contrived but it allows us to establish some idea of the efficiency of our generated code.

In the following tables the measurements for the TRIM code, the OBJ interpreter, the Orwell language, and the UCG-E system are referred to as the

	fib(10)	fib(15)	fib(20)
OBJ	0.483	5.783	****
ORWELL	0.120	1.440	18.270
TRIM	0.050	0.650	9.350
UCG-E	0.017	0.267	3.850

Table 1: Runtime in CPU Secs.

TRIM, OBJ, ORWELL, and UCG-E rows, respectively. A '****' in a data field indicates that this particular data point could not be measured due to a stack overflow in the Lisp system underlying OBJ. The tables display the data for the calculation of three Fibonacci numbers, namely: `fib(10)`, `fib(15)`, and `fib(20)`.

The first table, Tab. 1, displays the run times in raw CPU seconds for the various systems. Table 2 displays just this ratio of reductions/sec. The compiled TRIM code is roughly ten times faster than the OBJ interpreter and twice as fast as Orwell. The UCG-E system produces code which is three times faster than the TRIM code, six times faster than Orwell, and thirty times faster than the OBJ interpreter. Curiously, the UCG-E system looses speed as the Fibonacci numbers to be computed get larger. We postulate that this is due to memory allocation problems in UCG-E's runtime system.

The last table, Tab. 3, is a study of the impact of the different levels of optimization on the efficiency of the generated TRIM code. High-level optimizations are optimizations implemented in the TRIM compiler, and low-level optimizations are machine specific optimizations within the native C++ compiler. There are a couple of interesting observations here. First, compiling the TRIM code without any optimizations still gives us an efficiency gain of a factor of roughly four over the OBJ3 interpreter. That implies that just moving from the interpreter to the more concrete abstract machine (implemented in an efficient programming language) provides us with a speedup of a factor of four. Second, it is interesting to note that both the high-level optimizations and the low-level optimizations contribute about equally to the final efficiency of the generated code.

	fib(10)	fib(15)	fib(20)
OBJ	1000	1200	*****
ORWELL	4600	5200	5400
TRIM	10000	10000	10000
UCG-E	30000	26000	24000

Table 2: Reductions/Sec.

High	Low	fib(10)	fib(15)	fib(20)
−	−	3800	3800	3700
+	−	6000	5500	5400
−	+	6000	6700	6300
+	+	10000	10000	10000

+/− High – high-level optimization on/off
+/− Low – low-level optimization on/off

Table 3: Impact of Optimizations (reductions/sec.)

6 Related Work

There are two main approaches to compiling term rewriting systems: First, translating a term rewriting system into a subset of a more traditional programming language such as Lisp or Pascal. Second, compiling term rewriting systems onto a specifically designed abstract term rewriting machine.

The first approach was made popular by Kaplan in his seminal paper [16]. Here, Kaplan develops a scheme which translates rewrite rules into Lisp code. The paper also develops a number of optimizations such as common sub-expression elimination. The speed improvements of the order of magnitude of two which Kaplan reports over the interpreted code are quite impressive.

Another approach is taken by Geser, Hussmann, and Mück [7]. They show that by restricting the algebraic specification language to sequential, recursive function definitions they can translate the resulting rewriting system into very efficient Pascal code. They also report impressive speed improvements (a factor of 200).

The work by Heuillard [13] is another interesting approach. He develops a scheme which translates pattern matching into straightforward `IF THEN ELSE` code with the semantics of `IF THEN ELSE` defined as expected. By specifying a number of additional functions and predicates such as subterm selector functions and equality predicates for terms he defines a fairly universal translation scheme for pattern matching, since `IF THEN ELSE` code can be represented with ease in almost any modern programming language[5].

There have been many proposals for abstract term rewriting machines as target machines for the implementation of algebraic specification languages. More notably the abstract machine ARM [15] which is very similar to our TRIM machine with the exception that it does not support native order sortedness. Another important effort is the Equational Machine [20, 21] developed by Sherman and Strandh. By restricting the form of the term rewriting systems induced by an algebraic specification (left linear and orthogonal) and by developing a highly specialized abstract machine instruction set, their approach promises a level of analysis and optimization not witnessed in other approaches. However, we are not quite sure what these restriction mean to the user of the system; do these restriction on the underlying term rewriting system lead to unnatural specifications?

Two other approaches which are related to our own effort in that they also chose abstract stack machines as their target machines are [17] and [19]. However, neither of these machines have native support for order-sorted rewrite rules.

Another project which deserves mention is the Rewrite Rule Machine project at SRI [3, 8]. This is an effort to design an abstract term rewriting machine

[5]It is interesting to note that the UCG-E system [11] employs a very similar scheme which generates `IF THEN ELSE` statements in C++.

running on massively parallel hardware. The project shares our aim to design a machine appropriate for the implementation of OBJ3.

7 Further Work

How can we improve target code efficiency?

Our current abstract term rewriting machine maintains three stacks, and it performs a large amount of book keeping in order to keep these stacks coherent. We propose to go to a simpler abstract machine model where terms are possibly kept on the heap using pointers and pointer arithmetic instead of stacks. The term structure designed for fast rewriting in theorem provers advocated by Christian [6] might be considered here.

The most time consuming activity of any abstract term rewriting machine is pattern matching. Thus, the faster the pattern matching algorithm the faster the machine. A key insight we gained is the fact that the pattern matching process has a lot of structure which optimizers could exploit in order to increase code efficiency. However, because we currently compile to a fairly low-level machine, the inherent high-level structure of pattern matching is lost and our optimizers lose some of the important contexts for powerful optimizations. One possible solution is to stage the compilation process as a succession of compilation steps. Each step compiles the current abstract machine into a slightly lower level abstract machine such that each level provides unique optimization opportunities.

Our current machine maintains variable binding environments. Maintaining these environments during rewriting is very expensive. By realizing that variables are essentially place holders for subterms, the equation parser could compute the subterm addresses of the variables and exploit this knowledge during compilation (this is very much in the same spirit as determining the type of a term at parse time). Once we know the subterm address of a particular variable on the left side of an equation, the compiler may use this to generate code which copies an appropriate subterm directly into a right side instance of that variable. Clearly, this is more efficient than generating code which first assigns a subterm to a variable name in an environment and then fetches the subterm from the environment for each right side instance of that variable name.

Currently unsupported OBJ3 features

Built-ins: There are certain theories for which it is important to have a very efficient implementation. Consider for example the integers or the booleans. OBJ3 allows the user to custom build such theories and integrate them directly into the reduction engine as built-ins. To completely conform to the current OBJ3 programming environment, TRIM and its compiler should support such built-in theories in some way. In particular, built-in boolean operations such as term equality and inequality need to be provided to make conditional equational reasoning more useful with TRIM.

A/C rewriting: During the specification of a particular system it is often convenient to be able to specify certain operators as either associative or commutative, or both. It is in general not possible to specify associativity and commutativity as rewrite rules, since this will give rise to non-terminating term rewriting systems and thus would destroy any possibility of executing the pertinent specification in order to study its runtime behavior. Therefore, associativity and commutativity must be given as operator attributes. Unfortunately, the compilation of term rewriting systems which incorporate such operator attributes is not yet well understood. On the other hand, one might envision compiling associative operators by mapping any input term containing associative operators into its left-associative normal form which is then used during the actual reduction steps. However, we cannot employ the same trick for the compilation of term rewriting systems containing commutative operators, since there is no particular term such as the left-associate normal form which stands out as a candidate to be used as the representative of the commutative equivalence class. On the other hand, one could implement commutative rewriting by lexicographically ordering all subterms.

Evaluation strategies: Depending on the problem at hand it is sometimes convenient to have a flexible way of defining the evaluation strategies for particular operators during the specification of a system. In general, evaluation strategies fall into two classes: *lazy* (also called top-down or outermost) and *eager* (also called bottom-up). OBJ3 allows each operator in a specification to have its own evaluation strategy and the TRIM compiler should support this feature. Currently the compiler simply ignores user defined evaluation strategies and assumes a default, eager evaluation strategy for each operator.

Left-Linearity: OBJ3 allows non-left-linear equations in its specifications which our compiler currently does not support. However, the simple transformation from a non-left-linear equation such as

```
(var N:Int) eq N + (- N) = 0 .
```

into the left-linear equation

```
(vars N M:Int) eq N + (- M) = 0 if N == M .
```

would allow us to compile even non-left-linear equations. This transformation is due to Aida *et al.* [3].

References

[1] Alfred Aho, Ravi Sethi, and Jeff Ullman. Code optimization and finite church-rosser systems. In Randall Rustin, editor, *Design and Optimization of Compilers*, pages 92–105. Prentice Hall, 1972. Fifth Courant Computer Science Symposium, 1971.

[2] Alfred Aho, Ravi Sethi, and Jeff Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[3] Hitoshi Aida, Joseph Goguen, and José Meseguer. Compiling concurrent rewriting onto the rewrite rule machine. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems (CTRS Workshop, Montreal, Canada, June 1990)*, volume 516 of *Lecture Notes in Computer Science*, pages 320–332. Springer-Verlag, 1990.

[4] Hassan Aït-Kaci. The WAM: A (real) tutorial. Technical Report 5, DIGITAL Paris Research Laboratory, 1990.

[5] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[6] Jim Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10:95–113, 1993.

[7] Alfons Geser, Heinrich Hussmann, and Andreas Mück. A compiler for a class of conditional term rewrite systems. *Lecture Notes in Computer Science*, 308:84–90, 1988.

[8] Joseph Goguen. Semantic speicifations for the rewrite rule machine. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture (Workshop, Oxford, UK, September 1989)*, volume 489 of *Lecture Notes in Computer Science*, pages 216–234. Springer-Verlag, 1990.

[9] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000. http://www-cse.ucsd.edu/users/goguen/ps/iobj.ps.gz.

[10] Lutz Hamel. *Behavioural Verification and Implementation of an Optimising Compiler for OBJ3.* PhD thesis, University of Oxford, 1996.

[11] Lutz H. Hamel. UCG-E: An equational logic programming system. In *Proceedings of the Programming Language Implementation and Logic Programming Symposium 1992*, Lecture Notes in Computer Science 631. Springer-Verlag, 1992.

[12] Lutz H. Hamel and Joseph A. Goguen. Towards a provably correct compiler for OBJ3. In Manuel Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, Volume 844, pages 132–146. Springer, 1994.

[13] Thierry. Heuillard. Compiling conditional rewriting systems. *Lecture Notes in Computer Science*, 308:11–128, 1988.

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

[15] J.F.Th. Kampermann and H.R. Walters. ARM – Abstract Rewriting Machine. Technical report, CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, 1994. http://www.cwi.nl/ftp/gipe.

[16] Stéphane Kaplan. A compiler for conditional term rewriting systems. In P. Lescanne, editor, *Rewriting Techniques and Applications*, volume 256 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 1987.

[17] H. Klaeren and K. Indermark. Efficient implementation of an algebraic specification language. *Lecture Notes in Computer Science*, 394:69–89, 1989.

[18] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[19] Karel Richta and Simon Nesvera. The abstract rewriting machine. Research Report DC-91-04, Dept. of Computers Czech Technical University, Prague, September 1991.

[20] David Sherman. EM code semantics, analysis, and optimisation. Technical Report Rapport 94-01, GRECO de Programmation, August 1993.

[21] David Sherman and Robert Strandh. An abstract machine for efficient implementation of term rewriting. Technical Report TR-90-12, University of Chicago, Dept. of Computer Science, March 1990.

[22] David H. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Centre, SRI International, 1983.

A Compiler Translation Scheme

```
var LHS : LTerm .
var RHS : RTerm .
var COND : RTerm .
vars LT1 LT2 : LTerm .
vars RT1 RT2 : RTerm .
var Q : Eq .
var QL : EqList .
vars I K L : Int .
var OP : OpName .
var SO : SortName .
var X : VarName .
vars S1 S2 : SortName .
var R : SortRel .
var RL : SortRelList .
var M : Mod .
var SL : SortList .
var OPS : OpList .

*** make the following global
*** labels for the `call' instr.
op BEGINMOD : -> Label .
op MORPHMOD : -> Label .

*** order-sorted modules.
eq phiSyn( RL,OPS,M ) =
   [ phiSyn(RL) ] ; [ phiSyn(OPS) ] ;
     phiSyn(M) .
eq phiSyn( OPS,M ) =
   [ phiSyn(OPS) ] ; phiSyn(M) .

*** sort relations - identity
eq phiSyn(RL) = RL .

*** operator declarations - identity
eq phiSyn(OPS) = OPS .

*** sort lists - identity
eq phiSyn(SL) = SL .

*** modules.
*** empty module
eq phiSyn({ }) =
      EXIT .

*** regular module
eq phiSyn({ QL }) =
    BEGINMOD :
      IS NFORM ;
      JUMPF get.label.push.reset('CONTMOD) ;
      SAVEOP ;
      JUMP BEGINMOD ;
    get.label('CONTMOD) :
      LINK ;
      RESTORE ;
      phiSyn(QL) ;
      IS DONE ;
      JUMPT get.label('ENDMOD) ;
      IS FAILURE ;
      JUMPF BEGINMOD ;
      SAVEOP ;
      JUMP BEGINMOD ;
    get.label.pop('ENDMOD) :
      RESTORE ;
      RETURN .

*** equation lists.
eq phiSyn(Q * QL) =
      phiSyn(Q) ;
      IS REDUCED ;
      JUMPT get.label.push('CONTINUE) ;
      RESTORE ;
      phiSyn(QL) ;
    get.label.pop('CONTINUE) :
      NOP .

*** equations
*** unconditional
eq phiSyn(LHS => RHS ) =
      phiSyn(LHS) ;
      IS FAILURE ;
      JUMPT get.label.push('EQLABEL) ;
      KILLOP ;
      phiSyn(RHS) ;
      APPLY ;
    get.label.pop('EQLABEL) :
      NOP .

*** conditional
eq phiSyn(LHS => RHS if COND) =
      phiSyn(LHS) ;
      IS FAILURE ;
      JUMPT get.label.push('EQLABEL) ;
      PUSHFRAME ;
      phiSyn(COND) ;
      LOAD ;
      CALL BEGINMOD ;
      PEEK 'TRUE 'BOOL 0 ;
      JUMPF get.label('CONDFAIL) ;
      POPFRAME ;
      KILLOP ;
      phiSyn(RHS) ;
      APPLY ;
      JUMP get.label('EQLABEL) ;
    get.label('CONDFAIL) :
      POPFRAME ;
    get.label.pop('EQLABEL) :
      NOP .

*** left side terms
*** variables
eq phiSyn(x.l(X,SL)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      BIND X [ phiSyn(SL) ] ;
    get.label.pop('LABEL) :
      NOP .

*** constants
eq phiSyn(k.l(OP,SO)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      MATCH OP 0 ;
    get.label.pop('LABEL) :
      NOP .

*** unary ops
```

```
eq phiSyn(uop.l(OP,SO,LT1)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      MATCH OP 1 ;
      IS FAILURE ;
      JUMPT get.label('LABEL) ;
      phiSyn(LT1) ;
    get.label.pop('LABEL) :
      NOP .

*** binary ops
eq phiSyn(bop.l(OP,SO,LT1,LT2)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      MATCH OP 2 ;
      IS FAILURE ;
      JUMPT get.label('LABEL) ;
      phiSyn(LT1) ;
      phiSyn(LT2) ;
    get.label.pop('LABEL) :
      NOP .


*** right side terms
*** variables
eq phiSyn(x.r(X,SL)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      GET X ;
    get.label.pop('LABEL) :
      NOP .

*** constants
eq phiSyn(k.r(OP,SO)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      BUILD OP SO 0 ;
    get.label.pop('LABEL) :
      NOP .

*** unary ops
eq phiSyn(uop.r(OP,SO,RT1)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      phiSyn(RT1) ;
      BUILD OP SO 1 ;
    get.label.pop('LABEL) :
      NOP .

*** binary ops
eq phiSyn(bop.r(OP,SO,RT1,RT2)) =
      IS FAILURE ;
      JUMPT get.label.push('LABEL) ;
      phiSyn(RT1) ;
      phiSyn(RT2) ;
      BUILD OP SO 2 ;
    get.label.pop('LABEL) :
      NOP .
```