Implementing Logic Programming Languages By Graph Rewriting

Peter M^{c}Brien

March 1992

The Imperial College of Science, Technology and Medicine
Department of Computing
180 Queen's Gate
London SW7 2BZ

Submitted to the University of London in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Abstract

The use of *graph rewriting* as a computational model for computer programming languages has received considerable attention in the scientific literature over the past decade. Whilst the implementation of functional programming languages by graph rewriting is simple and intuitive, the implementation of logic programming languages is less direct and consequently has been more limited in practice.

After describing some of the problems associated with a 'direct' approach to implementing logic programming by graph rewriting, and comparing the situation to that found when using term rewriting, this thesis sets out to address the problems by proposing two innovations. The first is a modified form of graph rewriting, which supports some features of term rewriting, and directly supports the notion of the logical variable and backtracking on variables. The second is a compiler target language and corresponding abstract machine, which efficiently implements such modified graph rewriting on conventional Von Neumann computer architectures. A proof that the abstract machine correctly implements the compiler target language is given.

The approach is illustrated by demonstrating how it may be used to implement both Prolog and a more novel logic programming language called the PLL. The graph rewriting language and its associated abstract machine have been the subject of an implementation in the 'C' language, and thus the results described within this thesis have been realized in practice. Issues relating to the use of the techniques described on a parallel computer architecture are also briefly dealt with, but no practical work has been conducted in this direction.

Table of Contents

3 : TAM-Rewrite Rules 85

4 : Term & Graph Rewriting Abstract Machine 119

5 : Compilation of TAM-Rewrites to T-Code 147

Introduction

Current approaches to the implementation of high level computer programming languages recognize that there exists a large gap between the machine instruction code of computers on which programs may execute, and the very high level (in terms of abstraction from the machine architecture) of the languages. This gap is usually bridged by providing some intermediate level, which both supports some of the high level constructs and operations of the language being implemented, and is simple enough to make the implementation of the intermediate level a straightforward task on most machine architectures.

Figure I.1 shows how in its most general form the intermediate level acts to abstract away details of various machine architectures from the language implementor, and to abstract away the constructs needed by various languages from the architecture level.

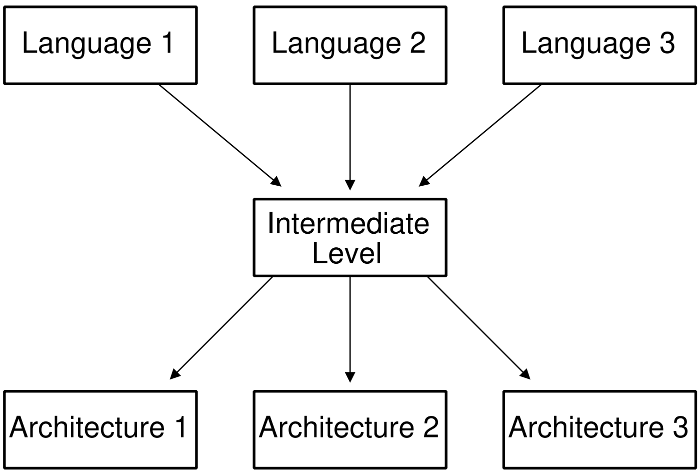


Figure I.1 : Role of the intermediate level

In practice two distinct types of intermediate level have been studied and applied:

(i) ***Abstract Machine Languages*** which provide a form of machine code instruction set for some abstract machine architecture more suited to the implementation of some specific small class of languages. Examples date back to the O-Code for BCPL [Ric71], and more recently WAM [War83] for Prolog, and the G-Machine [Aug84,Joh84] for lazy functional languages. In general an abstract machine will provide for efficient implementations, since the low-level abstract machine instructions are easily compiled to simple sequences of conventional machine instructions, and the abstract machine has been targeted at a very specific language or class of languages. The disadvantage that such machines have is that to support different languages requires considerable effort in redesign of the abstract machine.

In the context of Figure I.1, we would say that the number of languages supported is reduced to approximately one, and the range of architectures supported is that of the Von Neumann (i.e. sequential) type.

(ii) ***Compiler Target Languages*** such as *Dactl* [Gla88a] and *Lean* [Bar87b] provide a relatively high-level at which to aim compilation, in the form of graph rewrite rules. This is advantageous in that graph rewriting is a very general computational model which lends itself to the implementation of a wide class of languages. Language compilers are relatively easy to produce since there is a close relationship between the source and intermediate levels. However the lack of representation for the underlying machine features can make efficient implementation difficult.

These compiler target languages come very near to the 'ideal' of Figure I.1, but at some cost in execution efficiency. The class of languages implemented is very wide, and the range of computer architectures supported includes parallel as well as sequential types.

The work in this thesis advances upon the current position in two ways. Firstly we will extend the mechanics of graph rewriting to provide improved support for the implementation of logic programming. Secondly, a suitable compiler target language supporting the enhanced graph rewriting will be provided, which will compile to an abstract machine code for Von Neumann architecture machines. The intended result is to provide a system allowing for the rapid and efficient implementation of a wide range of logic programming languages.

Before outlining the structure of the thesis, it is worth first describing how we can use graph rewriting for the implementation of logic programming, and hint at some of the problems that will have to be dealt with.

Executing Logic Using Graph Rewrite Rules

To give an overview of how graph rewriting can be used to implement logic programming, this section presents a naive implementation in rewrite rules which provides a simple resolution mechanism for expressions in first order classical logic. The rules will be given in the syntax of the TAM-Rewrite language (introduced in Chapter 3), but will omit certain details of implementation. The TAM-Rewrite language extends the usual rewrite rule to optionally contain a *condition*, so that a rule has the form *head* :- *condition* ---> *body*. The *condition* must be satisfied as part of pattern matching the *head* against any part of the graph (which represents a logical formula) we are rewriting, and the matched graph is referred to as a reducible expression or *redex*.

The rules given below to implement logical equality will illustrate the use of the condition. The first rule will match any function named eq with two arguments (such functions we shall describe as eq/2). It then checks that the first argument x is a variable, and if so, changes all instances of x to become y using the *indirection* operator ':=', and changes the redex to become TRUE. If x was not a variable then we would say the first rule did not hold, and proceed to try any other rules for eq/2. In this case there is just one other rule, dealing with the reflexive case of rule (2). If no rules hold for eq/2 then we would leave the redex unchanged.

```
eq(x,y) :- var(x),x:=y ---> TRUE        (1)
eq(x,y) :- var(y),y:=x ---> TRUE        (2)
```

The following are some examples of the result of rewriting expressions using the above rules. The ! serves to distinguish a query from a rule definition.

!eq(a,1) ---> TRUE	variable a is bound to value 1 using rule (1)
!eq(a,b) ---> TRUE	variable a is bound to variable b using rule (1)
!eq(1,2) ---> eq(1,2)	no rules have been specified to deal with this case

Notice how it is the graphical representation of expressions which allows us to change all instances of a variable a to take value b with a single operation provided we have shared all instances of the variable as one element in the graph. In a term rewrite system this operation would only change the instance of a matched — to achieve the substitution of all instances of a by b would require the entire term to be searched for every occurrence of the a variable.

To implement conjunctions we need only specify the obvious reduction rules for truth values:

```
and(TRUE,x) ---> x                      (3)
and(FALSE,x) ---> FALSE                 (4)
and(x,FALSE) ---> FALSE                 (5)
```

and(x,TRUE) ---> x (6)

The following is an example rewrite sequence using these rules:

!and(eq(a,b),eq(a,1))	
---> and(TRUE,eq(b,1))	Make variable a become variable b using rule (1)
---> eq(b,1)	Reduce the conjunction using rule (3)
---> TRUE	Make variable b become value 1 using rule (1)

The implementation of disjunctions requires more care, since it is possible for a variable to be instantiated to different values in each disjunct. For example, the following rewrite sequence is incorrect with respect to the semantics of the logical expression:

!or(eq(a,1),eq(a,2))	
---> or(TRUE,eq(1,2))	Instantiate variable a with value 1 using rule (1)
---> or(TRUE,FALSE)	Assume rule now added to make eq(1,2) ---> FALSE

From the perspective of the logical semantics of the expression, we want the variable a to have alternative values 1 or 2, but from the graph rewriting perspective we may only assign to it a single value. The method used in Prolog implementations to overcome this problem is to *backtrack* on variable bindings between evaluating each disjunct. This entails recording the action of binding a variable on a *trail*, and providing some mechanism to reverse this binding at a latter stage. One possible implementation of this would be:

```
or(x,y) :- find_level(l) ---> left_or(!x,y,l)                    (7)
left_or(x,y,l) :- return_to_level(l) ---> right_or(x,!y,l)       (8)
right_or(x,y,l) :- return_to_level(l) ---> make_or(x,y)          (9)
make_or(TRUE,x) ---> TRUE                                        (10)
make_or(FALSE,x) ---> x                                          (11)
make_or(x,TRUE) ---> TRUE                                        (12)
make_or(x,FALSE) ---> x                                          (13)
```

The predicates find_level/1 and return_to_level/1 respectively find and return to the *state* or *level* of variable bindings at any particular stage as indicated by the state of the trail. Rule (7) records the state of the trail before rewriting the left disjunct x, after which rule (8) reverses all bindings made during that reduction by a call to return_to_level/1, and then rewrites the right disjunct y. The *activation* marking ! serves to isolate rewriting to one part of the graph, so that only variable bindings made in rewriting x are reversed by return_to_level/1 in left_or/2, and just those for y in right_or/2.

Methods by which the values found in rewriting each disjunct are returned to the user shall be left until we give a description of real implementations in Chapter 3. However, we may see how variable bindings are found and then discarded by looking at how a more complicated use of disjunction is executed using the above set of rules.

!and(eq(a,1),or(eq(a,2),eq(a,b)))	
---> and(TRUE,or(eq(1,2),eq(1,b)))	Bind a to value 1 using rule (1)
---> or(eq(1,2),eq(1,b))	Eliminate conjunction using rule (3)
---> left_or(!eq(1,2),eq(1,b),1234)	Start evaluation of disjunction using rule (7). find_level/1 returns a value which indicates variable state
!eq(1,2) ---> FALSE	Assume that a rule has been defined to do this!
---> right_or(FALSE,!eq(1,b),1234)	Restore any variable bindings made (there were none in this case) using rule (8)
!eq(1,b) ---> TRUE	Bind b to 1 using rule (2)
---> make_or(FALSE,TRUE)	Restore any variable bindings made (b becomes free again) using rule (9)
---> TRUE	Using rule (11)

The rewriting process found that the first disjunct was false, and that the second disjunct was true, with variable substitutions 1/a and 1/b.

The use of rewrites makes the implementation of the common axioms of first order logic very simple. Some examples are:

```
not(and(x,y)) ---> or(not(x),not(y))          (14)
not(not(x)) ---> x                            (15)
and(x,or(y,z)) ---> or(and(x,y),and(x,z))     (16)
```

However there are some subtleties of implementation that must be considered; such as the duplication of x in rule (16). Since we are using graph rewriting, the same graph is shared between the two disjuncts in the body of rule (16). When we rewrite the first disjunct we must ensure that the changes made are equally valid to the second disjunct — something which needs careful study as will be shown in Chapter 2.

Outline of Chapters

Having set out our objectives, general approach and sketching some of the issues to be addressed, we are now in a position to outline the structure of the main body of the thesis.

The material will be divided into five chapters covering the following areas:

Chapter 1: Background

The basis on which the original work in Chapters 2 to 5 was done is described, with appropriate references. Three broad areas are dealt with, in which we describe:

- what a *term rewriting system* (TRS) is, and how a type of *graph rewriting system* (GRS) called a *term graph rewriting system* may be developed from it. It is the latter that will be used as the GRS for the work described in Chapter 2.
- the compiler target languages *Dactl* and *Lean*, which represent practical implementations of graph rewrite systems, in the sense that they have allowed for the implementation of a variety of logic and functional programming languages to be achieved with acceptable execution efficiency. These languages will be used to influence the design of the TAM-Rewrite language introduced in Chapter 3.
- the abstract machines *G-Machine* and *WAM,* which support the efficient implementation of lazy functional and logic programming languages. The abstract machines will be used to influence the design of the TAM and T-Code introduced in Chapter 4.

Chapter 2: Graph Rewriting & Logic Programming Languages

The essential differences between functional programming and logic programming are used to motivate the implementation of logic programming in GRS. Using the term graph rewriting systems reported on in Chapter 1, it is demonstrated how logical variables and unification may be correctly implemented, and some problems with the execution of disjunctions made apparent. The notion of *trailing* and *backtracking* variables and a modified rewrite operator called a *term rewrite on graphs* are introduced to overcome these problems.

Chapter 3: TAM Rewrite Rules

The extensions to GRS introduced in Chapter 2 are included in a graph rewrite language *TAM-Rewrite* (Term and graph rewrite Abstract Machine Rewrites), which is designed to allow for its efficient implementation on sequential machine architectures. The use of the language to implement Prolog and a new logic programming language PLL is described. In so doing, we illustrate the use of the TAM-Rewrite language as an implementation vehicle for logic programming.

Chapter 4: The Term and Graph Rewriting Abstract Machine

An abstract machine and associated instruction set *T-Code* is introduced, to provide the execution platform for the TAM-Rewrite language. The T-Code instruction set is specified in a similar manner to the G-Machine instructions by using state transition rules. The design of T-Code was influenced by that of the G-Machine and WAM, and references to any similarity are made in the text. Like the G-Machine and WAM, the TAM is designed for efficient implementation on sequential machine architectures.

Chapter 5: Compilation of TAM-Rewrites to T-Code

A compilation scheme to translate TAM-Rewrite rules into sequences of T-Code instructions is described. A proof sketch is given showing that the T-Code state transition rules and the compilation scheme together provide a correct implementation of the TAM-Rewrite language.

History and Publication of the Material in this Thesis

This thesis has its origins in work conducted by the author on the Alvey *Pure Logic Language* project at ICL's System Strategy Centre in Bracknell. As part of the project, a new logic programming language *PLL* (the *Pure Logic Language*) based on a simple theorem proving algorithm for first order classical logic was developed, designed to permit the flexible querying of databases. An implementation of the PLL was produced by the author written in ‘C’. This used graphs to store PLL program structures, and computation was effected by rewriting the stored structure. An abstraction of the techniques used in this implementation made in the form of rewrite rules was presented at the *1st Conference of the UK Association for Logic Programming* [McB91a], and forms the basis of the PLL implementation reported on in Chapter 3.

As various possibilities for enhancing the PLL language were explored, it became difficult to keep the ‘C’ implementation in step with the specification of the language. It was then realized that if the abstract specification could be executed directly, then new possibilities in the PLL language could be implemented and tested much more rapidly. This implementation of the abstract specification was achieved by taking the normal definition of graph rewriting found in the literature, as well as associated implementations such as Dactl, and adapting them to produce a definition and implementation conducive to the efficient implementation of logic programming on a sequential machine.

The resulting TAM-Rewrite language, abstract machine and compilation rules have been implemented in ‘C’ as a portable language system, which is documented separately [McB92]. Much of Chapter 2 and the Prolog implementation in Chapter 3 has been published in the proceedings of the *Semagraph ’91 Symposium* [McB91b].

1 Background

In this chapter the areas of research which form the basis of this thesis are surveyed. The various areas will be covered in a 'top-down' ordering, starting with an introduction to the theory and terminology of *term* and *graph rewriting*, then describing some *compiler target languages* which provide a practical implementation of graph rewriting, and finally a description of some *abstract machine* languages which implement graph rewriting or logic programming languages in terms of idealized machine architectures adapted for the language concerned.

1.1 Term Rewrite Systems

Since *Graph Rewrite Systems* (GRS) can be viewed as an implementation of *Term Rewrite Systems* (TRS), and have been shown in some circumstances to be equivalent [Bar87a] it is convenient to first introduce in this section some of the notation, terminology and theorems used for TRS, and then describe in the next section the features of GRS which differentiate them from TRS, and the manner in which they may be regarded as equivalent. In this way it is made apparent what results from the field of TRS are relevant when using GRS. The material presented in this section is largely a synopsis of [Klo92].

1.1.1 Terms and Term Rewrites

Definition 1.1 : Term

For a given TRS there is a countably infinite set of symbols which form an alphabet A of the system. A set $F \subseteq A$ is taken to be used as names of functions (called *functors*), another set $V \subseteq A - F$ is used as names of *variables*, and a third set to be *constants* $C = A - F - V$, i.e. any member of the alphabet which is not a functor or variable. The language T of the TRS is the set of all *terms* that can be formed using the following BNF rules over these constructs:

<term>	::=	<function> | <constant> | <variable>
<function>	::=	<functor>() | <functor>(<arg_list>)
<arg_list>	::=	<term> | <term> , <arg_list>

It should be noted that constants are in many respects no more than a syntactic sugaring for functions with no arguments, so that we may write ‘123’ instead of ‘123()’. However the distinction will prove useful in later discussions regarding the implementation of logic programming.

The number of arguments to a function is called its *arity*, and we either restrict each functor to have a fixed arity, or regard the term $f(t_1,\ldots,t_n)$ as a shorthand for $f/n(t_1,\ldots,t_n)$, and refer to the functor alone as f/n. We say that a functor f/n has arity or *degree n*

Terms which contain no variables are called *closed terms* or *ground terms*, and terms with variables *open terms*. The Herbrand universe H is the set of ground terms of T, where $H \subseteq T$.

Definition 1.2 : Substitution

> A *substitution* σ is a term to term mapping satisfying $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$ for all functors f, and $\sigma(c) = c$ for all constants c. It thus acts to replace the variables v of a term with other terms t, where we write an instance of such a replacement as t/v and say that a substitution is the set of such replacements $\sigma = \{t_0/v_0,\ldots,t_n/v_n\}$.

Some alternative notations in common use are to write t^σ or $\{t_0/v_0,\ldots,t_n/v_n\}t$ instead of $\sigma(t)$.

Definition 1.3 : Term rewrite rules

> A *term rewrite rule* or *term reduction rule* is syntactically defined as
>
> <rule> ::= <term> → <term>
>
> For any rule $r : t \rightarrow s$, t is called the *head* of the rule and s the *body*. We place restrictions on the terms used so that
>
> (i) the *head* may not be a variable, and
>
> (ii) any variable in *body* must also appear in *head*.
>
> A rewrite rule $t \rightarrow s$ describes the set of rewrites $t^\sigma \rightarrow_r s^\sigma$ for all possible substitutions σ. The application of a rewrite rule $t \rightarrow s$ to a term x involves:
>
> (i) finding a subterm t^σ of x for some σ, in which case t is said to *match x*, and we call t^σ the *reduction expression* or *redex.*

(ii) *building* a copy s^σ of s using the same substitution σ used to make t match x. We call s^σ the *contractum.*

(iii) *replacing* the t^σ matched in x with s^σ.

Stated more informally, for a rule $t_p \rightarrow t_b$ and a term t_r, we say that t_p matches t_r if the only difference between the structure of the two terms occurs where there is a variable in t_p. Such variables are assigned the corresponding subterm in t_r, and where the variable appears in t_b the associated subterm of t_r is substituted.

Definition 1.4 : Term rewrite system

A term rewrite system (TRS) is a pair (T,R) where T is the set of terms of the system, and R the set of *rewrite rules* or *reduction rules.*

Example 1.1 : Boolean arithmetic TRS

$A = \{\ true,false,and,or\ \}$
$F = \{\ and,or\ \}$
$V = \emptyset$
$\therefore\ C = \{\ true,false\ \}$

T contains such terms as

true,
and(true,and(false,true))
and(true,false,true).

The following rewrite rules may define R:

$$and(true,true) \rightarrow true \quad (1)$$
$$and(false,true) \rightarrow false \quad (2)$$
$$and(true,false) \rightarrow false \quad (3)$$
$$and(false,false) \rightarrow false \quad (4)$$

The following is an example of a reduction sequence of the system, the redex selected at each step being underlined.

$and(true,\underline{and(false,true)})$
$\rightarrow \underline{and(true,false)}$ (by 2)
$\rightarrow false$ (by 3)

In may be deduced from the definitions that rules which use variables are a short hand for a set of rules with the variables replaced by all possible permutations of the terms of T.

Example 1.2 : Boolean arithmetic TRS using variables in rules

Change TRS of Example 1.1 to have $A = \{ \mathit{true},\mathit{false},\mathit{and},\mathit{or},\boldsymbol{x} \}$ and $V = \{ \boldsymbol{x} \}$. We may restate the rules as:

$$\begin{array}{rcll} \mathit{and}(\mathit{true},\mathit{true}) & \rightarrow & \mathit{true} & (1) \\ \mathit{and}(\mathit{false},\boldsymbol{x}) & \rightarrow & \mathit{false} & (2) \\ \mathit{and}(\boldsymbol{x},\mathit{false}) & \rightarrow & \mathit{false} & (3) \end{array}$$

Rule (2) is a shorthand for the (countable) set of rules

$$\begin{array}{rcl} \mathit{and}(\mathit{false},\mathit{false}) & \rightarrow & \mathit{false} \\ \mathit{and}(\mathit{false},\mathit{true}) & \rightarrow & \mathit{false} \\ \mathit{and}(\mathit{false},\mathit{and}(\mathit{true},\mathit{true})) & \rightarrow & \mathit{false} \\ \mathit{and}(\mathit{false},\mathit{and}(\mathit{true},\mathit{false})) & \rightarrow & \mathit{false} \\ & \mathit{etc} & \end{array}$$

and rule (3) is a shorthand for the set of rules

$$\begin{array}{rcl} \mathit{and}(\mathit{false},\mathit{false}) & \rightarrow & \mathit{false} \\ \mathit{and}(\mathit{true},\mathit{false}) & \rightarrow & \mathit{false} \\ \mathit{and}(\mathit{and}(\mathit{true},\mathit{true}),\mathit{false}) & \rightarrow & \mathit{false} \\ & \mathit{etc} & \end{array}$$

Notice how $\mathit{and}(\mathit{false},\mathit{false}) \rightarrow \mathit{false}$ appears in the set for both (2) and (3), that is there are certain redexes which both rules match. Any pair of rules for which the head of one rule matches a non-variable term in another are said to be *ambiguous, overlapping* or a *critical pair*. Example 1.3 illustrates a TRS with several critical pairs. Note that when comparing a rule with itself for ambiguity we place an additional restriction that the term matched must not be the whole rule head, otherwise all rules would be trivially defined as ambiguous. A more formal definition of critical pairs may be found in [Klo82,Sta80a].

Example 1.3 : A TRS with critical pairs of rules

$$\begin{array}{rcll} \mathit{end}(\boldsymbol{x},\mathit{end}(\boldsymbol{y},\boldsymbol{z})) & \rightarrow & \mathit{end}(\boldsymbol{x},\boldsymbol{z}) & (1) \\ \mathit{or}(\boldsymbol{x},\mathit{true}) & \rightarrow & \mathit{true} & (2) \\ \mathit{or}(\mathit{true},\boldsymbol{x}) & \rightarrow & \mathit{true} & (3) \\ d(\boldsymbol{x}) & \rightarrow & e(\boldsymbol{x}) & (4) \end{array}$$

Rule (1) alone is ambiguous (*self-ambiguous*) since the term *end(1,end(2,end(3,4)))* could be reduced by

$\underline{end(1,end(2,end(3,4)))} \rightarrow end(1,end(3,4))$
or $end(1,\underline{end(2,end(3,4))}) \rightarrow end(1,end(2,4))$

and thus the term *end(2,end(3,4))* is shared between two redexes, and so may be rewritten in different ways.

Rules (2) and (3) are ambiguous since the term *or(true,true)* could be reduced by either rule.

Rule (4) is non-ambiguous since even though it may match terms like *or(true,d(1))* together with rule (3), the redexes only overlap on the part of the term matched by a variable in rule (3).

Additional Notation for Sequences of Rewrites

When talking about rewrites being applied to terms, we often wish to refer to a sequence of rewrites being applied to a term, and so the following notation is introduced.

Reflexive Closure: $a \Rightarrow b$ iff $a \rightarrow b$ or $a = b$
Transitive Closure: $a \rightarrow^{+} b$ iff $a \rightarrow b$ or ($a \rightarrow^{+} c$ and $c \rightarrow^{+} b$)
Transitive Reflexive Closure: $a \Rightarrow^{*} b$ iff $a = b$ or $a \rightarrow^{+} b$

1.1.2 Normal Forms & Confluency

When the expression contains no redexes, no more rules may be applied, and the expression is said to be in a *normal form*. A reduction sequence which reaches a normal form is said to *terminate*, a *non-terminating* reduction sequence is one for which every term a in the sequence there exists a rule $a \rightarrow b$. The system is described as *strongly normalizing* if all reduction sequences terminate, and *weakly normalizing* if only some reduction sequences terminate for every term in the language T.

We would obviously desire any practical TRS to be at least weakly normalizing, and every term in the TRS to have a unique normal form. Such systems are described as *confluent* or having the *Church-Rosser* property. Deciding if a system is confluent is in general a difficult task, but fortunately there are some easily identifiable general classes of rewrite systems for which the property holds. A practical point to note here is that if a system largely complies with the rules for the given class, it is then easy to take the few rules that do not comply and check for confluency ‘by hand’ (see below for an example). Two general classes of systems which are confluent are *subcommutative systems* and *regular systems*.

Subcommutative Systems

A *subcommutative system* is one in which for any ambiguous pair of rules $a \rightarrow b$ and $a \rightarrow c$ there also exists a term d such that $b \Rightarrow^* d$ and $c \Rightarrow^* d$ are part of the system. Such systems may be generated from any TRS using the *Knuth-Bendix* completion algorithm [Knu70]. This may be loosely summarized as iteratively adding rules to the TRS of the form $b \rightarrow c$ or $c \rightarrow b$ wherever the terms b and c are both in normal form and $b \neq c$, until we have satisfied the subcommutative property. In practice the requirement for subcommutative systems proves cumbersome, since the number of rules necessary to achieve the property is likely to be large.

Regular Systems

A *regular system* has two properties:

- *Left-linearity*: no variable may be repeated in the head of a rule. For instance the rule $g(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) \rightarrow \boldsymbol{x}$ is left-linear, but $g(\boldsymbol{x},\boldsymbol{y},\boldsymbol{x}) \rightarrow \boldsymbol{x}$ is not.
- *Non-ambiguity*: there are no ambiguous pairs of rules.

A *weakly regular system* relaxes the non-ambiguity constraint to allow rules which match the same principle functor and have the same body. For the system given in Example 1.3, this would mean rules (2) and (3) are no longer considered ambiguous, but rule (1) still is since one instance of the rule could match an *end* inside the redex of another instance of the rule.

Example 1.3 may also be used to illustrate how to find that a system is confluent 'by-hand'. Only rule (1) breaks the rules for a weakly regular system, but viewed alone it is subcommutative. For example, starting with the term *end(1,end(2,end(3,4)))* the reduction sequences could be either:

$$\underline{end(1,end(2,end(3,4)))} \rightarrow \underline{end(1,end(3,4))} \rightarrow end(1,4)$$

or

$$end(1,\underline{end(2,end(3,4))}) \rightarrow \underline{end(1,end(2,4))} \rightarrow end(1,4)$$

By careful inspection of the behaviour of rule (1) we may satisfy ourselves that the subcommutative property holds for the self-ambiguous rule alone. There can be no conflict between rule (1) and the others, since they involve different functors, and so we may satisfy ourselves that the system as a whole is confluent.

1.1.3 Reduction Strategies

In general, several potential redexes may exist in a term. So far, the order in which we replace the redexes has not been specified, but supplying such an ordering may alter the properties we may state for a TRS. We may describe a redex which contains no other redexes as being *innermost*, and one which is not contained within another redex as being *outermost*. A redex which does not have a redex lexicographically wholly to its left is described as *leftmost*. Using this terminology, two *reduction strategies* that are commonly referred to in the literature are:

- *Applicative-order* which states that the leftmost-innermost redex is always taken first, and
- *Normal-order* which states that the leftmost-outermost redex is always taken first.

The choice of reduction ordering (e.g. normal-order) together with class of system (e.g. regular) will often determine whether a TRS is confluent and strongly or weakly normalizing. For a comprehensive survey [Klo92] should be consulted, but two important results should be stated here. Firstly, the *standardization theorem* of λ-calculus (which states that normal-order reduction is strongly normalizing if any reduction ordering is) does not hold for TRS. Secondly, a regular system is *weakly innermost normalizing* only if the system is also strongly normalizing. Since applicative ordering is the characteristic which makes a functional language *strict*, it follows that all such languages (e.g. Hope) require program rules to be strongly normalizing if the execution is to be assumed confluent.

1.2 Graph Rewrite Systems

In this section a definition of what is meant by a *graph* and *graph rewrite system* is given, and a specialization of graphs known as a *tree* is introduced, which will be used in making the connection between TRS and GRS. Some of the results given in [Sta80a,Bar87a] are reported on, including the extent to which a GRS may be used to implement a TRS. The definitions of graphs, terms and rewriting given here are based on those found in [Bar87a], tailored to have a closer resemblance to the definitions of terms and term rewriting given in the previous section.

1.2.1 Graphs

What we shall be describing here are strictly speaking *labelled graphs* and *labelled term graphs.* After keeping the distinction for these definitions, we shall in the remainder of this thesis refer to them simply as *graphs*.

Definition 1.5 : Labelled graph

A labelled graph G over an alphabet A is a triple (N,*lab*,*succ*), where A is divided into functors F, variables V and constants C in exactly the same manner as for terms, and:

- N is a finite set of nodes, where each $n \in N$ is identified by its *address* n.
- *lab* is the labelling function $N \rightarrow A$, where we can speak of the member of A chosen as the *label* or *name* of the node. There must be at most one node labelled by any variable $v \in V$.
- *succ* is the successor function $N \rightarrow N^*$, where the ordered set $\{n_1,...,n_k\} \in N^*$ is the set of *successors* or *arguments* of $n \in N$, and k is the *arity* of n. We may refer to a particular argument n_i as $succ(n)_i$, and refer to $succ(n)_i$ as being a *pointer* to n_i.

In a similar naming convention to that used for terms, we describe a graph where there are no nodes labelled as a variable (i.e. there is no $lab(n) \in V$ for all $n \in N$) as a *closed graph* or *ground graph*, and one which does contain variable nodes as an *open graph*. In [Sta80a,Bar87a] a variable is represented by a single *empty node*, which has *lab* undefined. We achieve a similar property by restricting the labelling of nodes with variables, so that each variable name is used at most once. The difference between the two methods is that in Definition 1.5 the labelling function returns the name of the variable, instead of it being undefined. This has the advantage that the graphs will syntactically appear more like terms, and corresponds to the implementation introduced in Chapter 3.

The definition of a graph allows common subgraphs to be shared, and for *cycles* where a *path* of pointers may be followed from node to node, which eventually returns to the first node.

Definition 1.6 : Path

A path from node n_0 to n_m of a graph G is a list of nodes and argument numbers $(n_0,i_0,...,i_{m-1},n_m)$ where $n_k = \mathrm{succ}(n_{k-1})_{i(k-1)}$ for $1 \leq k \leq m$. The *length* of the path is m.

A graph is *cyclic* if it contains a path of length greater than zero which ends at the node it started from, all such nodes being called *cyclic nodes*. An *acyclic* graph contains no cyclic nodes.

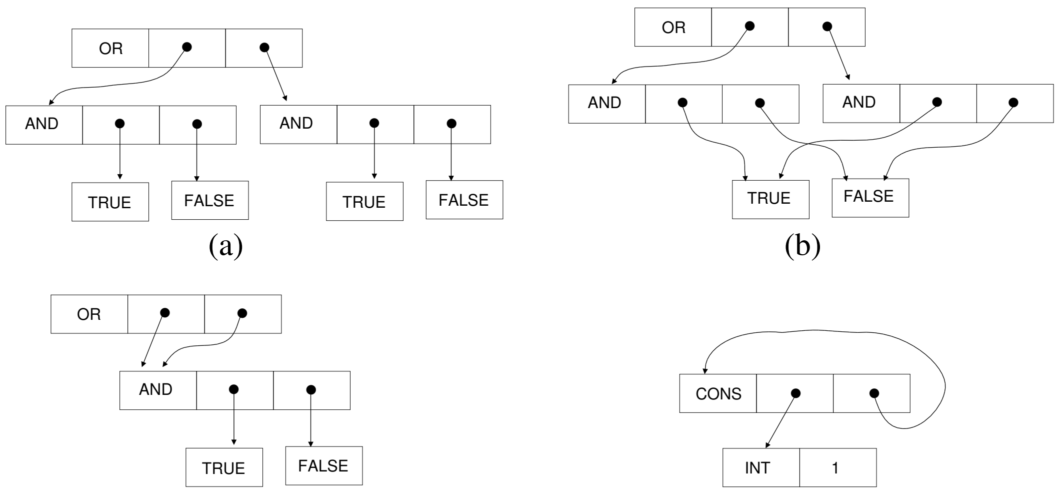
Definition 1.7 : Labelled term graph

> A labelled term graph is defined as a quadruple $(N, lab, succ, r)$ where $(N, lab, succ)$ is a graph and $r \in N$ which we nominate the *root* node of the graph.

Note that the definition of a term graph does not specify that there must be a path from r to all nodes of the graph.

Diagrammatic Representation of Graphs

Graphs may be drawn as diagrams by representing each node as a box with a series of *fields*, the left-most of which is the label of the node, and the remainder have directed *arcs* drawn from each field to any successor nodes returned by the *succ* function. The root node of the graph is placed at the top of the diagram. Figure 1.1 gives some examples of this convention.



(c) (d)

Figure 1.1 : Diagrammatic representation of graphs

Figure 1.1 (c) illustrates the diagram for the graph:

> $N=\{n_1, n_2, n_3, n_4\}$,
> $lab=\{n_1 \rightarrow or, n_2 \rightarrow and, n_3 \rightarrow true, n_4 \rightarrow false\}$,
> $succ=\{n_1 \rightarrow \{n_2\}, n_2 \rightarrow \{n_3, n_4\}\}$,
> $r=n_1$.

Figure 1.1 (a) and (b) show how what is intuitively speaking the same expression as Figure 1.1 (c) can have different graphical forms. Figure 1.1 (d) illustrates a graph which is root cyclic.

Linear Notation and Algebra for Graphs

Since we intend to discuss various manipulations of graphs, it convenient to introduce a small algebra to describe the various structures and operations in a textual format, and thus avoid the need to continuously draw detailed diagrams. Table 1.1 gives such an algebra based on an amalgamation of various conventions used in the literature, for which the only claim made is that it is sufficient for the discussions in this thesis, with no claims made for its expressive completeness.

Construct	*Semantics*
$g_0, g_1, \ldots$	labelled term graphs $(N_0, lab_0, succ_0, r_0), (N_1, lab_1, succ_1, r_1), \ldots$
$g \mid n$	a subgraph g_s of g rooted at node n ($r_s = n$) for which: $N_s = \{n_s \mid$ exists a path from n *to* $n_s\}$, $succ_s \subset$ succ which gives the successors of N_s, $lab_s \subset lab$ which labels members of N_s.
$n \in g$	n is node in graph g
$g_1 - g_2$	a difference subgraph g_d of g_1 and g_2 where $r_d = r_1$ if $r_1 \neq r_2$ and otherwise is undefined, N_d, $succ_d$ and lab_d are made of the difference applied to the respective components of g_1 and g_2.
$g_1 = g_2$	g_1 and g_2 are identical graphs
$g_1 \approx g_2$	g_1 and g_2 are equivalent graphs
$n : g$	the graph g, where n is the root node of graph g.
$f(g_0, \ldots, g_k)$	the graph with a root labelled by f and successors the graphs $g_0, \ldots, g_k$.
$f()$	the graph with a single node, labelled with f and having no successors.
1 2 "hello"	constant nodes
a b c x y z	variables

Table 1.1 : Graph algebra

The graph algebra may be used to describe the graphs of Figure 1.1 as follows:

(a) *or(and(true,false),and(true,false))*
(b) *or(and(**a**:true,**b**:false),and(**a**,**b**))*
(c) *or(**a**:and(true,false),**a**)*
(d) ***a**:cons(1,**a**)*

1.2.2 Graph Rewriting

When dealing with graphs, we have an intuitive notion of graphs such as *f(**x**)* and *f(**y**)* being in some senses equivalent, and *f(**x**)* being a more general version of *f(1)*. The definitions of graph *homomorphism*, *isomorphism* and *equivalence* give a formal method of comparing graphs disregarding the particular variables they contain. This will enable a definition of graph rewriting to be given where the rewrite rules express sets of possible rewrites on graphs using variables, in an analogous way to the method used for term rewrites.

Definition 1.8 : Graph homomorphism, isomorphism and equivalence

> A *homomorphism* is a mapping $h : N_1 \rightarrow N_2$ between two graphs $g_1=(N_1,lab_1,succ_1,r_1)$ and $g_2=(N_2,lab_2,succ_2,r_2)$ which has the properties that for all nodes $n \in N_1$
>
> $$lab_2(h(n))=lab_1(n) \text{ where } lab_1(n) \notin V$$
> $$succ_2(h(n))=h(succ_1(n))$$
>
> We say that $h\{n_1,...,n_k\}=\{h(n_1),...,h(n_k)\}$, and by abuse of notation apply the homomorphism to graphs as a function i.e. $h(N,lab,succ,r)$ gives the graph g_2 formed by applying h to g_1.
>
> An *isomorphism* is any homomorphism h which has an inverse homomorphism h' such that $g_2=h(g_1)$ and $g_1=h'(g_2)$. Thus we may say $h' \circ h=h \circ h'=id$, where id is the identity homomorphism for which $g=id(g)$ is obeyed for any g.
>
> A *rooted homomorphism* is one which obeys $r_2=h(r_1)$, and a *rooted isomorphism* is one where both the forward and inverse homomorphisms are rooted.
>
> Two graphs are *equivalent* if they have a rooted isomorphism, and we write $g_1 \approx g_2$.

The essence of a homomorphism is to make two graphs identical by replacing variables in one graph with any graph valid in the GRS. It can be seen that this is analogous to term substitution which serves to replace variables with any term of a TRS, so we may refer to a homomorphism as a *graph substitution*.

From the definitions it can be seen that two graphs g_1 and g_2 are *isomorphic* only if there exists a pair of homomorphisms between the two graphs such that $g_2=h(g_1)$ and $g_1=h'(g_2)$, where h and h' map variables (i.e. nodes labelled with a member of V) in one graph to variables in the other. If a variable were mapped to a non-variable, then the inverse mapping would have to map a non-variable to a variable, and so fail to be a homomorphism. Thus isomorphic graphs differ only in the nodes labelled by members of V.

Definition 1.9 : Graph rewriting

A graph rewrite rule is a triple (g_r, r_h, r_b) where g_r is an open labelled graph containing nodes r_h and r_b, which are the root nodes of the graphs *head* and *body*. Conventionally we write such a rule as *head* $\rightarrow$ *body*. A variable may not be part of the *body* if it is not part of the *head*.

The application of a graph rewrite rule to a graph g_0 has four steps:

(i) *matching* the rule against the graph g_0, which involves finding a node n a homomorphism h such that $g_0|n = h(head)$ and $body-head = h(body-head)$ except for any instances of the *succ* function that point to elements of the *head.* We call h(*head*) the *redex* and h(*body*) the *contractum.* This homomorphism can be regarded as the ‘minimum’ necessary, since it should leave the parts of the *body* not part of the *head* unchanged.

(ii) the *build phase*, where an isomorphic copy of $h(body-head)$ is made using the same homomorphism used to make *head* equal to the redex during matching. This copy is added to g_0 to give g_1.

(iii) the *redirection phase* updates the *succ* function and possibly the root of g_1 to give graph g_2 so that:

for all n,i if $succ_1(n)_i = h(r_h)$ then $succ_2(n)_i = h(r_b)$ else $succ_2(n)_i = succ_1(n)_i$
if $r_1 = h(r_h)$ then $r_2 = h(r_b)$ else $r_2 = r_1$

(iv) the *garbage collection phase*, where any nodes $n \in N_2$ for which no path exists from r_2 to n are removed from g_2 to give g_3.

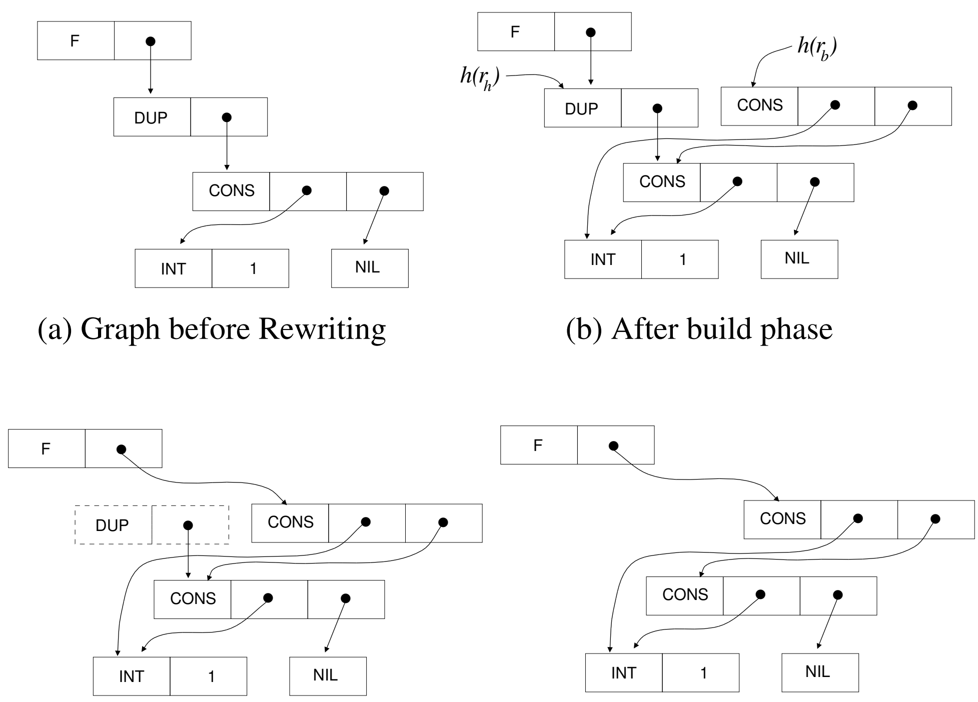
A more precise definition of graph rewriting may be found in [Bar87a,Han91], but this one serves our purposes here. It should be noted that g_r need not have a path to every node from either r_h or r_b, but any such disconnected nodes are ignored by the build phase which only takes a copy of *body*$-$*head*.

The original definition in [Bar87a] speaks of making $r_2 = r_b$ in the redirection phase, but this would seem to work incorrectly for *selector functions* such as $id(\boldsymbol{x}) \rightarrow \boldsymbol{x}$. For such functions we would return as the root of the contractum the variable $\boldsymbol{x}$ in the graph of the rule definition, and not the graph matched by $\boldsymbol{x}$, i.e. $h(\boldsymbol{x})$. Thus when we matched $id(1)$ we would return $\boldsymbol{x}$, and not *1* as intended. In [Han91] the definition was enhanced to make $r_2 = h(r_b)$ if $r_b \in head$, overcoming this problem. In Definition 1.9 we remove the restriction on the body root being part of the head by defining that the homomorphism found during pattern matching is the minimum necessary. Thus in Definition 1.9 we refer to just $h(r_b)$, since the homomorphism gives the identity function if $r_b \notin head$. The actions of the rewrite operation may be most clearly explained by an example.

With the notion of rewrite rule given in Definition 1.9, we can now give a definition of graph rewrite system, similar to that of term rewrite systems.

Definition 1.10 : Graph rewrite system

> A graph rewrite system (GRS) is a pair (G,R) where G is the set of graphs of the system, and R the set of *rewrite rules* or *reduction rules.*



(a) Graph before Rewriting (b) After build phase

(c) After Redirection Phase (d) Graph After Rewriting

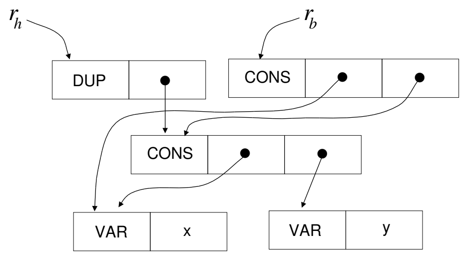
Figure 1.2 : Stages in graph rewriting

Example 1.4 : Application of a graph rewrite rule

We have a rule in our GRS that

*dup(**n**:cons(**x**,**y**))* $\rightarrow$ *cons(**x**,**n**)*

which has the graphical form:



The graph drawn in Figure 1.2 (a) may be written as *f(dup(cons(1,nil)))*, and it can be seen that the subgraph rooted at the *dup* node matches the head of the rewrite rule under a homomorphism *h* that maps the *dup* in the head to the *dup* in the graph, *cons* (pointed at by *dup*) to *cons*, ***x*** to *1* and ***y*** to *nil*. We say that the substitutions are *1/**x*** and *nil/**y***.

The build phase needs to make a copy of those parts of g_r which have no path from r_h, using the same homomorphism *h* that was used for the matching phase. In this case it is just the *cons* node pointed to by r_b which must be copied, and have its pointers to ***x*** and *cons(**x**,**y**)* be made pointers to *1* and *cons(1,nil)* in the new copy. Figure 1.2 (b) shows the resulting graph.

The redirection phase changes all pointers to the redex root to point instead at the contractum root (Figure 1.2 (c)). This leaves some nodes, shown in dotted outline, for which there is no path from the root of the graph. These are no longer of interest and the garbage collecting phase will remove such nodes to result in the final form of the graph (Figure 1.2 (d)).

In constructing the body of a rewrite, it might be thought that one could overwrite the redex root with the body root, and thus negate the need to redirect all the pointers to the redex root. This is indeed the case if the body root is a constant or functor that is not a node in $g \mid h(r_h)$, since the body root can not then be an existing node, and we can create new nodes in any position. However if the body root is a variable or a node in $g \mid h(r_h)$, then producing a copy makes two instances of the root of subgraph it matched exist. Further rewriting may then change one copy and not the other, and thus provide an incorrect implementation of graph rewriting semantics. A further more detailed point is that the implementation may have nodes of different sizes, and so it might not be possible to copy the contents of one node into a smaller node.

Construct	*Semantics*
$[g_1/g_0]g$	The graph which results in rewriting all occurrences of g_0 in g to become g_1
$!g$	The normal form graph that results in a series of rewrites where the first redex chosen is rooted at the root of g
$g_1 := g_0$	All pointers to the root of g_1 are redirected to point at the root of g_0

Table 1.2 : Graph manipulation operators

1.2.3 Trees Representing Terms

Definition 1.11 : Tree

A tree is a graph where there is exactly one path from the root node to all other nodes in the graph, and there are no cyclic nodes.

Figures 1.1(a) and 1.2 (a) give examples of trees. The definition of a tree has the effect of restricting the *succ* function of the graph so that each node except the root will be returned as the successor for exactly one node, and the root will not be the successor for any nodes. This can be seen easily in the diagrams, where every node has only one input arc.

Trees make a natural representation of terms in the context of GRS, since each function, constant and variable of a term will be represented by a node in the corresponding tree, each of which may be manipulated separately by graph rewrite operations. Since the graph rewrite will only ever redirect one pointer (each node only has one pointer in a tree) the operation behaves in the same way on trees as a term rewrite behaves on terms.

To convert a term to a tree, we create nodes of the tree by a recursive descent over the term, beginning with the principle functor of the term:

(i) a constant is represented by a new node labelled with the constant, and having no successors.

(ii) a variable is represented by a new node labelled with the variable, and having no successors.

(iii) a function is represented by a new node labelled with the functor and successor pointers to the tree representation of each of the functors arguments.

To form a graph from a tree we simply merge any equivalent branches of the tree. A simple algorithm for this is to repeatedly merge nodes which give identical results to the *lab* and *succ* functions by redirecting all the pointers from one towards the other. Applied to the tree in Figure 1.1(a), we may merge the nodes representing *1* and *2* to achieve the graph in Figure 1.1(b). This in turn may have both the nodes labelled with *and* merged since they have the same successors, to give Figure 1.1(c). This has no nodes satisfying the criterion for merging, so we stop with Figure 1.1(c) as the graphical form of the tree in Figure 1.1(a).

1.2.4 The Relationship Between TRS and GRS

A TRS may be converted into a GRS by first representing the terms of the TRS as trees (such a system is called a *graphical term rewrite system*), and then converting those trees to graphs. A more formal definition of this process may be found in [Bar87a], which in addition shows that any weakly regular TRS can be replaced by an equivalent GRS, whilst retaining its confluence property. However not all TRS will have an equivalent GRS, or vice versa, as Example 1.5 illustrates.

Example 1.5 : Non-equivalence between TRS and GRS

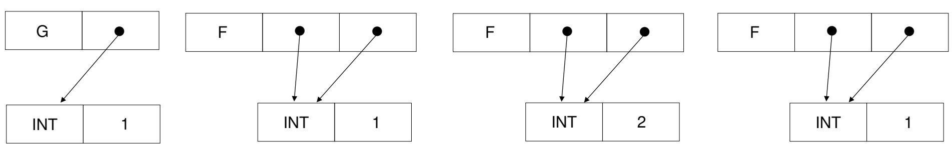
The following set of rewrite rules could be defined for a TRS or GRS:

$$\begin{array}{rcl} f(1,2) & \rightarrow & 0 \\ 1 & \rightarrow & 2 \\ 2 & \rightarrow & 1 \\ g(\boldsymbol{x}) & \rightarrow & f(\boldsymbol{x},\boldsymbol{x}) \end{array}$$

In a TRS, one possible reduction sequence would be (with the redex underlined in each step):

$$\underline{g(1)} \rightarrow f(1,\underline{1}) \rightarrow \underline{f(1,2)} \rightarrow 0.$$

But the equivalent GRS could not have such a sequence of reductions, since the shared node containing the number 1 when rewritten would replace both instances of 1 with 2. Indeed, we can only get the following infinite reduction sequence:



The Difference Between TRS and GRS

We are now in a position to state the key difference between a TRS and GRS, by looking in detail at the use of pointers in the rewrite operation. In the tree representation of a TRS, we limit each node to having just one pointer. Thus if we wish to change the tree by rewriting, it makes no difference if the pointer to a node or the node itself is changed. However, for a GRS we have potentially several pointers to the same node, and thus changing the node has the effect of changing all the pointers simultaneously, whilst changing a pointer (to be directed at a different node) effects only that pointer. The former is what is defined as a graph rewrite, the latter may be coined a *term rewrite on graphs* since it has the same semantics as a term rewrite, but executes on graphs.

The distinction between graph and term rewrites existed (in effect) in an early version of Dactl [Gla87], where one could specify in low level terms exactly how a rewrite operation was made. However, as provided in the language it proved difficult to implement efficiently, and was dropped from subsequent versions of the language. We shall see in Chapter 2 that this is a useful distinction to exploit in the execution of logic programs in a GRS environment, and shall give the concept of a term rewrite on graphs a more formal definition. Chapter 4 provides an instruction set which efficiently implements the two types of rewriting in a common environment.

1.2.5 Advantages of Using a Graphical Representation

The use of TRS or GRS to implement programming languages has the advantage that

(i) We may study the rewrite rules of a language theoretically to prove properties of termination, completeness and optimality of implementation. For example, Staples [Sta80b] has shown that normal-order reduction for the evaluation of a subcommutative GRS can be proved to be optimal.

(ii) The close relationship in computational models between declarative languages and term/graph reduction allows for the rapid implementation of new languages or language variants. The transformation that a compiler from declarative language program to rewrite rules must make is relatively straight forward when compared with the task of producing machine code directly from the declarative language program.

The use of GRS instead of TRS has two advantages:

(i) Graphs are easy to manage on a computer, since they allow for the efficient dynamic allocation of storage to new terms as they are created, and for the reclaiming of storage of deleted terms.

(ii) By representing terms as graphs it is possible to share common subterms as a single graph. This can lead to gains in computational efficiency, since the common terms only need to be evaluated once.

These factors have lead to graph reduction being widely adopted as the implementation technique for functional programming languages [Pey87]. This in turn has led to specialized architectures to support graph reduction. However, the application of graph rewriting to implement logic programming has been much more limited. When considering the implementation of Prolog, it should be noted that the nature of the language makes unification and backtracking a substantial part of the computation. As yet these processes are not well supported by GRS, and implementations of Prolog take a more procedural approach such as the *Warren Abstract Machine* [War83]. Other logic programming languages, such as the PLL [Bab91] and USF [Gab89] have a more production rule type of computational model, and so lend themselves to GRS more readily, but still require unification and backtracking to some extent. Chapter 2 will elaborate on this matter, by discussing how unification and backtracking may be provided in an enhanced version of the type of GRS defined in this chapter.

1.3 Compiler Target Languages

As previously discussed in the introduction, it is usually the case that programming languages are not implemented directly on a particular machine architecture, but instead on some idealized machine more directly supporting the constructs of the language. GRS make a natural choice for such an idealized machine, since the specification of a programming language's operational semantics is often very intuitively expressed in the form of rewrite rules.

To put GRS into use for the implementation of declarative programming languages, various *compiler target languages* (CTL) have been proposed. These CTLs permit the detailed specification of the reduction order used and which terms are selected during rewriting, but abstract away details of how the rewrites are evaluated. The two CTLs discussed here as examples of the approach are *Dactl* and *Lean*, which will serve to influence the design of the TAM-Rewrite language presented in Chapter 3. The general approach in the design of that language will be to remove the constructs of the Dactl and Lean languages which are concerned with their execution on parallel machines, and enhance them with the constructs to support logic programming that are introduced in Chapter 2.

1.3.1 Dactl

Developed originally at UEA as part of the Alvey Flagship project, *Dactl* (Declarative Alvey Compiler Target Language) is the most widely studied practical graph rewrite language so far to appear in the literature. It has undergone several revisions since it was first conceived [Gla87,Gla88a,Gla91], the version described here being Dactl1 [Gla88a] which was released for general use as a stable version of the language. This was specifically aimed at the implementation of both functional and logic programming languages.

In Dactl, the GRS presented in the previous section is enhanced to provide much greater control over the reduction order, pattern matching and creation of graphs in the rewriting process. In addition there is the provision of 'built-in' rules for arithmetic and other common operations.

Reduction Order

The aim in providing precise control over the reduction order is to enable the programmer to specify a reduction strategy, which has both theoretical (q.v. Section 1.1.3) and practical advantages. From the practical point of view, it restricts the nodes which need be considered during pattern matching, and indicates which nodes need to be considered after rewriting. This can be of particular importance for the execution of rewrites on parallel architectures, which Dactl is chiefly targeted at. The node markings *, # and [] describe when it is appropriate to consider a node for rewriting, and the arc markings ^ and Δ describe a process whereby the successful rewriting of a child node may trigger the attempt to rewrite a parent node. The semantics of the operators are outlined in Table 1.3.

Symbol	*Name*	*Description*
*	active	The node so marked may be treated as the root node of a possible redex, and an attempt to find a matching rewrite rule should be made.
#...#	blocked_n	The node so marked must wait for *n* activations from its children nodes. On each activation received from a child node the number of # markings is reduced by one. When the last # is to be removed, it is replaced by a *.
[]	idle	The node may not be the root node of a redex. This is taken as the default if no marking is present on a node.

^	notify	When the child node of an arc is used as the root of a redex, send an activation to the parent node.
Δ	non-notify	Take no action when the child node is used as the root of a redex. This is taken as the default if no marking is present on a node.

Table 1.3 : Dactl control markings

Patterns

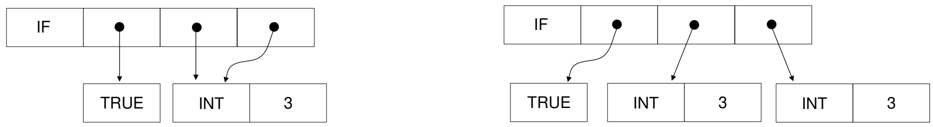
Complex patterns may constructed using binary operators (&, +, ...), general class names (INT, VAR, ...), variables and Dactl graphs. The latter may be duplicated in the rule head, in which case they specify that the same node must be matched by both occurrences of the variable, i.e. they ensure that two parts of a redex are the same subgraph. Table 1.4 describes a representative subset of the patterns allowed in Dactl.

Pattern	*Name*	*Description of Graph Matched*
f[*a b* ...]	function	any function with name *f* and arguments which match *a b* ...
a : *b*	label	*a* is a variable, which is set to the pointer of the graph that *b* matches
a & *b*	conjunction	anything matched by both *a* and *b*
a + *b*	disjunction	anything matched by either *a* or *b*
a – *b*	difference	anything matched by *a* but not *b*
INT		any integer
REAL		any floating point number
STRING		any string
ANY		any graph

Table 1.4 : Dactl patterns

The first or *principal* component in a pattern is called the root of the pattern, and it should match the root of a potential redex. The other patterns are given in a comma separated list.

In addition, the function and label constructs may be used in the description of graphs. Figure 1.3 (a) shows the graph described by z:If[True x:3 x], which shares the same instance of the number 3 since the number was labelled with x and that label used for the number's second occurrence. This may be contrasted with z:If[True x y], x:3, y:3 represented in Figure 1.3 (b) where we have named the number 3 twice, and hence have two distinct subgraphs. Example 1.6 shows how this is of significance in the pattern matching process.



(a) z:If[True x:3 x] (b) z:If[True x y], x:3, y:3

Figure 1.3 : Dactl graphs

Example 1.6 : Dactl Pattern Matching

a:If[(True + False) b b]

The above pattern matches graphs with the root functor If, first argument being either True or False, and the second and third arguments being the same graph. This pattern:

will match	z:If[True x:3 x]	(same number and node x)
	z:If[False x x]	(same node x)
but not	z:If[False x y], x:3, y:5	(not the same number)
	z:If[False x y], x:3, y:3	(not the same node)

Rewriting of Graphs

To provide sufficient control over the rewriting process, and thus to allow for optimization of implementations, the rewrite rules of GRS are extended by providing for general patterns (described above), *redirections* and *activations*. Rules take the following general form, where any or all of the contractum, redirections and activations can be empty:

<rule> ::= <pattern> => [<contractum>], [<redirections>], [<activations>]

A Dactl program will consist of a set of such rules, separated by a vertical bar ‘|’ if they may be considered for pattern matching against a graph simultaneously, and applied in any order, or separated by a semi-colon if they must be considered and applied in a left to right ordering. The various parts of a rule are as follows:

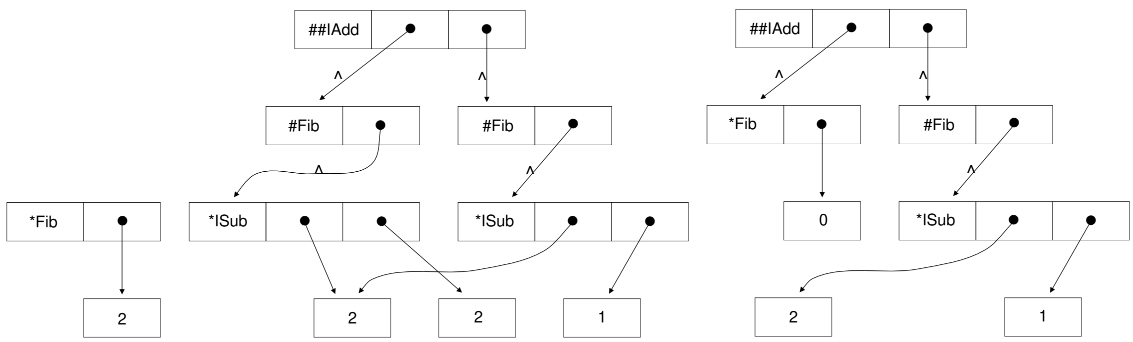
- The *contractum* has a role equivalent to that of the body graph in the GRS described in the previous section. It is described using the Dactl graph description language.

- The *redirections* allows for the redirection of pointers to nodes other than the root node of the redex (which is implicit if the contractum is present). They consist of a comma separated list of assignments x := g, where x is an identifier of some subgraph in the redex matched by the pattern, and g is any graph.

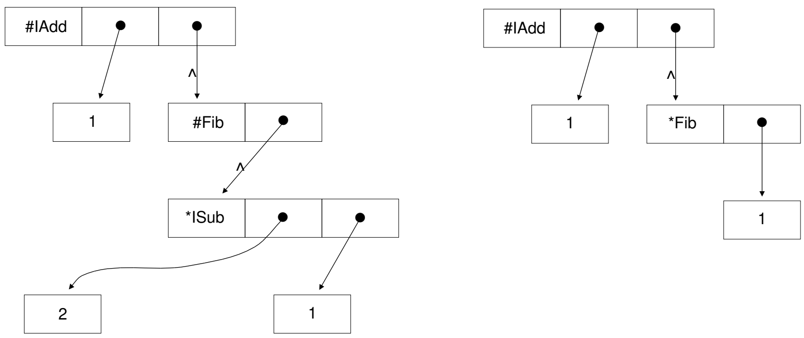
- The *activations* allow for the placing of control markings on various components of the redex and contractum. This makes the pattern matching process more selective, allowing the programmer to control which nodes are considered for rewriting. By placing an activation marker on every node we would achieve the unordered rewriting of 'pure' GRS.

The rewriting process consists of the usual pattern matching, building, redirections and garbage collection phase, with the addition of an activation phase at the end, which controls the process by which other nodes are considered for pattern matching.

Example 1.7 : Dactl rewrite rules to compute Fibonacci numbers



(a) Initial graph (b) Use rule (3) (c) Built-in ISub



(d) Use rule (1) (e) Built-in ISub

(f) Rule (2) (g) Built-in IAdd (h) no rules for 2

Fib[0] => *1; (1)

Fib[1] => *1; (2)

Fib[n:INT] => ##IAdd[^#Fib[^*ISub[n 2]] ^#Fib[^*ISub[n 1]]]; (3)

Fib[n:ANY] => #Fib[^*n] (4)

The functions IAdd and ISub have built-in definitions to perform integer addition and subtraction. Rules (1–3) are the usual definitions that would have been used for term rewrites. By placing ^* on the ISub functions in rule (3), we mean to consider those functions for rewriting, and then activate (i.e. remove the # and replace it with a *) the enclosing Fib after attempting the rewrite. The ## on the IAdd means both enclosed Fib functions must have been considered for rewriting before the IAdd is activated. Rule (4) is only used if all the others fail to match, and causes the execution of the Fib to recommence after the value of n is changed.

The sequence (a) to (h) of graphs below illustrates an execution sequence for the rewrite rules starting with *Fib[2]. It should be noted that there are several possible sequences; for instance the step from (b) to (c) could have been performed after the step from (d) to (e).

1.3.2 Lean

Developed jointly by the University of Nijmegen and the Dactl group at UEA, *Lean* [Bar87b] is a graph rewrite language with close similarities to Dactl. *Lean* has been developed into *Clean* [Koo91], and the two groups working on the languages have since produced *Concurrent Clean* in the Esprit Semagraph project. However, Clean is in many ways a cleaning-up of the Lean and Dactl languages to allow for the efficient implementation of functional languages. Thus from the point of view of this survey it is less interesting to study, since it not such a rich language and is aimed at the ‘wrong’ class of languages. This subsection will concentrate only on introducing those features of Lean which differentiate it from Dactl, other than the obvious syntactic ones.

(i) **Function Currying**

The functions of Lean may be supplied with insufficient arguments, a process know as *currying*. This is used to support those functional languages which provide for currying of functions, and as a convenience to leave out arguments where they are not of concern.

e.g. The following rule defines an apply function which applies a value to the curried form of the binary integer multiply operator *I.

```
Apply (*I a) b ---> *I a b
```

e.g. The following pair of rules define a function which returns TRUE if the argument is an empty list and FALSE if the argument is a non-empty list. Since we are not interested in the arguments of the Cons functor when deciding if it is an empty list, they can be omitted from the rule.

```
IsNil Nil ---> TRUE
IsNil Cons  ---> FALSE
```

(ii) **No Activation Markings**

No control is provided over which nodes may be considered for rewriting. However it is envisaged that the language may be extended either by providing Dactl-type markings, or by specifying that a general evaluation strategy is used.

(iii) **Global Node Identifiers**

By preceding a node identifier with @ a head pattern may match a globally defined node, and rewrite it. This has the obvious advantage (or to some people disadvantage) of allowing for some side effects to occur during rewriting.

e.g. The rule below increments (using the built-in ++I operator) a graph referenced by the global identifier @HdCount on each application of the rewrite rule for the function Head, which returns the head element of a Cons list.

```
r: Head (Cons a b), @HdCount: Total n:INT --->
        r:=a,
        @HdCount := Total (++I n)
```

Clearly the differences between Dactl and Lean enumerated above do not constitute an insurmountable difference in the computational model of the two languages. If so desired, we could sketch out a language with the properties of both as being based on Dactl with the addition of items (i) and (iii) from the above list.

1.4 Abstract Machines

When compiling a high-level language, whether procedural such as BCPL and Pascal, or declarative such as Hope and Prolog, we are usually interested in converting the constructs of the high level language into sequences of instructions in a Von Neumann computer architecture. However the exact details of the latter's instruction set will vary from one machine to another, and thus if we produce compilers which directly produce instruction codes for a particular machine we will be mixing the general issues of compiling a language with the specific issues of implementation on a machine. It was to avoid this confusion that the concept of an *abstract machine* was introduced, the first being O-Code [Ric71] for the BCPL language, more recent examples being the

G-Machine [Aug84,Joh84] for lazy ML and the WAM [War83] for Prolog.

An abstract machine for a particular language provides an architecture and instruction set in the style of Von Neumann architecture, but includes specific features to support the constructs of the language. For example, in lazy ML the need to handle expression graphs is assisted by providing graph manipulation instructions in the G-Code of the G-Machine, and for Prolog the WAM code has instructions to perform unification.

The implementation of abstract machines normally involves translating the abstract instructions to groups of instructions on a conventional von Neumann architecture. However, it is possible to implement abstract machines directly as new purpose designed machines, or translate the instructions into the instructions of a non-conventional (e.g. parallel) machine.

Two illustrative examples are considered in this section; the G-Machine which was developed to support lazy functional languages, and the WAM which was developed to support Prolog. Experience gained from the study of these abstract machines will be used in Chapter 4 to influence the design of the TAM, which supports the TAM-Rewrite language.

1.4.1 G-Machine

Since functional languages lend themselves naturally to implementation by graph reduction, much work has been conducted into the efficient execution of functional languages by graph reduction. The G-Machine was developed at the Chalmers Institute of Technology, Gothenberg, Sweden, by Johnsson and Augustsson [Aug84,Joh84], and summaries have appeared in [Fie88,Pey87] on which the following description is based.

Language Supported by the G-Machine

The language supported by the G-Machine is a functional programming language with lazy semantics, and is summarized by the following abstract BNF syntax (where braces indicate optional expressions which may be repeating):

<supercombinator> ::= <function name> { <identifier> } = <exp>
<exp> ::= <constant> | <identifier> | <function name> | <exp> <exp> |
<locals> in <exp> | letrec <locals> in <exp>
<locals> ::= <identifier> = <exp> | <identifier> = <exp> , <locals>

A *supercombinator* is a lambda expression containing no free variables or other lambda expressions within it, and thus is analogous to a term rewrite rule [Pey87]. A program in the language is made up of a set of supercombinators. For example, a program which generates infinite series of integers would be

```
series n = cons n (series (succ n))
succ n = + n 1
```

where series and succ are supercombinators (which are user defined functions), + a built-in function and n a variable. To execute the program we can add a special supercombinator prog (e.g. prog series 0) which is automatically called when the program is executed. Execution produces the *weak head normal form* (WHNF) of prog, where an expression (or its graphical form) is in WHNF if the principal functor (or root of the graphical form) is not a potential redex.

Abstract Machine

The G-Machine may be represented as a tuple <S, G, C, D> where the four components are as follows:

- S is a stack of pointers to nodes in G
- G is a directed and possibly disconnected graph
- C is a series of G-Code instructions
- D is the dump stack of pairs <S, C>

The dump stack is also used for holding temporary values during the evaluation of arithmetic and character primitives. An empty stack is represented by [], we refer to the top elements $n_0,n_1,...,n_m$ of the stack S as $n_0:n_1:...:n_m:S$, and n_0 is referred to as the top of stack (TOS). Nodes are described by giving the label followed by all fields, for instance:

- INT i is the node containing integer i
- AP n_1 n_2 is the application node with arguments n_1 and n_2
- FUN k C is the functor with k arguments, which is executed by G-Code sequence C

Instruction Code

The instructions presented here give a representative sample of the full instruction set, and provide sufficient basis for an example G-Code program to be given later in the sub-section. The area of optimization is not dealt with, but is fully covered in [Pey87]. The same style of presentation shall be used for the WAM (given in the next sub-section) and for the T-Code instruction set (introduced in Chapter 4), to allow for the ready comparison between the three instruction sets.

The operational semantics of each instruction is given as a *state transition* rule on the tuple representing the machine state. For convenience the instructions are divided into groupings according to their general purpose. Access to the graph G is specified by enclosing in square brackets parts of the graph referenced or updated. Thus the rule $\langle S, G, \mathrm{PUSHINT}\ i{:}C, D\rangle \Rightarrow \langle n{:}S, G[n=\mathrm{INT}\ i], C, D\rangle$ shows that a node n is added to the graph, labelled with INT, and a pointer to n is placed on the stack.

Stack Manipulation

These instructions allow for simple manipulation of entries on the stack without changing the current graph G or the dump D.

PUSH m — Push a copy of the mth element from the TOS onto the stack

$$\langle n_0{:}n_1{:}\ldots{:}n_m{:}S, G, \mathrm{PUSH}\ m{:}C, D\rangle \Rightarrow \langle n_m{:}n_0{:}n_1{:}\ldots{:}n_m{:}S, G, C, D\rangle$$

POP m — Remove the first m items from the stack

$$\langle n_1{:}\ldots{:}n_m{:}S, G, \mathrm{POP}\ m{:}C, D\rangle \Rightarrow \langle S, G, C, D\rangle$$

SLIDE m — Remove the first m items under TOS from the stack

$$\langle n_0{:}n_1{:}\ldots{:}n_m{:}S, G, \mathrm{SLIDE}\ m{:}C, D\rangle \Rightarrow \langle n_0{:}S, G, C, D\rangle$$

Graph Manipulation

These instructions are used in the creation of different types of nodes in G, at the same time adding to or removing from S pointers to the objects in G.

PUSHINT i — Create a node containing integer i in the graph, and push a pointer to it onto the stack

$$\langle S, G, \mathrm{PUSHINT}\ i{:}C, D\rangle \Rightarrow \langle n{:}S, G[n=\mathrm{INT}\ i], C, D\rangle$$

PUSHGLOBAL g — Create a node containing global value g in the graph, and push a pointer to it onto the stack

$$\langle S, G, \mathrm{PUSHGLOBAL}\ g{:}C, D\rangle \Rightarrow \langle n{:}S, G[n=\mathrm{GLOBAL}\ g], C, D\rangle$$

MKAP — Make a function application from the top two elements of the stack

$$\langle n_0{:}n_1{:}S, G, \mathrm{MKAP}{:}C, D\rangle \Rightarrow \langle n{:}S, G[n=\mathrm{AP}\ n_0\ n_1], C, D\rangle$$

UPDATE m — Update the node pointed to by mth stack entry with the contents of TOS (or if using *indirection* nodes in the implementation overwrite the mth node with an indirection to the TOS)

$$<n_0:n_1:\ldots:n_m:S,\ G,\ \text{UPDATE m}:C,\ D>$$
$$\Rightarrow\ <n_1:\ldots:n_m:S,\ G[n_m:=n_0],\ C,\ D>$$

CONS — Make a CONS node from top two elements of the stack

$$<n_0:n_1:S,\ G,\ \text{CONS}:C,\ D>\ \Rightarrow\ <n:S,\ G[n=n_0:n_1],\ C,\ D>$$

Function Call and Return

These instructions provide the mechanism for calling a function (EVAL), putting its arguments onto the stack (UNWIND), and returning from a call (RETURN). Example 1.8 illustrates the rather involved operation of the EVAL and UNWIND instructions. Note that the stack (to the left of the diagram) is conventionally drawn growing downwards to make pointers into the graph easier to draw.

EVAL — We have three situations to consider:

(i) An apply node will be evaluated in a new environment, created by storing the old stack and code sequence on the dump, and executing an UNWIND on the apply node to obtain the arguments on S and code sequence of the object we wish to evaluate on C.

$$<n:S,\ G[n=\text{AP}\ n_0\ n_1],\ \text{EVAL}:C,\ D>$$
$$\Rightarrow\ <n:[],\ G[n],\ \text{UNWIND}:[],\ (S,C):D>$$

(ii) Functions are in general dealt with by the UNWIND instruction. However, parameterless functions may have no parameters applied, and so must be dealt with as a special case by EVAL.

$$<n:S,\ G[n=\text{FUN 0 C'}],\ \text{EVAL}:C,\ D>\ \Rightarrow\ <n:[],\ G[n],\ C':[],\ (S,C):D>$$

(iii) Constants and CONS nodes are already in WHNF, and so need no work to evaluate.

$$<n:S,\ G[n=\text{INT i}],\ \text{EVAL}:C,\ D>\ \Rightarrow\ <n:[],\ G[n],\ C,\ D>$$

UNWIND — This instruction attempts to rewrite the outermost function. We have three situations to consider:

(i) An apply node has its left-hand argument placed onto the stack, and the UNWIND repeated. In this manner the UNWIND instruction follows a *spine* of AP nodes, placing onto the stack pointers direct to each AP node.

$$<\text{v:S, G[v=AP v' n], UNWIND:[], D}>$$
$$\Rightarrow <\text{v':v:S, G[v], UNWIND:[], D}>$$

(ii) A function with enough arguments on the stack converts all its arguments from apply nodes into the value being ‘applied’ to.

$$<v_0:v_1:\ldots:v_m:S,\ G[v_0=\text{FUN m C},\ v_i=\text{AP}\ v_{i-1}\ n_i\ (1\leq i\leq m)],\ \text{UNWIND:[], D}>$$
$$\Rightarrow <n_1:n_2:\ldots:n_m:v_m:S,\ G[v_0],\ C,\ D>$$

(iii) Any integer, CONS node or function with insufficient arguments is in WHNF, and so is returned (with any arguments it has), and the old environment obtained from the dump.

$$<v_0:v_1:\ldots:v_a:[],\ G[v_0=\text{FUN m C' }(a<m)],\ \text{UNWIND:[], (S,C):D}>$$
$$\Rightarrow <v_a:S,\ G[v_0],\ C,\ D>$$
$$<\text{n:[], G[n=INT i], UNWIND:[], (S,C):D}>$$
$$\Rightarrow <\text{n:S, G[n], C, D}>$$

RETURN

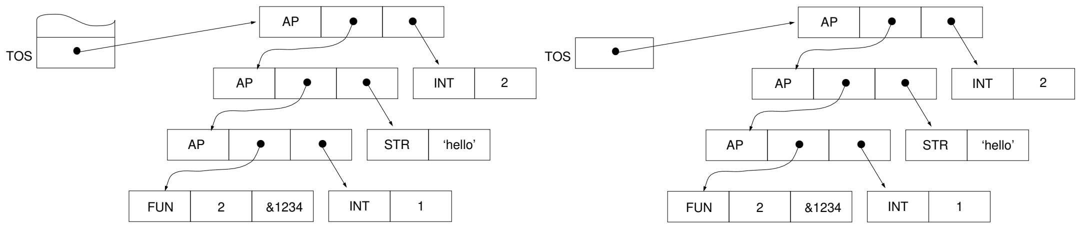
All actions necessary to reduce the supercombinator have been completed. The instruction needs only to restore the old environment and continue evaluation of that, with the result of the computation placed on the TOS of the old stack. (Note that this is the definition from [Pey87]; in [Joh84,Fie88] the G-Code sequence for the bodies of functions ended with a RETURN instead of an UNWIND, and so needed the RETURN instruction additionally to deal with evaluating the body).

$$<v_0:v_1:\ldots:v_m:[],\ G,\ \text{RETURN:[], (S,C):D}> \Rightarrow <v_m:S,\ G,\ C,\ D>$$

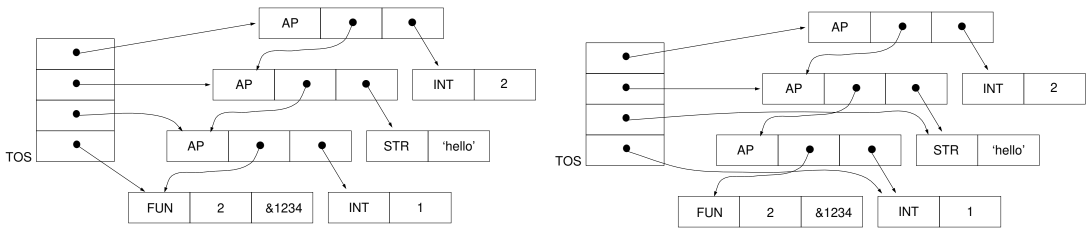
Example 1.8 : Execution of G-Code EVAL instruction on f 1 ‘hello’ 2

Initially (a) we have an apply (AP) node on the TOS. Rule (i) for EVAL causes a new stack to be generated with a single entry pointing to the AP node, and an

UNWIND is initiated giving (b). Successive applications of UNWIND rule (i) to the AP nodes will cause a pointer to each node on the 'spine' being placed on the stack (c).



(a) Before EVAL (b) Just before start of UNWIND



(c) After repeated application of UNWIND (i) (d) After completion of UNWIND

In (c) the supercombinator f is on the top of the stack, and has sufficient AP nodes (two are required by f, and three are present) for UNWIND rule (ii) to hold. Note how all further AP nodes down the stack are left there in case a further function application in the body of f uses them. In (d) the final result on the UNWIND operation is shown, where the root of the redex is followed on the stack by the number of arguments needed for the supercombinator f. In addition a jump will have been made to the G-Code sequence held at memory location &1234.

The sequence of parameters to the supercombinator created by UNWIND on the stack is referred to as its *context*, but no run-time record of where the current context is located on the stack is ever made. References to the context are made by offsets from the TOS, the offset used being computed during compilation.

Compilation Rules

The compilation of programs in the supercombinator language described above into G-Code instruction sequences may be divided into three areas or *schemes* to cater for different parts of the program:

- The **F**-scheme takes a function definition from the program, and produces a G-Code sequence which executes that function.

- The **C**-scheme provides the sequence of instructions needed to construct a graphical representation of any language expression.
- The **E**-scheme compiles expressions for immediate evaluation.

All rules take as an argument the expression to compile. **C**-scheme and **E**-scheme rules take additional parameters to compute the position of arguments (for the supercombinator being evaluated) relative to the top of the runtime stack. These arguments are firstly a function ρ which maps a variable to its position relative to the base of the current context and secondly a number *d* which gives the position of the current context's base from the TOS.

F-*Scheme and* R-*Scheme*: *Function Definitions*

The **F**-scheme assumes that the stack consists of what is produced by the UNWIND instruction. This should be a pointer to the function f with its arguments applied, followed by each argument, so that the first argument is at the TOS. The GLOBSTART pseudo-instruction records the function and arity so that other functions may reference the one being compiled.

$$\mathbf{F}[[\ f\ x_1\ x_2\ ...\ x_n = e\ ]] = \text{GLOBSTART f,n; } \mathbf{R}[[\ e\ ]]\ [x_1{=}n\ x_2{=}n{-}1\ ...\ x_n{=}1]\ n$$

The **R**-scheme produces the instructions necessary to construct the body of the function using the arguments provided (done by the **C**-scheme), update the root of the redex to be the body (UPDATE), and then evaluate it (UNWIND). For the purposes of this discussion it may be defined by a single rule:

$$\mathbf{R}[[\ e\ ]]\ \rho\ d = \mathbf{C}[[\ e\ ]]\ \rho\ d\text{; UPDATE d+1; POP d; UNWIND}$$

C-*Scheme*: *Expression Graph Construction*

$\mathbf{C}[[\ i\ ]]\ \rho\ d = \text{PUSHINT i}$ for any integer i

$\mathbf{C}[[\ x\ ]]\ \rho\ d = \text{PUSH}\ (d - \rho(x))$ for any variable x

$\mathbf{C}[[\ e_1\ e_2\ ]]\ \rho\ d = \mathbf{C}[[\ e_2\ ]]\ \rho\ d;\ \mathbf{C}[[\ e_1\ ]]\ \rho\ (d{+}1);\ \text{MKAP}$

$\mathbf{C}[[\ \text{cons}\ e_1\ e_2\ ]]\ \rho\ d = \mathbf{C}[[\ e_2\ ]]\ \rho\ d;\ \mathbf{C}[[\ e_1\ ]]\ \rho\ (d{+}1);\ \text{CONS}$

$\mathbf{C}[[\ \text{let}\ x_1 = e_1\ \text{in}\ e\ ]]\ \rho\ d = \mathbf{C}[[\ e_1\ ]]\ \rho\ d;\ \mathbf{C}[[\ e\ ]]\ \rho[\ x_1{=}d{+}1\ ]\ d{+}1;\ \text{SLIDE 1}$

$\mathbf{C}[[\ \text{letrec}\ x_1 = e_1\ \text{in}\ e\ ]]\ \rho\ d = \text{ALLOC 1};\ \mathbf{C}[[\ e_1\ ]]\ \rho[\ x_1{=}d{+}1\ ]\ d{+}1\ d;$
$\qquad \text{UPDATE 1};\ \mathbf{C}[[\ e\ ]]\ \rho[\ x_1{=}d{+}1\ ]\ d{+}1;\ \text{SLIDE 1}$

All except the last two rules should be self-explanatory. The rule for let causes code to generated which will construct the parameter e_1 which we want the variable x_1 to take inside e. We may then compile e as usual, having added the expression e_1 to the TOS position in the current context (which is where it will be at runtime) and updated the ρ function to include the new variable x_1. After e has been executed at runtime we must remove the local expression e_1 from the stack (SLIDE).

The rule for letrec needs to allocate the position for e_1 before it is created, since it may use itself recursively. This is achieved by allocating a space for it on the stack (ALLOC) and then compiling it with its own position added to the current context. This must be followed by a instruction to update the hole left with the result generated at runtime (UPDATE). The rest of the compilation is then as for the non-recursive case. If several expressions needed to be added using letrec then each would be enclosed in a pair of ALLOC and UPDATE instructions.

E-*Scheme*: *Expression Graph Evaluation*

To evaluate a graph we may trivially define an **E**-scheme which constructs the graph and then uses the EVAL instruction:

E[[e]] ρ d = **C**[[e]] ρ d; EVAL

However, it is possible to make many optimizations, such as disregarding the EVAL instruction for graphs we can tell are in normal form at compile time (such as integers) and by using special instructions which avoid graph construction to do arithmetic. A large number of optimizations for this and other tasks have been described in [Pey87].

To complete the description of the compilation scheme, Example 1.9 lists the code produced by the rules for a simple supercombinator definition.

Example 1.9 : G-Code for the series supercombinator

The following G-Code sequence would be the result of the compilation rules being applied to **F**[[series n = cons n (series (succ n))]]

GLOBSTART series,1	provides information to the system on what code is for
PUSH 0	make a copy of n on the stack
PUSHGLOBAL succ	push the function succ onto the stack
MKAP	remove above two and replace with pointer to (succ n)
PUSHGLOBAL series	push the function series onto the stack
MKAP	remove above two and replace with (series (succ n))
PUSH 1	make a copy of n on the stack

CONS	pop above two, push cons (n (series (succ n))) onto the stack
UPDATE 2	root of redex is updated to be cons (n (series (succ n)))
POP 1	remove n from stack
UNWIND	start next reduction

Strict and Lazy Evaluation

The G-Machine was designed for the evaluation of *lazy* functional languages, that is languages which do not require the arguments to be in normal form before evaluating a function. However the G-Code instruction set may equally well be applied to the languages with strict semantics. If the **C**-scheme compilation rules were modified to place an EVAL after the code sequence to construct a supercombinator application, then functions would be evaluated as they were created, thus providing strict execution semantics.

1.4.2 WAM

The *Warren Abstract Machine* (WAM) was developed specifically to implement Prolog, and has become the most common implementation method for that language, examples of its use being the implementation of BIM-Prolog and Quintus Prolog. Only a brief overview of the WAM is presented here; a full specification may be found in the original paper [War83] or in the comprehensive tutorial found in [Gab85]. However, the presentation here is made in a style similar to that used for the G-Machine, which is very different from the presentations in [War83,Gab85].

The design of the WAM is based on drawing an analogy between the way Prolog executes, and the execution of procedural languages such as BCPL or Pascal. If we consider a Prolog program such as fib given below, we can view the resolution process as being akin to a modified procedure calling mechanism, where instead of just passing parameters, we unify them with the procedure head, and we have the possibility of *backtracking* where the execution of a procedure is abandoned and another tried if some ‘error’ (logical failure) occurs.

```
fib(1,1).
fib(2,1).
fib(X,Y) :-
        XP is X−1,
        XPP is X−2,
        fib(XP,YP),
```

```
fib(XPP,YPP),
Y is YP+YPP.
```

The WAM provides a modification of the frame based execution system that is used to implement a conventional recursive procedural language. The major change is to cater for the added complication of backtracking, achieved by allowing frames to be left on the stack after the evaluation of a procedure so that when a procedure backtracks, its old frame can be restored. Note that in WAM terminology the whole current frame, which includes an environment of variables that are used in the procedure, is referred to as an *environment*.

Abstract Machine

The abstract machine may be represented as a tuple <C, H, S, T, R> where

- C is the code area, containing the sequence of instructions which form the current program.
- H is the heap containing 'global' structures, which are those items of data which exist outside the scope of the procedure in which they are created, and thus need to be kept outside the local frame for that procedure.
- S is the stack of
  - *environments* which consist of the variables Y_n which appear within a procedure, along with a continuation pointer to another environment and associated code, and
  - *choice points* which hold all the information necessary to restore a previous state in the computation upon backtracking.
- T is the *trail*, which is a stack of pointers to variables bound. By recording in a *choice point* the current top of the trail, backtracking of variable bindings is affected by freeing all variables on the trail above the recorded position.
- R is the set of machine registers, consisting of
  - CP continuation pointer: value of C to restore on succeeding in execution of a clause.
  - E last environment on S: i.e. the environment to use at present
  - B last choice point on S, i.e. the choice point to use at present
  - ST structure pointer into H
  - M current mode, which may be set to read or write
  - A_1, A_2, ... argument registers for passing parameters to procedures and storing temporary variables X_n

Temporary variables X_n are those variables that do not need to be held in the environment, and thus are lost between calls to procedures. Permanent variables Y_n have space allocated for them in the heap along with a place in the environment, and are thus preserved between calls to procedures. Rules for determining which variables are temporary and which global can be found in the references.

Instruction Set

In a similar manner to the presentation of the G-Machine in the previous subsection, the instructions are divided into groupings according to their purpose. It is only intended to present a few key examples. In particular, the instructions to deal with temporary variables are ignored, since they are just optimizations of the instructions which handle permanent variables.

The notation used for the WAM state transition rules introduces a method of recording positions in the stack S in the registers R. For example, the LHS of a rule <C, H, ...:V_2:V_1:E, T, R[E]> serves to use the E register as a reference to a position in the stack, and extract the entries V_1 and V_2 from that position. Note that any number of entries may be above V_2 on the stack.

get

The *get* instructions take the arguments supplied to the procedure and check that they match the parameters of the procedure, performing unification as necessary.

GET_VARIABLE n,A_i

Assign the ith argument to the nth variable in the current environment. This is used to fetch an argument which has a free variable as its pattern into one of the environment variables.

<GET_VARIABLE n,A_i:C, H, V_k:...:V_n:...:V_1:_:_:E, T, R[E,A_i]>
⇒ <C, H, V_k:...:A_i:...:V_1:_:_:E, T, R>

GET_CONSTANT c,A_i

Unify the ith argument to the constant c, calling the FAIL instruction if the two can not unify.

<GET_CONSTANT c,A_i:C, H, S, T, R[A_i=c]>
⇒ <C, H, S, T, R>
<GET_CONSTANT c,A_i:C, H, S, T, R[A_i=VAR]>
⇒ <C, H, S, A_i:T, R[A_i:=c]>

<GET_CONSTANT c,A_i:C, H, S, T, R[A_i]>
⇒ <FAIL, H, S, T, R>

GET_STRUCTURE f,A_i

Unify the ith argument to the structure with functor f. If the ith argument is a variable then we must create a new functor in the heap H and set M to be write so that any following unify instructions add to the partially complete new structure. If the ith argument is not a variable then we want to check the functor is correct, and set M to read mode so that following unify instructions check the structure.

<GET_STRUCTURE f,A_i:C, H, S, T, R[A_i=VAR]>
⇒ <C, H[f(...)], S, A_i:T,
R[A_i:=f(Q_1,Q_2,...), M:=write,ST=Q_1,Q_2,...]>
<GET_STRUCTURE f,A_i:C, H[f(Q_1,Q_2,...)], S, T, R[A_i=f(Q_1,Q_2,...)]>
⇒ <C, H, S, T, R[M:=read, ST=Q_1,Q_2,...]>
<GET_STRUCTURE f,A_i:C, H, S, T, R>
⇒ <FAIL, H, S, T, R>

put

The *put* instructions take values held by a procedure and loads them into the argument registers ready for a new procedure call.

PUT_CONSTANT c,A_i

Set the ith argument to be the constant c. This will be used to set one of the parameters of a new call to be a constant.

<PUT_CONSTANT c,A_i:C, H, S, T, R[A_i]>
⇒ <C, H, S, T, R[A_i:=c]>

PUT_VARIABLE A_k,A_i

Set the ith argument to be the kth argument, and create a variable in the heap which both should reference. This will be used to set one of the parameters of a new call to be a value stored in another argument register, and requires a permanent variable since the value must exist between two distinct calls

<PUT_VARIABLE A_i,A_k:C, H, S, T, R[A_i,A_k]>
⇒ <C, H[V=VAR], S, T, R[A_i:=A_k:=V]>

PUT_VARIABLE Y_k,A_i

Set some variable Y_k from an environment to be the ith argument. This will be used to pass one of the permanent variables from the heap as a parameter to a procedure call.

<PUT_VARIABLE Y_k,A_i:C, H[Y_k], S, T, R[A_i]>
⇒ <C, H, S, T, R[A_i:=Y_k]>

PUT_STRUCTURE f,A_i

Set the ith argument to be a structure with principal functor f newly created in the heap. This should be followed by unify instructions, which will fill in the arguments pointed to by the ST register since the mode M has been set to write.

<PUT_STRUCTURE f,A_i:C, H, S, T, R[A_i]>
⇒ <C, H[f(Q_1,Q_2,...)], S, T, R[A_i:=f(Q_1,Q_2,...),
M:=write, ST=Q_1,Q_2,...]>

unify

The *unify* instructions are used to unify items that already have been the subject of a *get* or *put* instruction.

UNIFY_CONSTANT c

Unify the next argument from S with the constant c. If in write mode then the argument is just updated with c. If in read mode then unification occurs in a similar manner to GET_CONSTANT.

<UNIFY_CONSTANT c:C, H, Q:S, T, R[M=write, ST=Q_1,Q_2,...]>
⇒ <C, H[Q_1:=c], S, T, R[ST:=Q_2,...]>
<UNIFY_CONSTANT c:C, H, S, T, R[M=read, ST=(Q_1=c),Q_2,...]>
⇒ <C, H, S, T, R[ST:=Q_2,...]>
<UNIFY_CONSTANT c:C, H, S, T,
R[M=read, ST=(Q_1=VAR),Q_2,...]>
⇒ <C, H, S, (Q_1:=c):T, R[ST:=Q_2,...]>
<UNIFY_CONSTANT c:C, H, S, T, R[M=read]>
⇒ <FAIL, H, S, T, R>

UNIFY_VARIABLE Y_n

Unify the next argument with a variable. In read mode this just assigns the value to the variable. In write mode a new variable is created in the heap before assigning it to Y_n.

<UNIFY_VARIABLE Y_n:C, H, S, T, R[M=read, ST=Q_1,Q_2,...]>
⇒ <C, H, S, Y_n:T, R[Y_n:=Q_1, ST:=Q_2,...]>
<UNIFY_VARIABLE Y_n:C, H, S, T, R[M=write, ST=Q_1,Q_2,...]>
⇒ <C, H[Q_1:=VAR], S, Y_n:T, R[Y_n:=Q_1, ST:=Q_2,...]>

indexing

The *indexing* instructions deal with the attempting of alternative clauses for a goal. Before trying a sequence of code for a clause, a choice point should be created by TRY_ME_ELSE so that if a FAIL is met then execution can jump to an alternative clause for that goal. That in turn may update the choice point with the next alternative using RETRY_ME_ELSE, or if it is the last alternative for the procedure it should eliminate the choice point using TRUST_ME_ELSE.

TRY_ME_ELSE C' This instruction precedes the WAM code for the first clause of a multi-clause procedure, setting up a choice point so that the environment may be reset on failure, and the next clause coded in C' tried.

<TRY_ME_ELSE C':C, H, S, T, R[E,CP,B]>
⇒ <C, H, A_n:...:A_1:H:T:C':B:CP:E:S, T, R[B:=S]>

RETRY_ME_ELSE C'

This instruction precedes any clause of a procedure bar the first and last. It serves to update the current choice point so that if failure occurs then the next clause of the procedure (coded in C') will be tried.

<RETRY_ME_ELSE C':C, H, ...:C'':_:_:_:B, T, R[B]>
⇒ <C, H, ...:C':_:_:_:B, T, R>

TRUST_ME_ELSE X

This code precedes the final clause of a procedure, and eliminates the choice point for the procedure so that the next FAIL will use the previous choice point. The argument X may be anything; it serves only to pad out the instruction to be the same size as RETRY_ME_ELSE, thus enabling one to add (*assert*) an additional clause by overwriting the TRUST_ME_ELSE instruction in what was the final clause with a RETRY_ME_ELSE instruction.

<TRUST_ME_ELSE X:C, H, ...:B':_:_:B, T, R[B]>
⇒ <C, H, B, T, R[B:=B']>

procedural

The set of *procedural* instructions provide control over the creation and use of environments during the calling and return from procedures. The situation is rather more complicated than that in the G-Machine since we have to deal with the notions of success and failure during evaluation. The ALLOCATE and DEALLOCATE instructions cater for the creation of an environment for a procedure call. The call to a procedure is made using CALL or EXECUTE, and the return from a procedure made by PROCEED or FAIL. The indexing instructions deal with trying alternatives.

ALLOCATE — Allocate a new environment on the stack, recording in it the current environment and continuation pointer. This instruction is used to start a clause with more than one goal.

<ALLOCATE:C, H, S, T, R[E,CP]> ⇒ <C, H, CP:E:S, T, R[E:=S]>

DEALLOCATE — Remove the environment created at the start of a clause with ALLOCATE. Used just before an EXECUTE instruction ends the clause.

<DEALLOCATE:C, H, ...:CP':E':E, T, R[E]>
⇒ <C, H, S', T, R[E:=E',CP:=CP']>

CALL C',n — Call a procedure from which execution will return back to the current clause. The n parameter specifies the number of permanent variables Y_n in the environment that need be kept.

<CALL C',n:C, H, Y_k:...:Y_n:...:Y_1:CP':E':E, T, R[E]>
⇒ <C', H, Y_n:...:Y_1:CP':E':E, T, R[CP:=C]>

EXECUTE C' — Jump to a new WAM code sequence. Used to execute the last goal of a clause.

<EXECUTE C':[], H, S, T, R> ⇒ <C', H, S, T, R>

PROCEED — Successfully complete a unit clause (that is a clause with no body, and thus no goals to EXECUTE or CALL)

<PROCEED:[], H, S, T, R[CP]> ⇒ <CP, H, S, T, R>

FAIL — This instruction is used to code the Prolog *fail* predicate, and in addition is called by any other instruction which fails. All variables on the trail above the location recorded in the current choice point (created by a TRY_ME_ELSE) are set to free, H and C are restored from the values held in the choice point.

<FAIL:C, H, ...:H':T:C':_:_:_:B, V_n:...:V_1:T, R[B]>
⇒ < C', H'[V_1:=free, ..., V_n:=free], ...:H':T:C':_:_:_:B, T, R>

Compilation Rules

The compilation of Prolog programs into sequences of WAM instructions is described in a rather *ad hoc* manner in the references [War83,Gab85]. What is presented here is an attempt at formalizing the techniques given into the same type of compilation rules as are used to define compilation for the G-Machine.

Procedures

The **P** rules compile a sequence of clauses for a single procedure *f/n*. They ensure that the final clause is always preceded by a TRUST_ME_ELSE instruction, that a multi-clause procedure has the first clause preceded by TRY_ME_ELSE, and any intermediate clause preceded by RETRY_ME_ELSE.

P[[$f(a_1,...,a_n)$:- $body_a$. *rest*.]] =
TRY_ME_ELSE L; **C**[[$f(a_1,...,a_n)$:- $body_a$.]]; LABEL L; **PL**[[*rest*]]
P[[$f(a_1,...,a_n)$:- $body_a$.]] = **PL**[[$f(a_1,...,a_n)$:- $body_a$.]]
PL[[$f(a_1,...,a_n)$:- $body_a$. *rest*]] =
RETRY_ME_ELSE L; **C**[[$f(a_1,...,a_n)$:- $body_a$.]]; LABEL L; **PL**[[*rest*]]
PL[[$f(a_1,...,a_n)$:- $body_a$.]] = TRUST_ME_ELSE FAIL; **C**[[$f(a_1,...,a_n)$:- $body_a$.]];

The LABEL pseudo-instruction instantiates its argument with the address where the compiler will place the next instruction that it generates.

Clauses

The **C** rules for clauses ensure that unit clauses end with PROCEED, that clauses with a single goal just EXECUTE that goal, and multiple goal clauses enclose evaluation of all bar the final goal in a pair of ALLOCATE and DEALLOCATE instructions so that the environment of the clause is maintained between successive goals.

The first **C** rule deals with the unit clause, which only requires that its arguments match in order to succeed. A clause with just a single goal needs just an EXECUTE for that goal. If the clause has more than one goal then CALL instructions are required for each goal except the last.

$\mathbf{C}[[\ f(a_1,\ldots,a_n).\ ]] = \mathbf{Get}[[\ a_1,\ldots,a_n\ ]]$; PROCEED
$\mathbf{C}[[\ f(a_1,\ldots,a_n) \text{ :- } g(b_1,\ldots,b_m).]] = \mathbf{Get}[[\ a_1,\ldots,a_n\ ]]; \mathbf{Put}[[b_1,\ldots,b_m]]$; EXECUTE g,m
$\mathbf{C}[[\ f(a_1,\ldots,a_n) \text{ :- } g_1(\ldots),g_2(\ldots),\ldots,g_k(b_1,\ldots,b_m).]] =$
$\mathbf{Get}[[\ a_1,\ldots,a_n\ ]]$; ALLOCATE; $\mathbf{CB}[[\ g_1(\ldots),g_2(\ldots),\ldots,g_k(b_1,\ldots,b_m)\ ]]$;
DEALLOCATE; EXECUTE g_k,m

The **CB** instructions generate the calls for all except the last goal. We rather fudge the issue of how to find the number of variables to be kept (which should be the second argument of a CALL instruction), by providing a *var* function which returns the number of distinct variables in the remainder of the clause from the goal being called onwards.

$\mathbf{CB}[[\ g_1(a_1,\ldots,a_n),g_2(\ldots),\ldots,g_k(\ldots)\ ]] = \mathbf{Put}[[\ a_1,\ldots,a_n\ ]]$;
CALL $g_1,\text{var}(g_1(a_1,\ldots,a_n),g_2(\ldots),\ldots,g_k(\ldots))$; $\mathbf{CB}[[\ g_2(\ldots),\ldots,g_k(\ldots)\ ]]$
$\mathbf{CB}[[\ g(a_1,\ldots,a_n)\ ]] = \mathbf{Put}[[\ a_1,\ldots,a_n\ ]]$;

The method by which **Get** and **Put** generate instructions is left as a matter of implementation of the compiler. To give an approximation of what is required, the following is a sketch of the rules necessary to implement **Get**. When **Get** is supplied with two additional arguments *k* and *a*, the *k* argument specifies the number of arguments to the clause, and *a* specifies which argument is currently being compiled.

$\mathbf{Get}[[\ a_1,a_2,\ldots,a_n\ ]] = \mathbf{Get}[[\ a_1\ ]]\ k\ 1; \mathbf{Get}[[\ a_2\ ]]\ k\ 2; \ldots ; \mathbf{Get}[[\ a_n\ ]]\ k\ n$
$\mathbf{Get}[[\ a_1\ ]]\ k\ a = \emptyset$ — a_1 is a variable
$\mathbf{Get}[[\ c\ ]]\ k\ a =$ GET_CONSTANT c,A_a — c is a constant
$\mathbf{Get}[[\ [a_1,a_2,\ldots,a_n]\ ]]\ k\ a =$ GET_LIST A_a; $\mathbf{Unify}[[\ a_1\ ]]\ k{+}1\ a; \ldots ;$
$\mathbf{Unify}[[\ a_n\ ]]\ k{+}n\ a$; UNIFY_NIL
$\mathbf{Get}[[\ [\ ]\ ]]\ k\ a =$ GET_NIL A_a

$$\textbf{Get}[[\, f(a_1,a_2,...,a_n)\,]]\ k\ a = \text{GET\_STRUCTURE f,}A_a;\ \textbf{Unify}[[\, a_1\,]]\ k{+}1\ a;\ ...\ ;\ \textbf{Unify}[[\, a_n\,]]\ k{+}n\ a$$

A full description of the **Unify** rules will not be given here, but to illustrate the general form the rules will take the rule to deal with variables is given.

$$\textbf{Unify}[[\, v\,]]\ k\ a = \text{UNIFY\_VARIABLE } A_k \qquad v \text{ is a variable}$$

The description of the WAM is concluded by giving in Example 1.10 the WAM instructions to implement the fib predicate given at the start of this subsection. Further examples of compiled code may be found in [War83,Gab85].

Example 1.10 : WAM instructions for fib predicate

C_1:	TRY_ME_ELSE C_2	if the 1st clause fails then jump to the 2nd clause
	GET_CONSTANT 1,A_1	can the 1st argument be unified with the integer 1?
	GET_CONSTANT 1,A_2	can the 2nd argument be unified with the integer 1?
	PROCEED	goal has succeeded, so continue with calling goal
C_2:	RETRY_ME_ELSE C_3	if the 2nd clause fails then jump to the 3rd
	GET_CONSTANT 2,A_1	can the 1st argument be unified with the integer 2?
	GET_CONSTANT 1,A_2	can the 2nd argument be unified with the integer 1?
	PROCEED	goal has succeeded, so continue with calling goal
C_3:	TRUST_ME_ELSE FAIL	if the 3rd clause fails then fail the calling goal
	ALLOCATE	make space to keep information between CALLs
	PUT_STRUCTURE –/2,A_3	put skeleton structure for –/2 onto stack, and set machine into *write mode*
	UNIFY_VARIABLE A_1	in write mode, so set first argument of –/2 to be X, the first argument of procedure
	UNIFY_CONSTANT 1	in write mode, so set second argument of –/2 to be integer 1
	PUT_VARIABLE Y_1,A_4	put new variable XP in Y_1
	PUT_VARIABLE Y_2,A_3	put X–1 in Y_2
	CALL is/2	evaluate is/2 with arguments XP and X–1
	....	similar code for other goals up to the final one ...
	DEALLOCATE	remove environment before final goal
	....	similar code to prepare final goal
	EXECUTE is/2	calculate the value of Y

Summary & Conclusions

In this chapter a survey has been made of work in three areas which will be the subject of development in the remainder of this thesis. A description has been given of what is a *term rewrite system*, and a type of *graph rewriting system* called *term graph rewriting systems.* It has been shown that for certain types of TRS there exists an equivalent GRS. The practical programming languages *Dactl* and *Lean* which implement this type of graph rewriting have been outlined. Finally the *G-Machine* and *WAM* abstract machines have been described, which shows how functional and logic programming languages may be efficiently implemented on a sequential machine architecture. Since functional languages and GRS have a close similarity in their computational semantics, it is apparent that the G-Machine will make a sound basis on which to begin design of an abstract machine to support GRS. The WAM may be used as a reference when trying to support those features of logic programming we might wish to incorporate into an abstract machine for GRS.

We will use the material described in this chapter as the basis for the original work in this thesis. Chapter 2 will develop the notion of GRS to incorporate primitives to support the implementation of logic programming, and Chapter 3 will then describe the *TAM-Rewrite* language which implements this modified graph rewriting as a programming language. Chapter 4 uses the knowledge of abstract machines to develop a new machine *TAM* and instruction set *T-Code* to support TAM-Rewrites, and Chapter 5 gives compilation rules to translate TAM-Rewrites into T-Code.

2 Graph Rewriting & Logic Programming Languages

In this chapter the issues relating to the use of graph rewriting for the implementation of logic programming languages will be discussed. Firstly, we need a definition of what a *logic programming language* is, and what shall be considered its *evaluation*. We may then compare logic programming with functional programming (which has a close similarity with GRS) to learn what are the essential differences between the two approaches, and use this to motivate the approach to the task of implementing logic programming in GRS. This task will first be attempted using the definition of GRS given in Chapter 1, which will reveal deficiencies in the definition when applied for this purpose. This will serve to motivate a discussion which leads to the proposal for enhancements of GRS to support the implementation more directly.

Stated informally, we may say:

- a *logic programming language* P consists of a syntactic language L, and a set of directed equivalence rules R which express transformations or *reductions* which may be applied to terms of the language.
- the *evaluation* of P consists of taking some input term $i \in L$ with free variables v, applying the rules R in some specified manner to i until it is in some normal form, and then returning the normal form expression together with the bindings of v. If at any stage more than one rule may be applied then the normal forms resulting from applying each of the rules should be found in turn. Thus a series of normal form expressions may be returned, each with associated variable bindings.

The rules R decide the logic that is being implemented by P, and may for pragmatic reasons cause side-effects. For example, the rule for Prolog `write(x) ---> TRUE` has the side-effect that `x` appears on the terminal screen.

The definition of a logic programming language is intentionally loose, and encompasses a much wider class of languages than those based on Horn-Clause logic, for which the title is usually reserved. Any computational logic which we may represent as term rewrite rules is acceptable; in particular the database query language *PLL* [Bab91] and executable temporal logic *USF* [Gab89] are included.

In [Bel86] a comparison was made between logic programming (restricted to the Horn-Clause subset) and functional languages, which revealed two fundamental differences between the two approaches:

- *Determinism*: logic programming uses a search-based computational model to achieve the effect of the non-deterministic evaluation of logical disjunctions, whilst functional languages have clear deterministic computational semantics. For example we might have the following as a legal Prolog program:

  ```
  max(A,B,A) :- A>B.
  max(A,B,B) :- A=<B.
  ```

  but in the functional language Hope [Fie88] we are only allowed to give multiple rules for a function if the heads of the rules can not match the same term, and so the program must be a single rule:

  ```
  max(A,B) <= if A>B then A else B;
  ```

  Whilst some other functional languages will permit several rules to match the same term, the choice of which rule is used is made according to some reduction strategy (e.g. sequential ordering) at the time matching is attempted, and can not be changed later.

- *Logical Variable*: logic programs use unification (essentially bidirectional pattern matching), and allow variables to be treated like any other data, and thus be passed to and returned from functions. A corollary of this is that the local variables of a function may remain in existence outside the scope of the function. For example, in Prolog we may write:

  ```
  make_list(0,[]).
  make_list(N,[H|T]) :- NP is N−1,make_list(NP,T).
  ```

  where the query make_list(3,X) gives X=[_1,_2,_3] in which _1 *etc* represent variables created in the execution of the program. For Hope, what appears to be semantically the same rule is illegal, since free variables such as H are not permitted:

  ```
  make_list(N) <= if N=0 then [] else H::make_list(NP) where NP=N−1;
  *** syntax error
  ```

Since the relationship between functional languages and GRS is both well established and close, these comments can be seen to be equally valid to the comparison of logic programming with GRS. A GRS is non-deterministic when no evaluation strategy is being used, but that is a 'blind' non-determinism which will only follow 'correct' reduction orders (i.e. ones not ending in failure) by chance. In contrast the searching mechanism of logic programming means that the 'correct' reduction order is always discovered (except for entering infinite loops or if 'non-logical' features of the language are used). The objective of [Bel86] was to bring together logic programming and functional languages into a common formalism, and towards that aim it was suggested that functional languages be extended to include unification, and that logic programming be restricted in their use of both search-based computations and the logical variable.

Many of the parallel logic programming languages [Sha87], such as GHC and Parlog may be regarded as meeting these criteria. They belong to a family of languages called *committed-choice* logic languages, which ease the problems associated with the parallel implementation of logic programming by making the rules of the program commit after the evaluation of some guard, and not be permitted to backtrack. Rules take the form

$$p(a_1,...,a_n) \text{ :- } guard_a \mid body_a.$$
$$p(a_1,...,a_n) \text{ :- } guard_b \mid body_b.$$

All the rules for $p(a_1,...,a_n)$ have their guards $guard_a$, $guard_b$ *etc* evaluated in parallel, during which no variables from the program goal may be instantiated. The first guard to succeed causes the rule to commit, and the evaluation of all other guards to be stopped. Only then does evaluation of the rule body begin, which may instantiate variables from the goal. Thus we never need consider the backtracking of variable instantiations, since the only variables that are instantiated before a rule commits are those local to the rule, and thus can be ignored for the purposes of backtracking. After a rule commits there can be no backtracking on the choice made by the guard, hence the name 'committed-choice'.

Such languages readily lend themselves to implementation in GRS, and implementations of both GHC [Gla88b] and Parlog [Pap89] have been achieved in Dactl. These implementations provide for the logical variable and a unification algorithm, but due to the nature of the languages they implement, do not address the issues concerned with non-determinism (in the sense of always finding the 'correct' clauses which lead to a solution). By contrast, the objective of this thesis is to implement all aspects of logic programming in GRS. Whilst the inclusion of unification into GRS is a clear necessity which has already been achieved in several implementations, the provision of logical variables which work with some evaluation mechanism for disjunctions (search-based or otherwise) still needs to be investigated.

Some theoretical work has been conducted into the use of graph based proof procedures for Horn Clause logic programs (essentially a form of 'pure' Prolog) [Raa90]. This work essentially uses graphs as a storage mechanism for the data structures of a proof procedure, and uses the clauses of the program as a kind of rewrite rule to construct a graph representing the complete derivation of a query clause. Once this is completed, the graph is traversed to ensure some criteria hold which determine whether the query clause holds. Variables are treated rather like terms; to make a substitution for a particular variable requires a search be performed of the graph for all occurrences of the variable. This approach does not help in the task of using graph rewriting as the implementation vehicle, since too much of the computation is specified outside of the rewriting mechanism. Also, on a more practical point, construction of the complete derivation for most real programs would require extremely large graphs to be constructed.

From the foregoing discussion it can be seen that previous attempts at using graph rewriting to implement logic programming have either chosen to only partially implement logic programming, or only used graphs and graph rewriting as a data model. The remaining sections of this chapter will develop the definition of a GRS to allow for the implementation of all aspects of logic programming, including support for non-determinism. This will be conducted by firstly considering how we may represent the logical variable in GRS, secondly providing an algorithm to unify graphs, and thirdly discussing the issues relating to the evaluation of disjunctions.

2.1 The Logical Variable

Of fundamental importance when implementing logic programming is the ability to handle the concept of a *logical variable*, which in the context of GRS we can define as follows:

Definition 2.1 : Logical variable

> A *logical variable* is a graph which we interpret as representing any unknown graph. The normal GRS restriction that no rewrite rule may contain a variable in the body which does not appear in the head is removed. Such *free variables* cause a new logical variable to be created in the graph as part of the normal contractum building phase.

The graphical representation of formulae lends itself naturally to the representation of the logical variable, by simply allocating a special node label (here the name VAR is used) to represent a variable node, and then allocating a node with that label for each variable. When the value of the variable is discovered during computation then the variable can be *instantiated* with the value, that is all occurrences of the variable are substituted with the value. This process of *variable instantiation* or *variable binding* could be easily achieved using the normal graph rewrite operation, where all pointers to the graph representing the variable are redirected to the value it is being bound to. This definition and use of the logical variable is similar to that found in Dactl implementations of logic programming [Gla88b,Pap89], where an *overwritable* symbol Var is used for variables, and graph redirection used for variable instantiation. An overwritable symbol is one which is not found as the root of any rewrite rule, but nevertheless may be changed by a redirection operation.

Since the definition of logic programming language requires that we are able to inspect the values taken by variables at the end of the computation, any 'external' reference to a variable must also be updated by the redirection. Also the need to try several rules means that in some circumstances we need to be able to *backtrack* (i.e. reverse) the binding of variables during computation (as shall be seen in the section describing the evaluation of disjunctions), which requires that we must record all the nodes effected by redirection so that the action may be reversed.

At this point it may be thought that the graph rewrite operation is becoming a rather cumbersome procedure, but a well known implementation procedure [Pey87] of *indirection functions* will be seen to overcome this problem, which here will be called *identity functions* to correspond with their intended semantics.

2.1.1 Identity Functions and Nodes

Definition 2.2 : The identity functor and identity functions

Nominate a privileged functor $\Im/1$ as the *identity functor*, which for all graphs t of a GRS obeys the identity rule $\Im(t) \approx_{\Im} t$, which says a node labelled with $\Im/1$ is regarded as being equivalent the node pointed to by its $succ_1$ entry. In particular, Definition 1.8 of homomorphism is changed to ignore nodes labelled with $\Im/1$, so that

$\forall n \,.\, lab(n)=\Im \rightarrow h(n)=h(succ_1(n))$

For pattern matching purposes, this has the consequence that if t is a potential redex then so is $\Im(t)$, and that during pattern matching phase of rewriting (q.v. Definition 1.9) we regard any node n with $lab(n)=\Im$ as being the node $succ_1(n)$.

We may use the identity functions to produce a modified definition of graph rewriting in Definition 2.3 which uses the new functions to replace the redirection phase with an *indirection phase*. This definition also adjusts the garbage collection phase to take account of the fact that logical variables are referred to from outside the graph, and thus their instantiations must not be removed by garbage collection.

Definition 2.3 : Graph rewriting using identity functions and logical variables

The graph rewriting of Definition 1.9 is modified in steps (iii) and (iv) to be:

(iii) the *indirection phase* only updates the *lab* and *succ* function so that

$lab_2(h(r_h))=\Im$
$succ_2(h(r_h))=h(r_b)$

(iv) the *garbage collection phase*, which finds all nodes $n \in N_2$ for which

- no path exists from r_2 to n, and
- no path exists to n from any of the nodes representing the logical variables.

Remove all such nodes from g_2 to give g_3.

The effect of Definition 2.3 on the rewriting process for a rule $t \rightarrow s$ is that the root of the redex is *overwritten* with an indirection function $\Im$ pointing to s, whereas previously we would have redirected all pointers to the redex root to instead point at s. It is apparent that this gives a gain in efficiency since there is now no need to redirect any of the pointers to the redex, whilst the body s may have any address. Set against this, there will be an overhead of having to follow indirection nodes when accessing graphs. We will refer to the graph rewriting of Definition 1.9 as the *redirection method* and that of Definition 2.3 as the *indirection method*.

The two types of rewriting will produce identical results in all circumstances where the redex root would have been garbage collected in the redirection method, since it can not then matter what the indirection method uses this node for. The redex root will be garbage collected unless it is also the contractum root, i.e. unless $h(r_h) = h(r_b)$. This would be trivially true for rules where $r_h = r_b$, but otherwise will only occur when a selector function 'removes' a path of length n from $h(r_h)$ to $h(r_b)$ from a cycle of length n, thus making $h(r_h) = h(r_b)$, i.e. we have a rule of the form $f(g(\boldsymbol{x})) \rightarrow \boldsymbol{x}$ and a cyclic graph $\boldsymbol{a}$:$f(g(\boldsymbol{a}))$. In this circumstance the indirection to the contractum root is incorrect, since rewriting using the redirection method should have left the redex root unchanged, but using the indirection method changes it to become $\Im(\Im(\Im(\Im(...))))$. Example 2.1 illustrates a situation where this phenomenon occurs.

Proof Sketch 2.1 shows that two methods are equivalent except for a GRS with cyclic graphs (with a slight exception in that a different number of nodes are found in the garbage collection phase). The restriction to acyclic graphs should not prove an onerous one in the context of logic programming, since it is unclear what type of logical expression would be represented by a cyclic graph. Thus we may safely use the indirection method as the implementation method in the TAM-Rewrite language, provided we are not using cyclic graphs in a GRS with selector functions.

Proof Sketch 2.1 : Rewriting using the indirection and redirection methods is equivalent

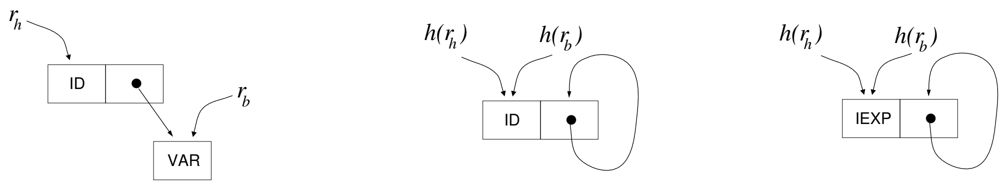
The two rewriting methods will be shown to be equivalent when rewriting acyclic graphs, in that if the result of rewriting graph g using the indirection method is graph g_i, and the result of rewriting g using the redirection method is g_r, then $g_i \approx_{\Im} g_r$. The practical difference is that the production of g_i will not involve garbage collecting the redex root. Since we have restricted the GRS so that $h(r_h) \neq h(r_b)$, we know that the two methods can differ only in the results of the updates to *succ,* the graph root, and the garbage collection phase.

For redirection: all instances of $succ(n)=h(r_h)$ become $succ(n)=h(r_b)$.
For indirection: all instances of $succ(n)=h(r_h)$ become $succ(n)=\Im(h(r_b))$.
Since $\Im(h(r_b)) \approx_{\Im} h(r_b)$ the two techniques have updated *succ* in the same manner.

For redirection: if the graph root $r=h(r_h)$ then it becomes $r=h(r_b)$.
For indirection: if the graph root $r=h(r_h)$ then it becomes $r=\Im(h(r_b))$.
Since $\Im(h(r_b)) \approx_{\Im} h(r_b)$ the two techniques have updated the graph root r in the same manner.

For redirection: the redex root $h(r_h)$ will not have any pointers to itself if $h(r_h) \neq h(r_b)$, and will thus be removed by garbage collection.
For indirection: the redex root $h(r_h)$ remains in the graph, and is not garbage collected.

Example 2.1 : Cyclic graphs can give different results when using indirections



(a) Rule $id(\boldsymbol{x}) \to \boldsymbol{x}$ (b) Redex $\boldsymbol{a}$:$id(\boldsymbol{a})$ (c) Result after indirection

Figure (a) shows the graph for the rewrite rule $id(\boldsymbol{x}) \to \boldsymbol{x}$ which matches the redex $\boldsymbol{a}$:$id(\boldsymbol{a})$ shown in Figure (b). The result of rewriting using the redirection method is also the graph shown in Figure (b), since $h(r_h)=h(r_b)$. However, the result of using the indirection method is that shown in Figure (c), which is the graph with an infinite 'equivalent' term $\Im(\Im(\Im(\Im(...))))$.

The identity function $\Im$ is represented as a graph in exactly the same manner as any other function (here the name IEXP is used). When speaking of the graphical form of the identity function we can say that it represents an *indirection* to its argument. Any rewrite operation may use it to avoid the need for redirecting pointers, but when used for logical variables it has a useful side-effect. If we use indirection during rewriting, we instantiate a variable to value x by overwriting the variable node with $\Im(x)$. Inspecting the location of the variable at any later time during the computation will enable us to read the value $\hat{W}\Im(x)$ (which is equivalent to x), and if necessary change it. Note that it is not sufficient to simply overwrite the variable with a copy of its value, since the original value may itself change at a later time. For instance, if the value is another variable, that may in turn be instantiated to a ground graph at a later time. Example 2.2 illustrates the use of the indirection method during rewriting, and how the location of variables may always be read to find their current instantiation.

Example 2.2 : Reduction using the indirection method

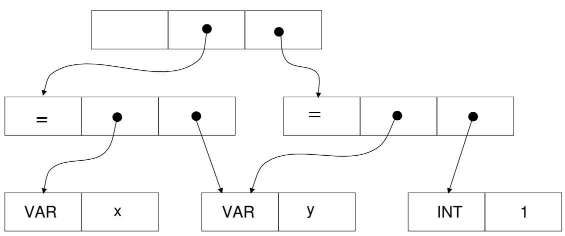
$\underline{\boldsymbol{x}=\boldsymbol{y}} \wedge \boldsymbol{y}=1$ (1)

$\rightarrow \underline{true \wedge \boldsymbol{x}=1}$ (2)

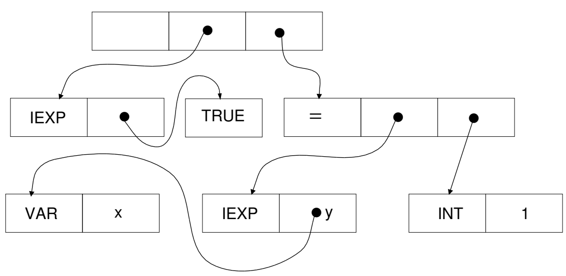
$\rightarrow \underline{\boldsymbol{x}=1}$ (3)

$\rightarrow true$ (4)

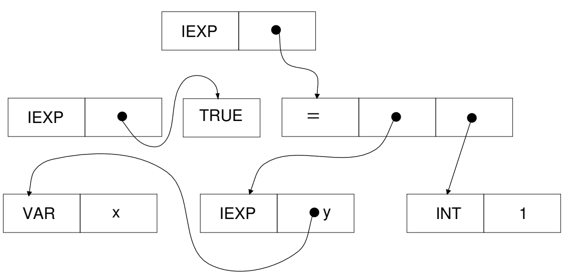
will result in the following sequence of graphs:



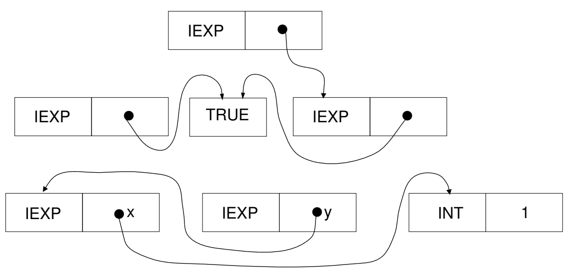
(1) Initial graph



(2) After reduction of ***x*=*y***



(3) After reduction of $\wedge$



(4) Normal form after reduction of ***y*=*1***

(1) The initial graph contains two VAR nodes for the ***x*** and ***y*** variables. By maintaining a record of the location of these variables, we may inspect the values they might have obtained when the normal form of the graph has been reached.

(2) The action of equating ***x*** and ***y*** requires two alterations to the graph. Firstly, the node representing ***y*** is overwritten with an indirection to the node representing ***x*** (the opposite might equally have been done), and secondly the node representing an equality between the two variables is overwritten with an indirection to TRUE.

(3) The reduction of the conjunction makes the equality between ***y*** and *1* the root of the graph via an identity function.

(4) The equality between a variable and a constant requires two actions. Firstly, the variable is overwritten with an indirection to the node representing *1*, and secondly the equality node is replaced by an indirection to TRUE as the root of the graph.

When reduction is complete we have TRUE as the normal form, and inspecting the location of the variable ***x*** shows it instantiated with $\Im(1)$, and ***y*** with $\Im(\Im(1))$, both of which are equivalent (under $\approx_{\Im}$) to *1*. The answer can be read as TRUE with the substitutions *1/**x*** and *1/**y***.

2.2 Unification of Graphs

The algorithm to unify a pair of term-graphs presented in this section is an adaptation of the well known *Robinson's Algorithm*, used in many Prolog interpreters. Algorithm 2.1 takes two graphs g_1 and g_2, and terminates with the answer true if there is a *most general unifier* (*m.g.u.*), in so doing making some variable substitutions, and terminates with false if no m.g.u. exists. An m.g.u. is the set of variable substitutions that makes the two graphs equivalent, and makes it possible for them to be unified with the widest set of graphs of the language. For example, we could unify g(X,1) with g(Y,Z) by making 6/X,6/Y,1/Z giving a unified form g(6,1), but the most general unification would be X/Y,1/Z giving a unified form g(X,1).

Algorithm 2.1 is presented using rewrite rules of the TAM-Rewrite language (q.v. Chapter 3). The use of rewrite rules enables some of the results reported upon in Chapter 1 to be used in Proof Sketch 2.2, which shows that Algorithm 2.1 is a correct implementation of the original term-based algorithm.

Algorithm 2.1 : Unification of term-graphs

(i) Two constants unify only if identical.

```
unify(x,y) :- ground(x),ground(y),x==y ---> true.                    (1)
unify(x,y) :- ground(x),ground(y),x=/=y ---> false.                  (2)
```

(ii) Two functions unify if they share the same functor, and the list of the arguments unify.

```
unify(f,g) :- f =.. fname::fargs, g =.. gname::gargs, fname==gname
        ---> push_stack(fargs,gargs).                                (3)

push_stack([],[]) ---> true.
push_stack(fh::ft,gh::gt) ---> and(unify(fh,gh),push_stack(ft,gt)).
push_stack(fh::ft,[]) ---> false.
push_stack([],gh::gt) ---> false.

and(true,true) ---> true.
and(false,x) ---> false.
and(x,false) ---> false.
```

(iii) No constant may unify with a function.

```
unify(f,c) :- f =.. fname::fargs,ground(c) ---> false.               (4)
unify(c,f) :- f =.. fname::fargs,ground(c) ---> false.               (5)
```

(iv) Two functions do not unify if they have different functors.

```
unify(f,g) :- f =.. fname::fargs, g =.. gname::gargs, fname=/=gname
        ---> false.                                                  (6)
```

(v) Two variables always unify, the choice of which we substitute for the other is arbitrary.

unify(v_1,v_2) :- var(v_1),var(v_2),v_1:=v_2 ---> true. (7)

(vi) A variable v does not unify with a graph g if it occurs strictly within g, i.e. if $v \subset g$. This is generally known as the *occurs check*, and due to the computational complexity of the operation it is frequently omitted from implementations of Prolog.

```
unify(v,g) :- var(v),subgraph(v,g) ---> false.                       (8)
unify(g,v) :- var(v),subgraph(v,g) ---> false.                       (9)
```

(vii) A variable unifies with any graph it is not strictly within, making the graph be the substitution for the variable.

```
unify(v,g) :- var(v),\+ subgraph(v,g),v:=g ---> true.     (10)
unify(g,v) :- var(v),\+ subgraph(v,g),v:=g ---> true.     (11)
```

Each group of rules is clearly equivalent to the textual statement of the algorithm, with the exception perhaps of (ii). Here, Robinson's algorithm states that corresponding pairs of arguments from the two functions being unified should be pushed onto a stack; Each pair is then popped from the stack in turn and unified, unification succeeding only if all these steps succeed. Here the function push_stack/2 simulates this by putting the pairs in unify/2 functions, building the 'stack' in a conjunction (which is the interpretation we shall construct with and/2 functions), and providing the obvious reduction rules for and/2.

Proof Sketch 2.2 : Algorithm 2.1 is a correct implementation of Robinson's algorithm in a GRS

It is assumed that the rules of Algorithm 2.1 are a correct specification of the TRS version of Robinson's algorithm. To demonstrate that it is correct to implement the algorithm using the equivalent tree rules in a GRS, it is sufficient to show that the rules form a weakly regular system (q.v. Section 1.2.4). It is clear that all the rules are left-linear, and are not self-ambiguous. To consider if the rules are ambiguous, a table may be drawn of the types of graph each rule may match for its two arguments arg_1 and arg_2 in the division of functions, variables and constants:

arg_1 / arg_2	function	constant	variable
function	(3) (6)	(5)	(8) (10)
constant	(4)	(1) (2)	(10)
variable	(9) (11)	(11)	(7) (10) (11)

Note that constants and variables have no subgraphs (they are single nodes) and so could not be matched by the argument which is tested for having a subgraph in rules (8) and (9). To prove that the whole system is unambiguous, each of the entries in the above table that contains several rules must be considered to ensure that rules cannot match the same graph unless they rewrite to the same body:

- (1) and (2): (1) only matches when the constants are identical, (2) only matches when they differ.

- (3) and (6): (3) only matches when the functors are the same, whilst (6) only matches when they differ.
- (8) and (10): no graph can both contain, and not contain, the same subgraph. The same argument applies for (9) and (11).
- (7), (10) and (11): all have the same body.

2.3 Logical Disjunctions

A fundamental feature of logic programming languages is that we allow the variables of the computation to have a series of values associated with them corresponding to the value the variable takes in different disjuncts, and provide some method for that series of values to be presented to the user. This conflicts with the graphical representation of variables, which inherently represents a single value. Furthermore, the multiple values associated with variables may mean alternative rewrites apply to the same graph as the variable takes different values. In this section these problems are illustrated by presenting some examples, and then some general solutions are proposed.

2.3.1 Phantom Variable Bindings

If we naively apply graph rewriting to a disjunction which gives different values to the same variable, we find that the operation of binding a variable to a value in one disjunct will result in a *phantom variable binding*, where the same value is given to the variable in all other disjuncts.

Example 2.3 : Phantom variable binding

We have the rewrite rule

eq(x,g) :- var(x),ground(g),x:=g ---> TRUE

stating that where x is a free variable and g is any ground term, then the x node may be overwritten with an indirection to the node representing the g. The following is then a 'correct' result (in terms of the semantics of GRS) of applying our rule to a disjunction:

or(<u>eq(y,1)</u>,eq(y,2)) ---> or(TRUE,eq(1,2))

The binding of the variable y in the eq(y,1) disjunct has also erroneously bound the y in the eq(y,2) disjunct.

There are at least three possible solutions to this problem:

- ***Backtracking*** where the evaluation of different disjuncts is taken in turn, and a record is made on a *trail* of all variables bound during the evaluation of a particular disjunct. Thus when evaluation of it is complete, the variables may be returned to the state in which they were previously, before evaluation of the next disjunct begins. For this to work successfully, it is important that we use the indirection rewrite method, so that the location of a variable is always fixed, and thus references on the trail regarding the variable can be made to this fixed location. A pseudo-code algorithm for evaluating a disjunction $\phi \vee \psi$ is as follows:

  (i) *record_variable_state*(***s***)
  (ii) $\phi := !\phi$
  (iii) $\phi := (\phi \wedge$ *return_variable_bindings*(***s***))
  (iv) *restore_variable_state*(***s***)
  (v) $\psi := !\psi$
  (vi) $\psi := (\psi \wedge$ *return_variable_bindings*(***s***))
  (vii) *restore_variable_state*(***s***)

  The procedure *record_variable_state* stores in ***s*** a reference suitable for use by the procedure *restore_variable_state*, which returns variable bindings to their values at the time ***s*** was recorded. The procedure *return_variable_bindings* returns a structure representing the variables bound since ***s*** was recorded.

- ***Variable Substitution*** where we use the equivalence that

$$\phi \vee \psi \equiv \phi \vee [\nu/\nu'] \, [\nu'/\nu] \, \psi \text{ (where } \nu' \notin \phi \vee \psi)$$

  whenever ν appears in both ϕ and ψ, so that there is no variable name clash between the two disjuncts. We may then apply rewrites to ϕ and $[\nu'/\nu]\psi$ without concern of phantom bindings occurring, and then use the outermost substitution to remove the new variable given to ν' in ψ. A pseudo-code algorithm for evaluating a disjunction $\phi \vee \psi$ is as follows:

  (i) $\boldsymbol{s} := id$
  (ii) WHILE $\nu \subset \phi \wedge \nu \subset \psi \wedge \text{variable}(\nu)$ DO $\psi := [\nu'/\nu]\psi$, $\boldsymbol{s} := [\nu/\nu'] \circ \boldsymbol{s}$
  (iii) $\phi := !\phi$
  (iv) $\psi := !\psi$
  (v) $\psi := \boldsymbol{s}(\psi)$

  In step (ii) we rename all variables in ψ which are also present in ϕ, and at the same time compose a function ***s*** which can reverse this renaming. Having done this, the two disjuncts in steps (iii) and (iv) can be rewritten without concern for conflicts over variable bindings, before in step (v) reversing the renaming of variables by use of the function ***s***.

- ***Set Based Evaluation*** where we store with each variable the set of values associated with it, and thus can represent the alternate values of various disjuncts directly.

2.3.2 Phantom Rewrites

The phantom variable effect discussed above is a manifestation of a more general problem concerned with sharing graphs between different disjuncts. Since each disjunct may have its own particular bindings for some variables, there is the possibility that different rewrite rules will apply to a certain redex in different disjuncts. The overwriting of the redex in one disjunct with the body of the rule, which matches only for the variable bindings of that disjunct, may be incorrect for other disjuncts containing that redex, and result in *phantom rewrites* occurring.

Example 2.4 : The phantom rewrite problem

$$(\boldsymbol{y}=1 \vee \boldsymbol{y}=2) \wedge \boldsymbol{x}=\boldsymbol{y}$$

$$\rightarrow (\boldsymbol{y}=1 \wedge \boldsymbol{x}=\boldsymbol{y}) \vee (\boldsymbol{y}=2 \wedge \boldsymbol{x}=\boldsymbol{y}) \tag{1}$$

$$(\boldsymbol{y}=1 \wedge \boldsymbol{x}=\boldsymbol{y}) \tag{2}$$

$$\rightarrow (TRUE \wedge \boldsymbol{x}=1) \tag{3}$$

$$\rightarrow \boldsymbol{x}=1 \tag{4}$$

$$\rightarrow TRUE \tag{5}$$

$$\rightarrow (\boldsymbol{y}=1 \wedge \boldsymbol{x}=1) \vee (\boldsymbol{y}=2 \wedge TRUE) \tag{6}$$

$$(\boldsymbol{y}=2 \wedge TRUE) \tag{7}$$

$$\rightarrow \boldsymbol{y}=2 \tag{8}$$

$$\rightarrow TRUE \tag{9}$$

$$\rightarrow (\boldsymbol{y}=1 \wedge \boldsymbol{x}=1) \vee (\boldsymbol{y}=2) \tag{10}$$

After (1) backtracking is used to reduce the disjunction, beginning with the left-hand disjunct. In step (4) of the rewrite we redirect the ***x*** variable to point at the number *1*, and overwrite the equality with *TRUE*. However this will also change the same equality in the right-hand disjunct, so that when step (6) is reached, we see that the equality ***x=y*** present at (1) has been erroneously changed to be *TRUE*.

The example shows that a reduction applied to an expression shared between two disjuncts, which is valid in the context of the evaluation of one disjunct, may result in an incorrect reduction in the other disjunct (as far as the logical meaning we associate with the expression that the graph represents). Loosely speaking, we can state that no graph should be overwritten inside the scope of a disjunction, if the rewrite rule used depends on the instantiation of a variable made also inside the scope of the disjunction, unless enough information is recorded so that the rewrite may be fully reversed at some later stage.

In broad terms, there are three solutions to avoiding this problem:

- ***Trailing Rewrite Operations*** : in the same way as Prolog trails the binding of logical variables, we could trail the graph nodes changed by a rewrite operation, and restore their previous values on backtracking. It is clear that such a trail would have a considerable implementation overhead since every rewrite operation would require at least one entry on the trail.

- ***Graph Copying*** where we do not permit expressions to be shared between disjuncts, and enforce a rule that when an expression x is 'pushed' inside the scope of a disjunction by rewrite rules such as $x \wedge (y \vee z) \rightarrow (x \wedge y) \vee (x \wedge z)$, then at least one instance of x must be copied. The operation of such copying is obvious and intuitive for all types of graph node except the variable node. The variable node represents the concept of an unknown graph by its position in memory, and copying it would produce a new (i.e. different) variable. This is obviously related to how we handle variables, and we will return to this point later on.

- ***Term Rewrites on Graphs*** where the result of the rewrite does not overwrite the root of the redex, but instead is returned as a graph representing the reduced form of the redex. This is used only to update the pointer to the redex which was used during the original pattern matching of the rule against the redex. In the context of graph rewriting we may refer to such rules simply as term rewrites.

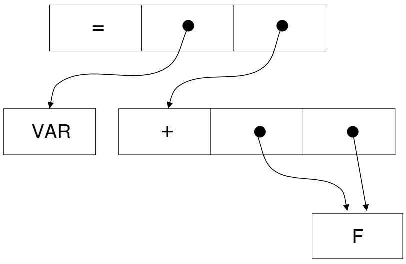
  The ---> (or for emphasis -t->) operator will be used in the TAM-Rewrite language to indicate rules which work by term rewriting, and the -g-> operator for rules which work by graph rewriting. Example 2.5 illustrates the difference in results produced by the two types of rewrite rules.

  The distinction between these two types of operation also appears in an early version of Dactl [Gla87], where a rewrite rule could specify both the overwriting and redirections for a redex. At the time [Gla87], there seemed to be no clear semantics or practical use for providing distinct types of rewriting. Also, allowing for the programmer to specify the details of how rewriting was to be conducted would preclude certain implementation optimizations, and thus the distinction between the two types of rewriting was dropped from the next version of Dactl [Gla88a].

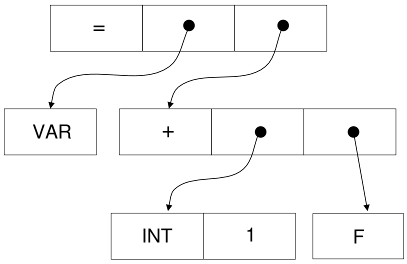
Example 2.5 : Different results of term and graph rewrites on x=F()+F()

Term rewrite rule F() -t-> 1

x=<u>F()</u>+F() -t-> x=1+F()



Graph before rewrite

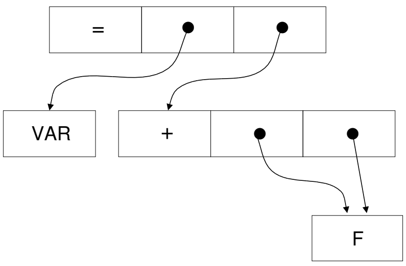


Graph after rewrite

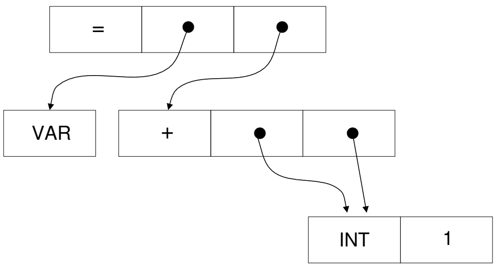
The rewrite operation only redirects the pointer ‘involved’ in the pattern matching to point to the contractum 1, leaving all other references to the redex F() unchanged.

Graph rewrite rule F() -g-> 1

x=<u>F()</u>+F() -g-> x=1+1



Graph before rewrite



Graph after rewrite

The rewrite operation causes all pointers to the redex F() to be redirected to the contractum 1.

A comparison of *graph copying* and *term rewrites on graphs* as the solution to the phantom rewrite problem can be only subjective at this stage. It can be recognized that graph copying has a considerable overhead since it copies graphs which may never be overwritten, which is frequently the case in practice. For example, consider rules such as

$FALSE \wedge \boldsymbol{x} \rightarrow FALSE$
$TRUE \vee \boldsymbol{x} \rightarrow TRUE$

The graph matched by $\boldsymbol{x}$ could represent a large expression which is being 'dropped', and thus if previously copied would have no pointers to it, and would be subject to garbage collection. In contrast, the term rewrite approach only 'copies' (i.e. does not put an identity function in) the redex root node. Thus the worst case is that it 'copies' a single node which is not shared.

Relating the Phantom Variable Binding and Phantom Rewrite Solutions

A stated before, the phantom rewrite effect is a more general instance of the phantom variable binding effect, and thus the choice between graph copying or term rewriting is related to the choice of solution for dealing with phantom variable bindings. This leads to there being two combined approaches to the execution of logic programming using graph rewriting, where either:

- *backtracking* should be used with *term rewrites*. During execution, we proceed by rewriting each disjunct in turn. Any redirections away from variable nodes made during the rewriting of one disjunct are always made using logical binds — backtrackable graph rewrites. Thus they may be reversed before proceeding with the rewriting of the next disjunct. Any other changes to part of the graph shared between disjuncts are made using term rewrites, thus preserving the structure of the graph in the context of the other disjuncts.

- *variable substitution* should be used with *graph copying*. In making a copy of the graph, any new variable formed would represent the substitution for the variable being 'copied'. Graph rewriting without any backtracking may then be used in a uniform manner for both variables and the general structure of the logical formula.

In making the choice of which overall method to use, we should also consider the computer architecture on which execution will take place. If we crudely split computer architectures into parallel and sequential types, the following comments can be made:

- *sequential machines* will execute the term rewrite solution more efficiently than copying, due the latter tending to produce the wasted copies previously mentioned.

- *parallel machines* in effect are continuously copying graphs in that they must pass graphs from one processor to another. If the communication between processors includes the substitution of variables then there may be less overhead in using the copying solution.

For the purposes of this thesis we shall only investigate further term rewrites combined with the use of backtracking, since the aim is an implementation on sequential machines.

2.3.3 Term Rewrites on Graphs

The previous subsection described the notion of a term rewrite on graphs in a practical context. In this subsection a more formal definition is given, by relating term rewrites to the definition of graph rewriting introduced by Definitions 1.9 and 2.5. Before this can be done, the concept of *context* in pattern matching must be defined.

Definition 2.4 : Pattern matching contexts

> The process of pattern matching a rule against a graph as described in Definition 1.9 involves searching the graph g for a subgraph $g_0|n$ matching the rule head. Unless n is the root, this process will arrive at the node n as $succ(n_c)_i$ of another node n_c. The *context* of the pattern matching may then be identified as the tuple $<n_c,i>$. If it is the root of the graph that matches the rule then the context is undefined.

In Example 2.5 we see that the rewrite of F() has a context of the first successor of the + node, since that was the route used during pattern matching to find the redex. As presented so far, the choice of which F() we reach in pattern matching is arbitrary, but if we use Dactl-like control markings in the rewrite rules we may specify the route used to reach a node. In the example, writing x=(!F())+F() indicates the first *succ* of + is the context for any term rewrite on F(), whilst x=F()+(!F()) indicates the second *succ* is the context.

Definition 2.5 : Term rewrites on graphs

> A term rewrite on graphs behaves in exactly the same manner as a graph rewrite, except that the redirection phase (step (iii) of Definition 1.9) will only change the instance of the *succ* function identified by the context $<n_c,i>$ of the rewrite, and does not change all instances of *succ* which point at the root of the redex.

Note that to achieve the term rewrite on graphs we can not use the indirection technique of Definition 2.3, since that always has the effect of changing all the instances of the *succ* function which point at it. The behaviour of a term rewrite on graphs is illustrated in Example 2.5.

2.3.4 Using Term and Graph Rewrites

From the preceding discussion it is apparent that the choice of using a term or graph rewrite rule for a particular reduction is critical to the correct implementation of a logic language. The purpose of this subsection is to describe guidelines for the choice of which type of rule to use in given situations. Before giving these guidelines, we need a definition of the elements of a logic programming language, which is given in Definition 2.6, which is based on that in [Hog91]. So as to avoid confusion with the terminology of term rewriting we shall refer to the 'terms' of logic languages as *logical terms*.

Definition 2.6 : Logical terms, atomic formulae and well-formed formulae

> The set of functors in Definition 1.1 are divided into *function* symbols *S, predicate* symbols *P*, and *connective* symbols *C*. For example, if *F*={f,g,likes,=,and,or,not} then we might have *S*={f,g}, *P*={likes,=} and *C*={and,or,not}.
>
> A *logical term* is a constant, variable or function made up using a function symbol and logical terms.
>
> e.g. 1 mary john X f(1,f(mary))
>
> An *atomic formula* is a function made up of a predicate symbol and logical terms.
>
> e.g. likes(john,mary) 1=2
>
> A *well-formed formula* (wff) is a function made up of a connective symbol and atomic formulae or wff.
>
> e.g. and(likes(john,mary), likes(mary,john))

The definition of a well-formed formula is made more general than usual to emphasize that the techniques described here are applicable to any type of predicate logic, not just classical logic. The discussion will be limited to consider only first-order languages, which have the restriction that a variable may only represent a logical term, and not any well-formed formulae of the language [Ric89]. The guidelines for performing rewrites on first-order languages are given in Theorem 2.3.

Theorem 2.3 : Use of term and graph rewrite rules to evaluate logical expressions

> A graph rewrite rule may only be used in the evaluation of a first-order logic language when
>
> (i) the variables in the head-pattern of the rule only match logical terms, and

(ii) the class of graph matched by the head-pattern can only be produced by graph rewriting operations.

Note that we must consider any condition expression that uses variables from the head pattern as being part of the pattern. For example, f(1) and f(x) :- x==1 both act to check that the argument is an integer, so the condition expression x==1 must be regarded as part of the rule's pattern.

The first clause of Theorem 2.3 dictates that we may use functors that are used to form a wff or predicates in a graph rewrite head pattern, but not the constants or any functors used to form terms. Thus only term rewrites may use constants in their head patterns, and if in the language being implemented we can assign the functor *f/k* to a variable, then *f/k* may not appear in a graph rewrite pattern. The second clause of Theorem 2.3 means that if a structure matched by a pattern can be formed by a term rewrite, then it is not safe to use a graph rewrite. The reason for this restriction is that a term rewrite could have been made to avoid breaking the theorem, and thus using the graph so produced to match against the pattern of a graph rewrite rule would indirectly break the theorem. An example will make this point clear. We have the rules:

```
not(not(x)) -g-> x          (1)
g(1,x) -t-> not(f(x))       (2)
```

Starting with the graph not(g(1,x)) we may use rule (2) to reduce it to not(not(f(x))) and then rule (1) to reduce that to f(x). However the second step uses a pattern structure that is produced using rule (1), which contains a term in its head pattern, and thus could vary according to the value taken by a variable. Rule (1) alone would not break the theorem, since the pattern not(x) could not be produced as the result of a term rewrite if not term rewrite has not/1 as the body root.

Note that a corollary of Theorem 2.3 is that any head pattern containing just variables as the pattern argument may be used in a graph rewrite rule. This intuitively corresponds to our understanding that a functor *f/a* is a unique entity, and that if we state an unqualified rule *f/a* → *body* about it then we can replace all instances of *f/a* with the *body*. This of course assumes no side-effects occur during rewriting, which is not enforced by the TAM-Rewrite language presented in the next chapter. It is thus the responsibility of the programmer who uses side-effects in rules to fully understand the impact that this has on the correctness of any implementation.

Example 2.6 : Valid graph rewrites for Prolog resolution

Formulae Allowed in Patterns	*Formulae Not Allowed in Patterns*
predicates likes(...) loves(...) = >	*numbers* 1 34 9.7
connectives , ; \+ ->	*'atoms'* dog cat
variables X Y Z	*functions* p(...)
	succeed fail

The table above shows that none of the Prolog logical terms (namely numbers, functors, variables and what are in Prolog terminology 'atoms') are permitted in the head of a graph rewrite rule. We include succeed and fail as terms since we need some representation of truth and falsity if we are to use rewrites in the resolution of a Prolog query.

Note that a Prolog *function* is any predicate which appears as an argument to another predicate. From the table is can be deduced that the following rewrites are invalid according to Theorem 2.3:

```
and(succeed,x) -g-> x
eq(x,1) :- var(x),x:=1 -g-> succeed          % x redirected to 1
eq(x,y) :- var(x),x:=y -g-> succeed          % x redirected to y
less(3,2) -g-> fail
```

The third case violates the theorem by making the rule variable x match only free variables of the redex (which represent variables in the formula being reduced). The following rules are valid since the rule variables may match any graph:

```
and(x,y) -g-> not(or(not(x),not(y)))
odd(x) -g-> or(x=1,or(and(xp is x–2,odd(xp))))
```

Example 2.7 : The phantom rewrite problem solved using term rewrites

$(\boldsymbol{y}=1 \lor \boldsymbol{y}=2) \land \boldsymbol{x}=\boldsymbol{y}$

-g-> $(\boldsymbol{y}=1 \land \boldsymbol{x}=\boldsymbol{y}) \lor (\boldsymbol{y}=2 \land \boldsymbol{x}=\boldsymbol{y})$ (1)

$(\boldsymbol{y}=1 \land \boldsymbol{x}=\boldsymbol{y})$ (2)

-t-> $(TRUE \land \boldsymbol{x}=1)$ (3)

-t-> $\boldsymbol{x}=1$ (4)

-t-> $TRUE$ (5)

-t-> $(\boldsymbol{y}=1 \land \boldsymbol{x}=1) \lor (\boldsymbol{y}=2 \land \boldsymbol{x}=\boldsymbol{y})$ (6)

$(\boldsymbol{y}=2 \land \boldsymbol{x}=\boldsymbol{y})$ (7)

-t-> $(TRUE \land \boldsymbol{x}=2)$ (8)

-t-> $\boldsymbol{x}=2$ (9)

-t-> *TRUE* (10)

-t-> $(\boldsymbol{y}=1 \wedge \boldsymbol{x}=1) \vee (\boldsymbol{y}=2 \wedge \boldsymbol{x}=2)$ (11)

The steps of Example 2.4 are repeated here, but using term or graph rewrites in accordance with Theorem 2.3, which dictates that only step (1) uses a graph rewrite. Notice that the use of a term rewrite in step (5) leads to the $\boldsymbol{x}=\boldsymbol{y}$ structure remaining in the second disjunct at step (6), and thus the correct reduction sequence is followed in steps (7-11).

Proof Sketch 2.4 : Theorem 2.3 is complete and correct

We have graphs x, d_1 and d_2 for which $x \subseteq d_1$ and x $\subseteq d_2$ (where the $\subseteq$ means subgraph or equal graph) and we are interpreting d_1 and d_2 as logical disjuncts.

Correctness

To prove correctness, consider if it was not the case that we could use graph rewrite rules according to the theorem to apply the same reduction sequence to x for both d_1 and d_2. In this case some subgraph of x must have a different normal form in d_1 and d_2; that is (using $\Rightarrow^*$ to mean the application of rewrite rules until normal form is reached)

$x \Rightarrow_1^* x_1^*$ during evaluation of d_1
$x \Rightarrow_2^* x_2^*$ during evaluation of d_2
$x_1^* \neq x_2^*$

The different normal forms indicate that at some place in the two rewrite sequences a different rule must have applied in one disjunct compared with the other. This would entail that a pattern of one rule failed to match where another succeeded, due to one disjunct having a different structure to the other. Only variables can change their values in such a manner, by the use of backtracking between the evaluation of the two disjuncts. In first-order languages a variable may only take a logical term as its value, but the theorem specifically precludes a graph rewrite rule to depend on a logical term in its pattern. This leads to a contradiction, since we must have a rule that uses a logical term in its pattern if the original premise is to be made to hold.

Completeness

To prove completeness, consider the case if we could use graph rewrite rules which the theorem prohibits. Then at some stage of the reduction sequence $\Rightarrow_1^*$

we use a term rewrite depending on the value of a logical term, which may be different in the evaluation of $\Rightarrow_2^*$ and thus mean that a different reduction sequence should be followed. However, the use of a graph rewrite would change (irreversibly) the structure of the original graph x, and thus possibly invalidate the second reduction sequence.

Summary & Conclusions

This chapter has described the possible problems that arise in the using conventional GRS in an intuitive manner for the evaluation of graphs given logical interpretations. In particular two phenomena have been observed:

- The *phantom variable binding* where a variable bound in the evaluation of one disjunct remains bound in the second disjunct.
- The *phantom rewrite* where a rewrite rule valid for one disjunct, but depending on the value held by a variable, is invalid in another disjunct.

The standard solution of backtracking variables has been shown to be equally valid for GRS in order to solve the phantom variable phenomenon, and the use of term rewriting on graphs resolves the phantom rewrite problem. It has also been shown that to a limited extent one may use ‘pure’ graph rewrites in accordance with some set of guidelines. Whilst it would be perfectly possible to achieve the solutions offered by encoding algorithms in conventional GRS, it may be realized that this amounts to writing an interpreter in GRS for an enhanced GRS with term rewrites and backtracking on variables. To avoid this inefficiency, the key points of term rewriting on graphs and backtracking of variables are used to form the basis for the *TAM-Rewrite* language presented in the next chapter.

3 TAM-Rewrite Rules

The previous chapter introduced some modifications to the standard definition of GRS which allowed for the implementation of logic programming languages to be more readily achieved. We now use those modifications as a motivation for introducing a new compiler target language *TAM-Rewrite* (Term and graph Abstract Machine Rewrites) specifically tailored to support implementation of logic programming languages. This rewrite language has an associated *TAM* abstract machine and instruction set designed to be efficiently implemented on Von Neumann architecture machines. This abstract machine will be introduced in the next chapter, and the compilation from rewrites to abstract machine instructions will be detailed in Chapter 5.

The key features which differentiate the TAM-Rewrite language from other compiler target languages such as Dactl and Lean (q.v. Section 1.3) are itemized below.

- ***Unification*** is provided as a built-in operation of the language (q.v. Algorithm 2.1). Unification provides a version of the 'normal' pattern-match operation which works in 'both' directions, and allows for a variable to be treated as a value of another variable.

- ***Term and Graph Rewrites*** are supported as distinct operations in accordance with Definitions 2.3 and 2.5.

- ***Logical Variables*** are introduced by allowing variables not in the head of a rewrite rule to be used in the body (q.v. Definition 2.1). There is a *trail* provided to store the contents of variables before they are bound to some value. During application of a rewrite rule any such variables are created as part of the build phase in graph rewriting. The following operations may be performed on logical variables:

  - *Logical Bind* to set a variable to some value, recording the previous value on the trail so that it may be restored at a later time.

  - *Find Logical State* where we can obtain a reference to the values of all variables at a certain point in the rewrite process (the *logical state*), and use it at a later time to restore the values of any variables which have been the subject of a *logical bind* operation since the recording of the logical state.

- *Restore Logical State* restores the values of any variables which have been the subject of a *logical bind* operation since the recording of the logical state.

- *Return Logical Variables* which provides a mechanism to return the current instantiation of named variables in such a manner that the values returned can not be changed by a future *restore logical state* operation.

These operations, plus the usual built-in functions (such as arithmetic and list operations) are added to the conventional rewrite rule by providing a *condition clause* to be satisfied as part of the head pattern matching process. The *TAM-Rewrite rules* take one of the forms given below:

$head(a_1,a_2,...)$:- *condition* -t-> *body* .
$head(a_1,a_2,...)$:- *condition* -g-> *body* .
$head(a_1,a_2,...)$:- *condition* .

The third form rule is used for programming additional predicates for the use in the condition clause of the first two types of rule, and such rules are referred to as *condition rules*. The other two forms of rules are referred to as *rewrite rules*. As is conventional in GRS, the *head* is a pattern that must be matched against a graph, the subgraph matched being the *redex*. All rules must be left-linear. Variables in the head behave as formal parameters, the actual values being the part of the redex that the variable is found to match during the pattern matching.

The *condition* is a sequence of predicates which must hold for the variable bindings made in matching the head. It can be seen to serve as a kind of guard, but unlike a guard the condition may change parts of the graph in either a reversible or irreversible manner. Some predicates have purely logical semantics, such as unification =, and others only cause side-effects and always succeed, such as indirection :=. If the condition should fail, then any variables instantiated by purely logical operations will be restored to their previous values. Variables which do not appear in the head are termed *logical variables,* and will cause a new variable graph node to be created. Since these are members of the graph, they will persist outside the scope of the rule.

The *body* is the graph pattern used to construct the contractum based on the variable substitutions found in matching the head against the redex. As with the condition part of the rule, the body graph may contain variables that do not appear in the head, which create new logical variables. A term rewrite rule will be specified by using the -t-> operator (or the synonym --->), and cause only the context of the pattern matching to be redirected to the root of the contractum. A graph rewrite using -g-> will cause the effect of all pointers to the redex root being redirected to the contractum root.

It should be mentioned early on that the TAM performs rewriting by the use of the indirection method, which is semantically equivalent to using the redirection method only for acyclic graphs (q.v. Proof Sketch 2.1). Thus any use of cyclic graphs, whilst not precluded, needs very careful study, and is not pursued in this thesis. The use of the indirection method has important consequences for our treatment of the logical variable, since it means that to read a variable's current instantiation we need only ever to look at the node in the graph allocated for that variable when it was first created.

In general, the TAM-Rewrite language ignores indirection nodes, in that a reference to node x which contains an indirection to node y yields a pointer to node y. In most situations this is the required behaviour, but when we want to manipulate a logical variable (such as reset it to be free) we must not follow the indirection. This is achieved by prefixing the reference with the *actual* operator $.

Before giving a detailed description of the different components of a TAM-Rewrite rule, a few examples will illustrate the general structure and usage of TAM-Rewrite rules.

Example 3.1 : Boolean reduction

```
and(true,x) ---> !x.          (1)
and(false,x) ---> false.      (2)
and(x,true) ---> !x.          (3)
and(x,false) ---> false.      (4)
```

Given the query !and(!and(true,false),true) where ! is the activation marking which indicates which nodes should be considered for rewriting, the reduction proceeds as follows (since activations are always considered using applicative ordering):

!and(<u>!and(true,false)</u>,true)
---> !and(!false,true) (by 1)
---> <u>!and(false,true)</u> (no rule for false)
---> false (by 2)

Removing the innermost activation from the query results in a different reduction sequence, but the same normal form:

```
!and(and(true,false),true)
---> !and(true,false)                                   (by 3)
---> false                                              (by 1)
```

Example 3.2 : Fibonacci numbers

```
fib(1) ---> 1.
fib(2) ---> 1.
fib(n) :- n>2 ---> !add(!fib(!sub(n,1)),!fib(!sub(n,2))).
add(x,y) :- z is x+y ---> z.
sub(x,y) :- z is x–y ---> z.
```

This should be a self-explanatory implementation of the well-known recursive algorithm for the computation of Fibonacci numbers. An alternative to the final fib rule is to use a *meta-call*, where as part of the rule condition we have an expression containing an activation, and use the arithmetic predicate is/2. The third rule then becomes:

```
fib(n) :- n>2, np is n–1, npp is n–2, f is (!fib(np))+(!fib(npp)) ---> f.
```

Example 3.3 : Timing 100 rewrite activations

```
iterate(X).
iterate(X) :-
        X>1,
        XP is X–1,
        iterate(XP).

time100_rewrite(x) :-
        time(s),
        ( iterate(100),r:=!x,fail() ; succeed() ),
        time(e),
        t is e–s,
        write(t),nl()
        ---> r.
```

This example illustrates both the more procedural aspects of the TAM-Rewrite language, and the use of condition rules. The condition rule iterate/1 will hold for a number of times equal to its integer argument. Thus the sequence iterate(100),r:=!x,fail() causes the rewriting of x indicated by !x to be carried out 100 times. The alternative of succeed() is provided so that the rule holds after the 100th attempt, printing the timing using write(t) before returning the result of the rewriting of x as the body of the rule.

Note that there is an assumption that the rewriting of x does not change the graph, so that we can repeat the same operation any number of times we like. This would only be the case if no indirections (:=) or graph rewrites (-g->) are used.

Before detailing in Section 3.2 the precise syntax and semantics of what the head, condition and body of a rule may be, the next section will describe the general rewrite algorithm that is used to decide the sequence in which rewrite rules will be attempted. The final two sections of this chapter describe how the TAM is used to implement two logic programming languages; Prolog and the PLL.

3.1 The Rewrite Algorithm

The search of a graph for redexes is a selective process, where candidate nodes for matching against rewrite rules are marked, or *activated*, by the ! function, and *deactivated* (the default) by the # function. Rewrite rules are regarded as an ordered set R_i ($1 \leq i \leq n$) of n rules, and the activation nodes of a graph G are regarded as a set $G_!$, ordered by taking always the left-most inner-most activation of G as being the first element.

For selecting rewrites from R_i to test against $G_!$ the Markov algorithm [Tre76] is used. This involves taking the first element $g \in G_!$, and searching sequentially R_i for a head/condition pair which are satisfied. Once such a pair is found, we build the contractum from the body of the rule, and perform redirection or indirection as dictated by the use of a term or graph rewrite operator in the rule. This may introduce new activation nodes into $G_!$, and the search of R_i recommences with the first g in the new $G_!$.

When a complete pass of R_i has been made without finding a match, the activation is stripped from g, and it is removed from $G_!$. The search of R_i recommences for the new first element of $G_!$, and reduction is complete when $G_!$ is empty.

The following is a pseudo-code program to implement the above algorithm; the predicate $\boldsymbol{a}$:== $\boldsymbol{b}$ should be read as holding when $\boldsymbol{a}$ pattern matches $\boldsymbol{b}$, and each rule of R_i is of the form $\boldsymbol{h_i}$:- $\boldsymbol{c_i} \rightarrow \boldsymbol{b_i}$. Overall, the algorithm may be summarized as evaluating the activation operators using applicative ordering, and searching for rules sequentially.

(i) $\boldsymbol{i} = 1, \boldsymbol{g_1} = first(G_!)$
(ii) if $\boldsymbol{h_i}$:== $\boldsymbol{g_1}$ and $\boldsymbol{c_i}$ then $\boldsymbol{g_1} := \boldsymbol{b}_i$; goto (i)
(iii) $\boldsymbol{i} := \boldsymbol{i}+1$; if $\boldsymbol{i} \leq \boldsymbol{n}$ goto (ii)
(iv) $G_! := G_! - \boldsymbol{g_1}$
(v) if $G_! \neq \emptyset$ goto (i)

The operation $\boldsymbol{g_1} := \boldsymbol{b}_i$ in step (ii) of the algorithm is only correct for graph rewrite rules; term rewrites should have only the context of the activation changed.

3.2 TAM Rewrite Rules

This section describes the syntax and semantics of the TAM-Rewrite rules. It is intended to give sufficient detail of the language for the descriptions of the implementations of Prolog and the PLL in the following sections to be understood, but will omit much of the detail associated with both the semantics of various condition clause predicates and the precise syntax of the language. A full definition of these aspects of the language may be found in Appendix A.

3.2.1 TAM Syntax

Definition 3.1 gives an abstract syntax for TAM-Rewrite rules using a modified BNF notation, where expressions in { } are optional.

Definition 3.1 : TAM language abstract syntax

<rule> ::= <head> :- <condition> -t-> <body>. |
<head> :- <condition> -g-> <body>. |
<head> :- <condition>.
<head> ::= <function>
<condition> ::= <condition> , <condition> | <condition> ; <condition> |
\+ <condition> | <graph> <c_binary_op> <graph> |
<graph> is <arith> |
<graph> takes <arith> | <function>
<body> ::= <graph>

<graph> ::= <terminal> | <function> | <list> | <condition> |
<graph> <binary_op> <graph> | <unary_op> <graph>
<function> ::= <identifier>({ <graph> })

<terminal> ::= <truth_value> | <integer> | <string> |

<identifier> | <pattern_class>
<pattern_class> ::= *integer | *number | *list | *var | *nonvar | *atom

<binary_op> and <unary_op> are the binary and unary operators of the TAM-Rewrite language, excluding any binary operators given special semantics in the condition, which are defined by <c_binary_op>. The definitions of these classes may be found in the complete syntax presented in Appendix A.

A full syntax is given in Appendix A. Particular points to note here are that:

- the syntax for a condition and a rule is simply a restriction of the more general syntax for a graph, and thus we may manipulate TAM-Rewrite rule definitions as graphs.
- the parameters of a function are simply one graph; it is a matter of interpretation that we consider a structure (a,b,c...) to be a list of parameters.
- all rules must be left-linear.
- *variables* are any sequence of alphabetical characters and the underscore character. TAM-Rewrite identifiers are not case sensitive, for example x and X are the same variable.
  e.g. x y X Y example_variable
- *strings* are any sequence of characters enclosed in a pair of double quotation marks.
  e.g. "simple example of a string" "f(x,y)"
- *functors* are any valid variable name followed immediately by an opening bracket.
  e.g. f(x,y) example_function(example_variable)
- *global variables* are variable names with @ prefixed. They have scope across all the rules, and it is the programmers responsibility to ensure the first use of a global variable is to assign an initial value to it.
  e.g. @x @y @example_global_variable

3.2.2 Graphical Representation of TAM Expressions

Table 3.1 lists the construction rules for building TAM expression graphs from textual expressions. For each type of BNF expression we name an instance *e*, and give the result of applying the graph creation operator , where *e* yields the pointer to the graphical representation of *e*.

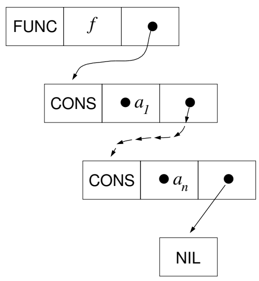
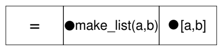
BNF Expression	*Instance*	*Graph* (= *Instance*)
<integer>	i	$\boxed{\text{INT}}\boxed{i}$
<string>	s	$\boxed{\text{STR}}\boxed{s}$
<truth_value>	t	$\boxed{t}$
<pattern_class>	c	$\boxed{c}$
<list>	[]	$\boxed{\text{NIL}}$
<list>	$[x\|y]$	$\boxed{\text{CONS}}\boxed{\bullet x}\boxed{\bullet y}$
<graph> <binary_op> <graph>	$x\ op\ y$	$\boxed{op}\boxed{\bullet x}\boxed{\bullet y}$
<unary_op> <graph>	$op\ x$	$\boxed{op}\boxed{\bullet x}$
<identifier>	v	$\boxed{\text{VAR}}\boxed{v}$
<identifier>(<graph>)	$f(a_1,\ldots,a_n)$	

Table 3.1 : Graphical representation of TAM-Rewrite expressions

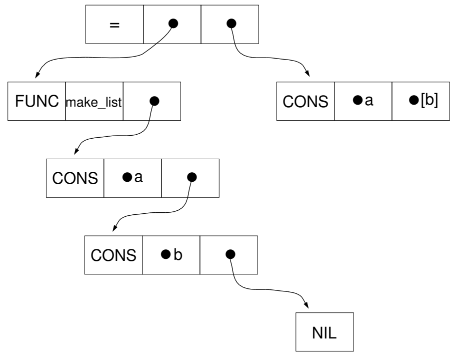
Note that the above rules do not represent the fact that all occurrences of a particular variable (which syntactically is any identifier not followed by parenthesis) should be represented by the same node in the graph. Using this restriction, any binding of a variable can be made simply by overwriting the variable node with an indirection to the value. For clarity of presentation, we sometimes add the variable name to the VAR node, although this serves no practical purposes since the address of a node serves as its unique identifier. Thus we omit variable names from the implementation described in Chapter 4. Example 3.4 shows the steps involved in the creation of a graph.

Example 3.4 : Graphical representation of make_list(a,b)=[a,b]

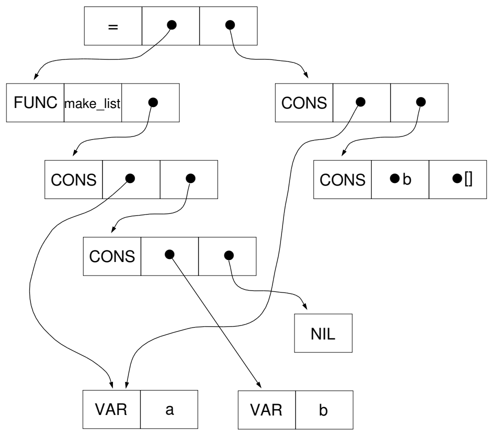
The following diagrams show the sequence of graphs generated when we apply graph creation to the textual expression make_list(a,b)=[a,b], i.e. (make_list(a,b)=[a,b]).



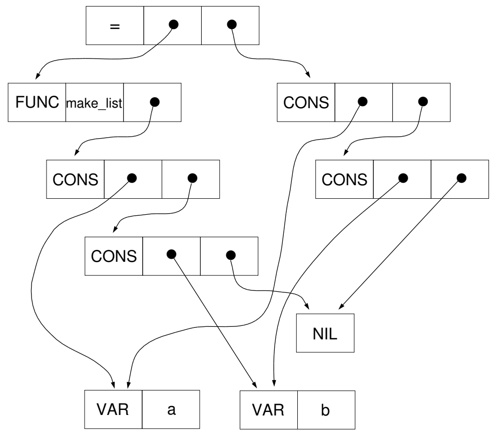
(a) (make_list(a,b)) = (a::b::[])



(b) make_list(a, b) = (a):: (b::[])



(c) make_list(a,b) = a:: (b):: ([])



(d) graphical form complete

3.2.3 Head of Rewrite Rules

Rule heads may contain any expression of the language whose root is a functor (i.e. a function application, which is represented by a FUNC node) and which contains only single instances of variables (i.e. is linear). Variables appearing in the head are local to the rewrite rule, and represent the local name for the subgraph they matched during pattern matching of the head. The only valid TAM-Rewrite expressions given special semantics in the head of a rule are the *label operator* and *pattern classes*.

The Label Operator

<labelled_expression> ::= <graph> : <graph>

The *label operator* permits two patterns to be provided for matching against a subgraph of the redex. It thus acts as a kind of *pattern conjunction* operator, where pattern matching only succeeds if both patterns match the redex. This operator will usually be used to match both a variable and a pattern against the redex subgraph. For example, the pattern x : f(a,b) makes variable x match any f/2 functor, with the variables a and b matching the parameters. Note that the restriction on the head root being a functor means we cannot label the head root with a variable, as one can in Dactl. This restriction is a consequence of the optimizations used in implementation (q.v. Chapter 4).

Pattern Classes

<pattern_class> ::= *integer | *list | *var | *nonvar | *atom | *atomic

The *pattern classes* allow for matching to be made according to node type (as opposed to a specific instance of a node or variable) for some useful classes of nodes. The graph matched by each of the pattern classes is given in Table 3.2.

Pattern	*Graph Matched*
*integer	any INT node
*list	any with a root CONS node or a NIL node
*var	any VAR node
*nonvar	any not matched by *var
*atom	any STR node
*atomic	any STR or INT node

Table 3.2 : TAM-Rewrite pattern classes

Example 3.5 : Arithmetic with one free variable

```
eq(x,add(y:*integer,z:*integer)) :- x is y+z ---> true
eq(x:*integer,add(y:*integer,z)) :- z is x–y ---> true
eq(x:*integer,add(y,z:*integer)) :- y is x–z ---> true
eq(x:*integer,add(y:*integer,z:*integer)) ---> false
```

The pattern classes here serve to choose the correct use of the condition arithmetic evaluation predicate is/2. Note that for these rule definitions the pattern classes are superfluous, since the is/2 predicate fails if its operands do not evaluate to integers or variables before the predicate is called, and thus in effect has the *integer check built-in. The final rule acts as a default case, which is considered only after the other three rules would have failed to satisfy their conditions when rewriting with three integers which do not add-up.

3.2.4 Conditional Expressions

A conditional clause is a conjunction of predicates which must hold before we say that the rewrite rule holds for a redex, and hence that rewriting of the graph should commence. The condition can be regarded as an extension of the pattern matching process, but with certain features provided which allow for side-effects to occur, such as indirections. It should be noted that there is an intended similarity of the condition clause syntax and semantics with those of Prolog. The chief differences are that there is no unification in the head, and that once a condition has succeeded the rewrite rule commits, in that no other rewrite rule for the function concerned will be attempted.

The instantiation of the local variables appearing in the head of the rule is found during pattern matching. Any variables appearing in the condition or body, but not the head of a rule, will cause new *logical variables* to be created, in the form of new VAR nodes added to the graph. Since they are part of the graph, they will continue to exist outside the scope of the rule, and may only be reclaimed by garbage collection. By contrast, the head variables can be reclaimed once the rewrite is completed, and thus do not use any permanent space. Thus it is wise to avoid the use of logical variables as much as possible.

As with Prolog, the predicates of the clause are evaluated in a strict left-to-right ordering. We may divide the predicates provided into two groups; *logical predicates* and *non-logical predicates*. The former will record any bindings made of variables so that they may be reversed at some later stage, whilst the later do not make such records. Frequently two versions will be provided of what are in effect the same operation, to allow for these different uses.

Graph Indirection and Activation

The TAM-Rewrite language provides a *indirection* operator, similar in semantics to the *redirection* operator found in Dactl [Gla88]. However, the availability of a trail to allow for the restoration of variable values means that we may in fact permit two kinds of indirection:

a := b	All pointers to the root of a are in effect made to point to the root of b. In practice this is achieved by overwriting the root of a with an indirection to the root of b.
a ::= b	A logical version of a := b, where in addition to the indirection being performed, information is recorded so that the operation may be reversed at a later stage. In practice this is achieved by recording on a *trail* the contents of the root of a before overwriting the root of a with an indirection to the root of b. The operations to find and restore the state of variables will refer to this trail.

An operation which records the previous value of a node on the trail will be referred to as a *logical bind* operation. The use of the logical bind operation on nodes representing logical variables has been studied in Chapter 2 as part of the solution to the phantom variable binding problem.

It should be noted that the TAM-Rewrite language allows the logical bind operation to be applied to any type of graph node. The usefulness of this has yet to be studied in detail, but combined with the backtracking facilities in the language which fully restore nodes to their old values, it allows any graph to be treated as a constraint which can be restricted by rewriting, and then latter restored to its original state. For example, at one time we may have [1,10] representing a constraint that some value lies in the range one to ten. When rewriting a disjunct containing [1,10]<5 we would change the constraint to be [1,4], but upon backtracking would want the previous constraint [1,10] to be restored. This is in essence the only primitive operation required by constraint logic programming languages [Hen89] above those required by logic programming, and thus its provision allows for the implementation of constraint logic programming to be achieved in TAM-Rewrites.

Any graph appearing in a TAM-Rewrite condition clause may be made subject to rewriting before it is used by the condition, providing a kind of 'meta-rewrite' facility. This is achieved by prefixing the graph with activation operator !. Thus the condition clause a:=!b will cause rewriting of b to be conducted until it reaches normal form before the usual redirection operation is made. The semantics of this use of graph rewriting has not been studied in detail, and it is only provided as an implementation convenience which in the context of a sequential implementation is useful in avoiding some intermediate stages in rewriting. For example, if we have the rule

```
g(a,b) :- a:=!b ---> f(b)
```

we can always replace it with two rules

```
g(a,b) ---> !g2(a,!b)
g2(a,b) :- a:=b ---> f(b)
```

and thus preserve the two step redirection phase/activation phase found in Dactl. Note that any non-termination of the !b will in both cases leave the first rule in some senses suspended. When the !b appears in the guard, then the guard evaluation will never finish evaluating, and when !b is in the body then the body construction will never finish.

Logical Variables

A group of predicates provides control over the state of any variable which has been changed using a logical bind operation. It is the status of the trail which we use to indicate the current state or *level* of variable bindings, and thus these predicates record information regarding the trail. We may record the current level using find_level/1, and restore it using return_to_level/1.

find_level(L) Records in L a value which represents the current state of the logical variables in the graph.

return_to_level(L) Return all variables to their values which held when L was recorded using find_level/1. Only variables made using logical bind operations will be affected by this operation.

When we have reached a normal form in rewriting a logical expression, we will need to return to the user the current instantiations of the free variables supplied in the original query. This may sometimes be performed by printing out the values, or by returning a graph containing the instantiations. The latter is performed by the return_bindings/4 predicate, which builds a graph with the values of specified variables recorded so that they will not be changed by the execution of return_to_level/1.

return_bindings(V,F1,F2,R)

Return in a structure R the bindings of the variables in list V, where each variable is paired with its binding by the arity two functor F2, and the pairings are built together using the arity two functor F1. Note the use of the $ operator in the example, so that we give the predicate the actual node in the graph representing the variable, and not what it may happen to be bound (if the node is labelled with IEXP).

```
% X=1 and Y=2 before call
return_bindings([$X,$Y],or(_,_),equal(_,_),R)
---> R=or(equal(X,1),equal(Y,2))
```

Connectives and Condition Rules

As well as the conjunction connective, a condition clause may have both disjunctions a ; b with the semantics that the condition holds for either a or b, and negation by failure \+ a with semantics that the condition clause does not succeed. It should be noted that a rule of the form f(...) :- a ; b ---> g(...), having succeeded for a, will not succeed again for b; i.e. b is an alternative to try only if a fails. This is due to the rule committing as soon as its condition clause succeeds, The condition clause may contain user defined predicates, which are defined using *condition rules* of the form *head* :- *condition*. For most purposes these condition rules may be regarded as semantically equivalent to Prolog clauses, except for the lack of unification in the head. *Example 3.6* shows how the omnipresent *Fibonacci* program can be rewritten using condition rules.

Example 3.6 : Fibonacci numbers generated using condition rules

```
fib(1,x) :- cut(),x=1 .
fib(2,x) :- cut(),x=1 .
fib(x,y) :-
        x>2,
        np is x-1,
        npp is x-2,
        fib(np,xp),
        fib(npp,xpp),
        y is xp+xpp.
```

Notice the use of the cut/0 predicate in Example 3.6, which like the Prolog namesake serves to remove all following rules from the pattern matching search process.

3.2.5 Body of Rewrite Rules

The body of the rewrite rule can be any valid TAM expression. The contractum of a rewrite is built using the body as a pattern, substituting any variables from the head with the subgraph of the redex matched. Any variable not appearing in the head will result in a new logical variable being created.

As dictated by the rewrite algorithm presented in Section 3.1, the subgraphs marked with the *activation operator* ! are considered for rewriting using applicative ordering, and are rewritten before any indirections from the redex to the contractum root are made. The consequences of using this ordering are illustrated in Example 3.7.

Example 3.7 : Order of reduction

> The expression !a(!b(!c,!d),!e) will result in the functors being considered for rewriting in the order c,d,b,e,a. In terms of the rewrite algorithm, we can say $G_!$={ c,d,b,e,a }.

Since side-effects are permitted in TAM-Rewrites, it is important to understand exactly how the body is constructed. The construction of the contractum proceeds from the leftmost innermost node of the body. Any activation marking found causes rewriting to be attempted immediately. Thus in Example 3.7 the graph for c will be constructed and rewritten before d,b,e and a are constructed.

This method of graph construction, whilst efficient to implement, precludes the use of labels that Dactl allows in the body of rules. For instance, we can not have a body f(x:1,x), where the label x causes the graph constructed to share the two instances of 1. However, by creating such shared graphs as part of the condition we may achieve an identical result. For the example we would need to write x:=1 ---> f(x,x).

3.3 An Implementation of Prolog

In this section an implementation of Prolog is presented as our first fully worked-out case study of how the TAM-Rewrites are used to implement a logic programming language. The subset of the Prolog language in [Ste86] implemented will include *negation by failure, cut* and a *write* predicate, and thus includes the principal 'non-logical' features of the language which might be expected to cause problems in any implementation. The implementation will be described by first detailing how we achieve the basic SLD resolution mechanism used by Prolog, then describing how to perform backtracking in the resolution, and finally how negation by failure and cut are implemented. A complete listing of the TAM-Rewrite program for the subset of Prolog may be found in Appendix B.

It should be noted that much more efficient implementations of Prolog (based on the WAM) are already available, so this implementation should be regarded mostly as useful for illustration purposes only, though since it is based on graph rewriting it is much more adaptable. Prolog is chosen so that the computational semantics of the language being implemented is familiar to the reader, and thus the reader is not distracted from the explanation of how the TAM-Rewrites are used.

3.3.1 SLD Resolution

Before describing the implementation in detail, it is useful to describe the principle behind the Prolog computational model. What is given here is a summary of the description found in [Hog91]. In general a first order logic program will consist of a set of *clauses*, which are disjunctions of positive or negative *literals*. A positive literal is a predicate of the form $p(a_1,a_2,...)$ and a negative literal is a negated predicate of the form $\neg p(a_1,a_2,...)$. The basic computation mechanism is that of *resolution.* This begins with finding two clauses which share a literal, positive in one clause and negative in the other.

The resolution step then consists of applying the following rule, where the RHS is referred to as the *resolvent*.

$$\boldsymbol{a} \vee \textit{rest-of-clause1} \text{ and } \neg\boldsymbol{a} \vee \textit{rest-of-clause2}$$
$$\rightarrow \textit{rest-of-clause1} \vee \textit{rest-of-clause2}$$

Asking a query involves applying resolution to the negated query, and determining if an empty clause (a contradiction) may be produced as the resolvent after repeatedly applying resolution. In this case the query succeeds and we return TRUE as the answer. If not, then the query fails and the answer FALSE is returned. As part of the resolution, variables in the query may be instantiated, and these variable instantiations should be returned as part of a successful answer. The basic computation mechanism used in Prolog is a version of this, called *SLD resolution*, the letters standing for:

- *Selection* of a literal, which in the context of GRS means the use of activation markings ! to select a function (representing a literal) for rewriting. The order in which activation markings are applied and evaluated decides the computation rule. For Prolog, the computation rule states that the activations are applied strictly left to right.

- *Linear* strategy, meaning that at each stage we only consider literals in the last resolvent for further resolution. Thus we can use a graph to store the query, and by rewriting that graph we produce the resolvent, which may in turn be rewritten again, since we only ever need the most recent resolvent.

- *Definite* clauses are used to store the program, which are clauses with just one positive literal (i.e. of the form $a \vee \neg b \vee \neg c \vee ...$). This means they can all be written as conditional rules (a *if* $b \wedge c \wedge ...$), and then in turn as rewrite rules (of the form $a \rightarrow b \wedge c \wedge ...$).

We shall store the conjunction of literals as a composition of and/2 functions, where the clause $b \wedge c \wedge ...$ becomes and(b,and(c,...)). SLD resolution with a left-most selection rule will entail attempting to resolve the first argument of the principal and/2 function (b in this case). The use of the *closed world assumption* means that any reduction which does not eventually result in the empty clause (answer to query is TRUE) is treated as having failed (query is FALSE). Thus to resolve an and/2 function we need the following rules:

```
and(x,y) -t-> !try_and(!x,y).
try_and(FALSE,x) -t-> FALSE.                    (1)
try_and(TRUE,x) -t-> !x.                        (2)
```

```
try_and(or(x,tail),y) :-
        find_level(var_level),
        tail:=y
        -t-> !backtrack(x,var_level).                (3)
try_and(x,y) :-
        write("failing on "),
        write(x),nl()
        -t-> FALSE.                                  (4)
```

(1) If any literal of the current clause can not be resolved, then the whole clause can not be resolved.

(2) If the first literal resolves to the empty clause, then consider the next literal for resolution.

(3) If there is a choice of rewrites available for the left-most literal, then consider each in turn by the use of backtracking. The or/2 function stores a list of the alternative versions of the clause after the left-most literal has had one resolution step applied. If the left-most literal b may be reduced to b_1,b_2,... then this is of the form [and(b_1,tail),and(b_2,tail),...]. The backtracking is conducted by backtrack/2, which requires the current state of the logical variables to be given, which is supplied by the find_level/1 predicate.

(4) If the left-most literal can not be resolved to TRUE, then the clause can not be resolved, so return failure.

3.3.2 Prolog Predicates

Built-in predicates are implemented by rewriting to TRUE if they hold, and FALSE if not. For example, to implement the Prolog unification predicate = (which here is encoded in graphical form using an eq/2 function) we use the TAM-Rewrite unification predicate, and if it succeeds as part of the condition of the rule then we resolve the eq/2 to TRUE, and otherwise return FALSE.

```
eq(a,b) :- a=b -t-> TRUE.
eq(a,b) -t-> FALSE.
```

Side-effects, such as the printing of terms provided by the *write* predicate is achieved by similar side-effects being used in the condition part of the rewrite rule. For the case of the Prolog *write*, there is a identical write/1 predicate provided for the conditions of TAM-Rewrites. Note that predicates such as write/1 can not fail, so no second rule is needed to deal with the possibility of failure of the condition.

```
p_write(a) :- write(a) -t-> TRUE.
```

User defined predicates are directly implemented as rewrite rules. For example, the user defined predicate *both2(x,y) :- x=y,y=2* can be written as follows:

```
both2(x,y) -g-> !and(eq($x,$y),eq($y,2)).
```

Two particular features of this rule should be noted. Firstly, it uses graph rewriting (as opposed to term rewriting), since it has unrestricted pattern matching in the head and does not check the rule arguments in the condition, and thus meets the criteria of Theorem 2.3. Secondly, in a graph rewrite rule, any of the head variables which appear in the body should normally be prefixed with the $ operator. This ensures that any variable node matched is used in construction of the body, as opposed to the usual action which is to use the variable node's current instantiation. This is important when we use graph rewriting, since there is the possibility that the body graph is going to be shared between several disjuncts, which will be executed by backtracking. The use of backtracking requires the actual variables to be present in the graph, and not the instantiations found during rewriting of the first disjunct.

3.3.3 Posing a Query and Returning the Result

The typical Prolog interpreter will accept as input a query goal, and then return to the user the result TRUE for each way it can resolve that goal to the empty clause, listing any variable instantiations it made during resolution. We may achieve a similar result by appending any input query with end_of_resolution/0, the rewriting of which should list the instantiations of the input variables before returning the result FALSE, thus causing backtracking to occur and the next alternative to be tried. Prolog queries may then be made using prolog_query/2, which records the variables of the query in a global variable @visible_vars, before appending the query clause with the end_of_resolution/0, and initiating resolution. The condition of end_of_resolution/0 then uses @visible_vars to decide which variables to print before rewriting to FALSE.

```
prolog_query(query,var_list) :-
        @visible_vars:=var_list
        -t-> !and(query,end_of_resolution()).
end_of_resolution() :-
        display_vars(@visible_vars)
        -t-> FALSE.
```

We now have enough of the implementation presented to give in Example 3.8 the resolution of a simple goal which does not involve the use of backtracking.

Example 3.8 : Evaluation of a Prolog query *both2*(*x,y*)

```
prolog_query(both2(x,y),[x,y])
-t-> !and(both2(x,y),end_of_resolution())                          (1)
-t-> !try_and(!both2(x,y),end_of_resolution())                     (2)
        !both2(x,y)                                                (3)
        -t-> !and(eq(x,y),eq(y,2))                                 (4)
        -t-> try_and(!eq(x,y),eq(y,2))                             (5)
        -t-> try_and(TRUE,eq(y,2))                                 (6)
        -t-> !eq(y,2)                                              (7)
        -t-> TRUE                                                  (8)
-t-> !try_and(TRUE,end_of_resolution())                            (9)
-t-> !end_of_resolution()                                         (10)
-t-> FALSE                                                        (11)
```

(1–2) Begin by appending end_of_resolution/1 and starting the left to right selection rule.

(3–8) Looking at the evaluation of both2/2 in isolation, it results in the instantiation of both x and y with 2.

(9–10) First literal has resolved to TRUE so move on to next.

(11) Print out instantiations of x and y, then return result FALSE which would cause backtracking to occur if there were any alternatives to try.

3.3.4 Backtracking

Backtracking involves attempting each of the alternative rewrites of the left-most literal. The backtrack/2 function expects a list of alternative versions of the current clause. Rule (2) causes each disjunct to be rewritten in turn, and the variables returned to their previous state (using return_to_level/1) before attempting the next alternative. Rule (1) states that failure is returned when all alternatives have been exhausted. The function as a whole succeeds if any of the alternatives succeed.

```
backtrack([],var_level) -t-> FALSE.                                (1)
backtrack([a|b],var_level) :-
        res := !a,
```

```
        return_to_level(var_level)
        -t-> !make_or(res,!backtrack(b,var_level)).                    (2)

    make_or(FALSE,x) -t-> x.
    make_or(TRUE,x) -t-> TRUE.
```

Example 3.9 gives the sequence of steps followed by the TAM Prolog interpreter in the evaluation of a simple Prolog disjunction.

Example 3.9 : Evaluation of the query *(x=1;x=2),y=x*

```
    !and( or( [and(eq(x,1),tail), and(eq(x,2),tail)] , tail ),
                       and(eq(y,x),end_of_resolution()) )
    -t-> !try_and(!or( [and(eq(x,1),tail), and(eq(x,2),tail)] , tail ),
                       and(eq(y,x),end_of_resolution()))                (1)
    -t-> !backtrack( [and(eq(x,1),and(eq(y,x),end_of_resolution())),
                       and(eq(x,2),and(eq(y,x),end_of_resolution()))], 0)   (2)
        !and(eq(x,1),and(eq(y,x),end_of_resolution()))
        -t-> !try_and(!eq(x,1),and(eq(y,x),end_of_resolution()))       (3)
            !eq(x,1)
            -t-> TRUE                                                  (4)
        -t-> !try_and(TRUE,and(eq(y,x),end_of_resolution()))           (5)
        -t-> !and(and(eq(y,x),end_of_resolution())                     (6)
        -t-> !try_and(!and(eq(y,x),end_of_resolution())                (7)
            !eq(y,x)
            -t-> TRUE                                                  (8)
        -t-> !try_and(TRUE,end_of_resolution())                        (9)
        -t-> !end_of_resolution()                                      (10)
        -t-> FALSE                                                     (11)
    -t-> !make_or(FALSE,!backtrack( [and(eq(x,2),
                                 and(eq(y,x),end_of_resolution()))],0)   (12)
        !and(eq(x,2),and(eq(y,x),end_of_resolution()))
        ....
        -t-> !end_of_resolution()                                      (13)
        -t-> FALSE                                                     (14)
    -t-> !make_or(FALSE,!make_or(FALSE,!backtrack([],0)))              (15)
    -t-> !make_or(FALSE,!make_or(FALSE,FALSE))                         (16)
    -t-> FALSE
```

(2) When try_and/2 finds an or/2 function it uses the graph rewrite redirection := to make all the tails of the various disjuncts point to the remainder of the input query. This is a safe operation, since we are in effect making the rewrite and(or(a,b),tail) -g-> or(and(a,tail),and(b,tail)), the only non-variable in the head being the or/2 functor, which can only be produced as the consequence of this rule or a user rule. Both of these are produced by graph rewriting, and so Theorem 2.3 is satisfied.

backtrack/2 will now consider each of the disjuncts in turn, and return the variables to level 0 (as found by the find_level/1 in step 2) before attempting each disjunct. In this way the parts of the disjunct which are both shared and changed will be returned to their original state before the evaluation of each disjunct. The exception is where a rule can be applied using graph rewriting in accordance with Theorem 2.3, which in practice mostly refers to the user supplied rules with their unrestricted patterns.

(3-10) The various parts of the conjunction are evaluated. Only the variables are overwritten by graph rewrite operations. The end_of_resolution/0 causes the bindings of the input query variables (x and y both equal to 1 here) to be displayed before the result FALSE is returned. At this stage the second disjunct will have been affected by the changes in variable binding, and will thus be and(eq(1,2),and(eq(1,1), end_of_resolution())).

(11) The condition of the backtrack/2 function now returns to their previous values all variables bound during the rewriting of the first disjunct. Thus the second disjunct is in its original state (with x and y as free variables) before it is evaluated.

(12-14) As for (3-10), except this time the variables x and y are both bound to 2.

(15-16) The result of whole disjunction is failure — Prolog only gives its results by side-effects during evaluation, and returns FALSE when all alternatives have been exhausted.

To illustrate how disjunctions in user defined rules (i.e. multiple clauses in a predicate definition) are achieved, Example 3.10 shows how the well-known Prolog program to compute *Fibonacci* numbers is compiled into a TAM-Rewrite rule.

Example 3.10 : Compilation of a Prolog program to compute Fibonacci numbers

Prolog	*TAM-Rewrite*
fib(1,1). *fib(2,1).* *fib(x,y):-* *x>2,* *xp is x−1,* *xpp is x−2,* *fib(xp,yp),* *fib(xpp,ypp),* *y is yp+ypp.*	fib(x,y) -g-> or([and(eq($x,1),and(eq($y,1),tail)), and(eq($x,2),and(eq($y,1),tail)), and(gt($x,2),and(p_is(xp,$x−1), and(p_is(xpp,$x−2), and(fib(xp,yp), and(fib(xpp,ypp), and(p_is($y,yp+ypp),tail))))))], tail).

3.3.5 Cut

The *cut* predicate of Prolog has the effect of removing from the search space all further alternatives for the current clause. To achieve this operation in our implementation we must modify the backtrack/2 predicate to stop trying the alternatives when a mark(m) function has had its single argument m changed from a variable to any ground value. We can then achieve the cut by providing a cut(m) function which when rewritten grounds the variable m that it shares with mark(m). This leads to two extra rules being inserted into the backtrack/2 definition, the first of which stops backtracking when the mark is ground and frees the mark in case it is used in some other disjunct, and the second extra rule causes the mark to be ignored if not ground.

```
cut(m) :- m:=TRUE -t-> TRUE.

backtrack([],var_level) -t-> FALSE.
backtrack([mark(m)|b],var_level) :-
        nonvar(m),
        make_free_var($m)
        -t-> FALSE.
backtrack([mark(m)|b],var_level) -t-> !backtrack(b,var_level).
backtrack([a|b],var_level) :-
        res := !a,
        return_to_level(var_level)
        -t-> !make_or(res,!backtrack(b,var_level)).
```

In Example 3.11 the fib/2 example from Example 3.10 is modified to include pairs of cut/

1 and mark/1 functions.

Example 3.11 : Compilation of a Prolog program using cuts to compute Fibonacci numbers

cfib(1,y):-!,y=1. *cfib(2,y):-!,y=1.* *cfib(x,y):-* *xp is x−1,* *xpp is x−2,* *cfib(xp,yp),* *cfib(xpp,ypp),* *y is yp+ypp.*	cfib(x,y) -g-> or([and(eq($x,1),and(cut(m1),and(eq($y,1),tail))), mark(m1), and(eq($x,2),and(cut(m2),and(eq($y,1),tail))), mark(m2), and(p_is(xp,$x-1),and(p_is(xpp,$x-2), and(cfib(xp,yp),and(cfib(xpp,ypp), and(p_is($y,yp+ypp),tail)))))], tail).
Prolog	*TAM-Rewrite*

3.3.6 Negation By Failure

In Prolog the principle of finite failure is used to compute negation. A clause is regarded as failing, and thus its negation succeeding, only if resolution of the clause results in failure in some finite resolution sequence. In order that this form of negation is sound, a restriction is placed on its use, called the *safe computation rule*. This states that the clause must be ground, and that the computation of the negation is completed in isolation from the resolution of the remainder of a query. With this rule, the negation is usually called *negation by failure*. In practice the restriction on the call being ground is not always enforced, so we must deal with variables being present in the call.

To implement negation by failure, we record in rule (1) below the current state of variable bindings by a call to find_level/1 before rewriting the body of the negation. If the result is TRUE, then we return FALSE as the result of the negation in rule (2), otherwise the resolution failed and we return TRUE in rule (3). In either case, before returning the result, any variables bound in the resolution must be returned to their previous values by calling return_to_level/1 with the variable state found by rule (1).

```
not(x) :-
        find_level(vars)
        -t-> !try_not(!x,vars).                                  (1)
```

```
try_not(TRUE,vars) :-
        return_to_level(vars)
        -t-> FALSE.                                   (2)
try_not(x,vars) :-
        return_to_level(vars)
        -t-> TRUE.                                    (3)
```

3.4 An Implementation of the PLL

The *Pure Logic Language* (*PLL*) [Bab91] was the result of an Alvey project of the same name. The PLL was intended to be data query language with entirely declarative semantics, based on first order classical logic. In addition, it was intended to limit the knowledge of the computational semantics of the language necessary for the construction of queries. The result was to a large extent a full logic programming language, which for many problems is as practical as a pure Prolog language would be.

The basis of the PLL is the transformation of logic formulae to equivalent formulae which are in some sense 'more concrete' than the originals. The transformations are based on the common axioms of first-order logic plus rules for user defined predicates. Some features of the PLL which differentiate it from Prolog are:

- No use of the closed world assumption is made. Instead the notion of a formula of the language being in normal form is used. A step by step reduction process is applied to formulae, and if at any step the system can not proceed then the formula is returned as the most reduced version. This contrasts with Prolog which will fail unless it can resolve a query to the empty clause. For example, the PLL query $\exists x.\ x=2 \wedge y>x$ would give $y>2$ as the result.

- A corollary of not using the closed world assumption is that negation by failure is not used, since the evaluation of an expression can never 'fail', only reach a normal form. For example, if the predicate $p(x)$ is undefined, then the PLL query $p(x)$ will return $p(x)$, and the query $\neg p(x)$ returns $\neg p(x)$. In Prolog the results of these queries would be *false* and *true* respectively, with the interpreter possibly warning that the predicate is undefined and contains a free variable in the negation.

- The order of terms in a conjunction is in general not critical; a term may be 'skipped over' if it can not be completely reduced to variable bindings, and returned to later after other terms have been evaluated. For example, the PLL query $x=y+2 \wedge y=2$ would give $x=4 \wedge y=2$ as the result, whilst in Prolog the query would fail due to the left literal being attempted first, failing and thus causing the whole query to fail.

It may be realized that features of the PLL language described above are naturally supported by a graph rewriting implementation. For example, the necessity to be able return any type of formula (rather than just a conjunction of variable bindings) is readily supported by using graph rewriting, since if at any stage no rule applies then the applicative-order normal form has been reached, and the result is its most reduced form.

A detailed description of the PLL language will not be presented in this thesis, but an overview of the language can be found in [Bab91], and a guide to using the language with some example programs is available [McB88]. The key aspects of the language will be presented here by giving examples of PLL programs, queries and answers as appropriate to the particular aspect of the implementation being described. A complete listing of the PLL implementation is presented in Appendix C.

3.4.1 Conjunctions

The computation of conjunctions in the PLL may be conducted in a similar manner to the SLD resolution of Prolog, except we must now allow for literals which do not rewrite to the empty clause being suspended in some manner while we try to evaluate some other literal, instead of saying immediately that the computation failed. For example, the following is a PLL query and corresponding answer:

$! \, \mathbf{x}{<}\mathbf{y}, \mathbf{x}{=}1, \mathbf{y}{=}\mathbf{x}{*}2$

$\rightarrow \mathbf{x}{=}1, \mathbf{y}{=}2$

During execution the $\mathbf{x}{<}\mathbf{y}$ literal will be rewritten first, but is found to be in normal form. Next $x{=}1$ will be reduced to *TRUE* with the substitution $1/\mathbf{x}$, and then $\mathbf{y}{=}\mathbf{x}{*}2$ reduces to *TRUE* with the substitution $2/\mathbf{y}$. We then attempt the $\mathbf{x}{<}\mathbf{y}$ literal again (which is now $1{<}2$), and find it now reduces to *TRUE*. In practice, this trying of alternative literals is achieved by the use of the following algorithm for the evaluation a conjunction *a,b*.

(i) Rewrite *a* to its normal form
(ii) Rewrite *b* to its normal form
(iii) If the rewrite of *b* caused variables to be bound then rewrite *a* again, otherwise stop
(iv) If the rewrite of *a* caused variables to be bound then rewrite *b* again, otherwise stop
(v) Goto step (iii)

The test of whether variables have been bound in each of steps (iii) and (iv) is because it is only worthwhile attempting to rewrite an expression if we know that it might not be in normal form. This could only occur if the graph has changed since it was last rewritten, which could only be due to variables being bound. This algorithm is effected by the following sequence of rules:

```
and(a,b) -t-> !s1(!a,b,–1).                                            (1)
s1(TRUE,b,l) -t-> !b.                                                  (2)
s1(FALSE,b,l) -t-> FALSE.                                              (3)
s1(a,b,l) :- find_level(new_level),new_level=\=l -t-> !s2(a,!b,new_level).   (4)
s1(a,b,l) -t-> and(a,b).                                               (5)
s2(a,TRUE,l) -t-> !a.                                                  (6)
s2(a,FALSE,l) -t-> FALSE.                                              (7)
s2(a,b,l) :- find_level(new_level),new_level=\=l -t-> !s1(!a,b,new_level).   (8)
s2(a,b,l) -t-> and(a,b).                                               (9)
```

Essentially we try the left conjunct using s1/3 rules and the right conjunct using s2/3 rules. The third parameter to the rule specifies the variable binding level before rewriting of the conjunct was attempted, and is used in rules (4) and (8) to decide if the other conjunct needs rewriting attempted again. The value of –1 supplied for this parameter in rule (1) ensures that the right-hand conjunct is attempted at least once by rule (4). Rules (5) and (9) reconstruct the conjunction as the normal form if the rewriting does not need to be attempted again.

Example 3.12 : Evaluation of the PLL query $x{=}y{+}2 \wedge y{=}2$

```
!and(eq(x,add(y,2)),eq(y,2))
---> !s1(!eq(x,add(y,2)),eq(y,2),–1)                    (by 1)      (10)
        !eq(x,add(y,2))
        ---> eq(x,add(y,2))                             (eq rules)  (11)
---> !s1(eq(x,add(y,2)),eq(y,2),–1)                                 (12)
---> !s2(eq(x,add(y,2)),!eq(y,2),1234)                  (by 4)      (13)
        !eq(y,2)
        ---> TRUE                                       (eq rules)  (14)
---> !s2(eq(x,add(2,2)),TRUE,1234)                                  (15)
---> !eq(x,add(2,2))                                    (by 6)      (16)
---> TRUE                                               (eq rules)  (17)
```

(10–12) Reduction of the left conjunct is attempted first, but results in no change. Once rule (4) has executed once, the binding level parameter has its true value.

(13–15) The right hand conjunct is attempted, and reduces to TRUE whilst instantiating y to 2.

(16) Since one of the conjuncts has reduced to a truth value we now need only consider the other conjunct.

(17) This reduces to TRUE instantiating x to 4.

3.4.2 Equality & Arithmetic

The PLL unification *=/2* predicate is to a large extent equivalent to its Prolog namesake, except in its handling of the arithmetic operators. Here it behaves more like the Prolog *=:=/2*, where the two arguments are treated as arithmetic expressions to be evaluated. Thus in rules (4) and (5) below we use the TAM-Rewrite built-in unification to perform the function of the PLL *=/2*, with the exception of arithmetic operators being made using rules like (2) and (3). Just the rules for subtraction are presented here, but a full listing is given in Appendix C.

```
eq(x,y) -t-> !make_eq(!x,!y).                    (1)

make_eq(z,sub(x,y)) -t-> !make_sub(z,x,y).       (2)
make_eq(sub(x,y),z) -t-> !make_sub(z,x,y).       (3)
make_eq(x,y) :- y=x -t-> TRUE.                   (4)
make_eq(x,y) -t-> FALSE.                         (5)
```

The PLL can instantiate a free variable appearing at any position of an equality involving arithmetic operators, provided that there is at most one free variable in that equality. Thus the PLL is capable of performing the following reduction:

$$! \, \mathbf{x}=4*4 \wedge 8+\mathbf{x}=\mathbf{y}-2$$
$$\rightarrow \mathbf{x}=16 \, , \, \mathbf{y}=48$$

The manner in which the evaluation of an arithmetic operator in a unification predicate is conducted, works on the principle that the formula should be rearranged to have the single free variable on one side of the equality, and the ground values on the other. Rules (6–8) below will perform a reduction of any sub/2 operator if the two arguments are ground. If the equality had just one free variable, then after the application of (1,6–8) it must be in the form $v=c_1$ or $c_1=v-c_2$ or $c_1=c_2-v$, where c_1 and c_2 are integer constants and v is any graph (but we hope a variable). The first case would be handled by (4) and (5) above, the last two cases by rules (9) and (10) respectively. If none of these forms has been reached then there must have been more than one free variable, so we return the reduced form of the equality in rule (11).

```
sub(x,y) -t-> !make_sub(!x,!y).                          (6)
make_sub(x,y) :- z takes x−y -t-> z.                     (7)
make_sub(x,y) -t-> sub(x,y).                             (8)

make_sub(x,y,z) :- yval is x+z -t-> !make_eq(yval,y).    (9)
make_sub(x,y,z) :- zval is y−x -t-> !make_eq(zval,z).    (10)
make_sub(x,y,z) -t-> eq(x,sub(y,z)).                     (11)
```

3.4.3 Conditional Operators

Conditional predicates are like those of Prolog, except if the arguments are not integers we leave the condition unchanged in the graph, instead of using the closed world assumption to say that it is false. Thus rules (1) and (2) below for less/2 cause a reduction only when their arguments are integers, otherwise the less/2 node in the graph is left unchanged (the condition x<y of rule (1) would fail if either x or y was not an integer).

```
less(x,y) :- x<y -t-> TRUE.                              (1)
less(x:*integer,y:*integer) -t-> FALSE.                  (2)
```

3.4.4 Posing a Query and Returning the Result

Asking a PLL query requires an additional argument specifying the free variables of the query, so that the instantiation of these variables will be returned as part of the answer to the query. The pll_query/2 rule sets the global variable @visible_vars to hold this list of free variables, @delay_or to hold the disjunctions found so far in the query to be empty, and @inside_not to be FALSE indicating that what is currently being evaluated is not the body of a negation.

```
pll_query(f,var_list) :-
        @visible_vars:=var_list,
        @delay_or:=[],
        @inside_not:=FALSE,
        -t-> !encapsulate(f,[]).
```

The encapsulate/2 rule serves as the entry point of the rewrite rules to evaluate a PLL query. Its basic steps are:

(i) Find the current state of variable bindings.

(ii) Rewrite the query f, placing any disjunctions found in @delay_or.
(iii) Use rewrite_delay_or/2 to empty @delay_or, building a disjunction of answers to the query as each disjunct held in the global variable is rewritten.
(iv) Restore the variable bindings to their previous state, thus making the evaluation of the query leave any parts of the query graph unchanged, except for the application of graph rewrite rules in accordance with Theorem 2.3.

```
encapsulate(f,tail) :-
        find_level(btp),
        @delay_or:=tail,
        fprime:=!f,
        fprime:=!rewrite_delay_or(@delay_or,fprime),
        return_to_level(btp)
        -t-> fprime.
```

Note how fprime can be used of the RHS of :=/2 to obtain a reference into the graph, and on the LHS as a logical variable to be updated. (The TAM-Rewrite language assumes the $ in front of the LHS of the :=/2 and ::=/2 predicates).

3.4.5 Disjunctions

In essence, the PLL handles queries by progressively reducing them to *disjunctive normal form* (DNF). This is achieved by associating a second graph with a query, stored in the global variable @delay_or, which represents disjuncts that are to be added to the DNF we are constructing. This is encoded as a list of all the disjuncts we have found during rewriting, and DNF is produced by repeatedly taking the first disjunction off this list and rewriting both disjuncts to normal form until the list is empty. The rewriting of a disjunction just needs to place the disjunction on @delay_or, which will be used later by rewrite_delay_or/2 in the encapsulate/2 rule to produce the DNF form of the query. Example 3.13 illustrates this process.

Example 3.13 : Use of @delay_or in evaluating $(\boldsymbol{x}=2 \vee \boldsymbol{x}=3) \wedge (\boldsymbol{y}=4 \vee \boldsymbol{y}=5)$

(i) A new PLL query begins with @delay_or empty, and will add the two disjuncts of the example query to @delay_or:

$! (\boldsymbol{x}=2 \vee \boldsymbol{x}=3) \wedge (\boldsymbol{y}=4 \vee \boldsymbol{y}=5)$	@delay_or=[]
$\rightarrow TRUE\ , (\boldsymbol{y}=4 \vee \boldsymbol{y}=5)$	@delay_or=[x=2,x=3]
$\rightarrow (\boldsymbol{y}=4 \vee \boldsymbol{y}=5)$	@delay_or=[x=2,x=3]
$\rightarrow TRUE$	@delay_or=[y=4.y=5,x=2,x=3]

(ii) We then take one of the disjunctions off @delay_or, and rewrite both of its disjuncts to normal form. For example, having removed $\boldsymbol{y}=4 \vee \boldsymbol{y}=5$, its first disjunct has the normal form *TRUE*, with the substitution $4/\boldsymbol{y}$:

! ($\boldsymbol{y}=4$)	@delay_or=[x=2,x=3]
→ *TRUE*	@delay_or=[x=2,x=3]

(iii) For each of the disjuncts of step (ii) take the last disjunction off @delay_or, and reduce both of its disjuncts to normal form. Again looking at the first disjunct, we find the normal form *TRUE* with substitution $2/\boldsymbol{x}$:

! ($\boldsymbol{x}=2$)	@delay_or=[]
→ *TRUE*	@delay_or=[]

(iv) Thus one of the disjuncts of the DNF is *TRUE* with the substitutions $4/\boldsymbol{y}$ and $2/\boldsymbol{x}$, which is returned to the user as $\boldsymbol{y}=4 \wedge \boldsymbol{x}=2$. The other disjuncts of the DNF are found in a similar manner by considering the various alternatives at steps (ii) and (iii), resulting in a DNF form:

$$(\boldsymbol{y}=4 \wedge \boldsymbol{x}=2) \vee (\boldsymbol{y}=4 \wedge \boldsymbol{x}=3) \vee (\boldsymbol{y}=5 \wedge \boldsymbol{x}=2) \vee (\boldsymbol{y}=5 \wedge \boldsymbol{x}=3)$$

To implement this process as rules, we have in rule (1) below the pushing of the disjunction onto the list of delayed evaluations @delay_or as the only action needed for reducing or/2. The rule rewrite_delay_or/2 should be used with the list of delayed disjunctions as its first argument, and the current reduced form of the query as its second argument. It thus has four situations to deal with in rules (3–6), depending on whether or not there are any disjunctions waiting evaluation, and whether or not the query reduced to TRUE. Rule (2) provides an optimization that if the query has already resulted in FALSE, then there is no point is evaluating the various disjuncts further. Rules (3) and (4) deal with the case where there are more disjuncts to try, using the rules for lr_or/2 to force strict left to right evaluation, so that we are only ever rewriting one disjunct at a time. Rules (5) and (6) return the bindings we have found for the free variables of the query, if we have run out of disjuncts to try.

```
or(x,y) :- @delay_or:=[x,y|@delay_or] -t-> TRUE.                          (1)

rewrite_delay_or(x,FALSE) -t-> FALSE.                                     (2)
rewrite_delay_or([x,y|tail],TRUE)
        -t-> !lr_or(!encapsulate(x,tail),encapsulate(y,tail)).            (3)
rewrite_delay_or([x,y|tail],c)
        -t-> !lr_or(!encapsulate(and(x,c),tail),encapsulate(and(y,c),tail)).   (4)
```

```
rewrite_delay_or([],TRUE) :-
        return_bindings(@visible_vars,and(a,b),eq(c,d),vars)
        -t-> vars.                                                (5)
rewrite_delay_or([],x) :-
        return_bindings(@visible_vars,and(a,b),eq(c,d),vars)
        -t-> !make_and(x,vars).                                   (6)

lr_or(TRUE,x) -t-> TRUE.                                          (7)
lr_or(FALSE,x) -t-> !x.                                           (8)
lr_or(x,y) -t-> !make_or(x,!y).                                   (9)
```

3.4.6 Negation

The PLL negation is equivalent to the negation of first order classical logic, and thus differs radically from the Prolog negation by failure. Rules (1−5) implement some well know theorems of classical logic, and their operation should be self explanatory. Rule (6) causes a new PLL query to be initiated to evaluate the body of the negation, saving the current computational state as represented by the @inside_not and @delay_or global variables before using the encapsulate/2 function to find the normal form of the negation's body. If that has been reduced in any way, then the test \+ new_a==a will succeed, and the reduction of the negation's body repeated until either one of rules (1–5) holds, or the body can not be reduced further.

```
not(TRUE) -t-> FALSE.                                             (1)
not(FALSE) -t-> TRUE.                                             (2)
not(and(a,b)) -t-> !or(not(a),not(b)).                            (3)
not(not(a)) -t-> !a.                                              (4)
not(or(a,b)) -t-> !and(not(a),not(b)).                            (5)
not(a) :-
        tmp_inside_not:=@inside_not,
        tmp_delay_or:=@delay_or,
        @inside_not:=TRUE,
        new_a:=!encapsulate(a,[]),
        @inside_not:=tmp_inside_not,
        @delay_or:=tmp_delay_or,
        \+ new_a==a
        -t-> !not(new_a).                                         (6)
```

The setting of the @inside_not flag is used by the rules which evaluate quantification.

3.4.7 Quantification

The provision of true variable quantification is another aspect of the PLL which makes a radical departure from the Prolog language. Since the language is 'pure' in the sense that no non-logical operations are provided, universal quantification can be implemented in terms of existential quantification using the equivalence $\forall \boldsymbol{x}.\Phi \equiv \neg\exists \boldsymbol{x}.\neg\Phi$ in rule (1).

```
all(x,f) -t-> !not(some(x,not(f))).                          (1)
```

For existential quantification we can use *Skolemization* in rule (2) if the quantification is not part of the body of a negation. (In essence Skolemization says that a existentially quantified variable can be treated like an unknown constant). Rule (3) is another well known equivalence rule of classical logic. Rule (4) tries evaluating the body of the quantification in isolation, and if it reduces to TRUE or FALSE, or the quantified variable becomes instantiated, then we can remove the quantification in rules (5–7).

```
some(x,a) :-
        @inside_not==FALSE
        -t-> !a.                                             (2)
some(x,or(a,b))
        -t-> !or(some(x,a),some(x,b)).                       (3)
some(x,a) :-
        tmp:=@delay_or,
        new_a:=!encapsulate(a,[]),
        @delay_or:=tmp
        -t-> !make_some(x,new_a).                            (4)

make_some(x,TRUE) -t-> TRUE.                                 (5)
make_some(x,FALSE) -t-> FALSE.                               (6)
make_some(x,a) :- nonvar(x) -t-> a.                          (7)
make_some(x,a) -t-> some(x,a).                               (8)
```

3.4.8 User Defined Rules

The PLL user defined rules are, since they state logical directed equivalence rules, almost exactly equivalent to rewrite rules. Thus the PLL rule *define p(...) tobe body* can be translated to the graph rewrite rule *p*(...) -g-> *body*. The PLL allows guards to be added to rules, stating which of the rule parameters must be ground before the rule is invoked. These are easily translated to similar checks in the condition part of a TAM-Rewrite rule. Examples 3.14 and 3.15 illustrate the compilation of two PLL rules.

Example 3.14 : Compilation of the PLL *fib/2* rule

define fib(x,y) guard x ∨ y tobe
x=1 ∧ y=1
∨ x=2 ∧ y=1
∨ some (xp,xpp,yp,ypp) x>2 ∧ xp=x−1 ∧ xpp=x−2 and
fib(xp,yp) ∧ fib(xpp,ypp) ∧ y=yp+ypp

PLL

```
fib(x,y) :- nonvar(x);nonvar(y) -g->
    !or( and(eq(x,1),eq(y,1)),
         or(  and(eq(x,2),eq(y,1)),
              some(xp,some(xpp,some(yp,some(ypp,
                  and(greater(x,2),and(eq(xp,sub(x,1)),
                  and(eq(xpp,sub(x,2)), and(fib(xp,yp),
                  and(fib(xpp,ypp),eq(y,add(yp,ypp)))))))))))
            )
        ).
```

TAM-Rewrite

Example 3.15 : Compilation of the PLL *append/3* rule

define append(x,y,z) guard [x,y] ∨ [z] tobe
x=[] ∧ y=z
∨ some (xh,xt,zt) x=xh::xt ∧ z=xh::zt ∧ append(xt,y,zt)

PLL

```
append(x,y,z) :- nonvar(x),nonvar(y);nonvar(z) -g->
    !or( and(eq($x,[]),eq($y,$z)),
         some(xh,some(xt,some(zt,
           and(eq($x,[xh|xt]),and(eq($z,[xh|zt]),append(xt,$y,zt))))))
        ).
```

TAM-Rewrite

Strictly speaking, the use of a graph rewrite where the rule has a guard breaks Theorem 2.3, since the execution of the rule depends on whether a variable has been instantiated. However, the guards in the PLL only provide a method of optimizing the reduction order; the rule is still equally valid with or without its guards instantiated, and so can be applied even if the guards are not satisfied without effecting the correctness of the implementation.

Summary & Conclusions

This chapter has presented the TAM-Rewrite compiler target language, and demonstrated its use in the implementation of two logic programming languages; Prolog and the PLL. The TAM-Rewrite language has graph rewriting as its computational model, tailored by using the definitions introduced in Chapter 2 for the implementation of logic programming languages.

We may relate the TAM-Rewrite language to Dactl by considering the points where the former differs from the latter:

- logic programming support in the form of unification, trailing and backtracking of variable bindings, and providing both term and graph rewrites in a common environment.
- optimized for sequential machine implementation — reflected by the strict sequencing of rule evaluation and activation markings.
- complex conditions to rules may be specified, including the possibility of defining rules with Prolog-like execution semantics, which must be satisfied before a rule commits.

The next chapter details the abstract machine and T-Code instruction set which supports the efficient execution of TAM-Rewrites, and Chapter 5 details the compilation of TAM-Rewrites into T-Code.

4 Term & Graph Rewriting Abstract Machine

This chapter introduces the ***T**erm and Graph Rewriting **A**bstract **M**achine* (*TAM*) and its *T-Code* instruction set. The instruction set is designed to support the efficient execution of TAM-Rewrite rules, and a close similarity between the operations described by the rewrite rules and those performed by T-Code will be made apparent. A formal description of the compilation from TAM-Rewrite rules to T-Code instructions is given in the next chapter. In this chapter some sketches of the general structure which the compiled form of rules takes will be given, to make apparent how the definitions of the instructions implement various aspects of the TAM-Rewrite rules.

The presentation of the TAM and T-Code begins with an outline of the evaluation mechanism used for rewrite rules, followed by a description of the abstract machine and instruction set. The actions of various key instructions are described using a state transition rule notation like that used in Chapter 1 for the G-Machine and WAM. A complete listing of the transition rules for the T-Code instruction set is given in Appendix D.

4.1 Evaluation Mechanism

Before describing in detail the various instructions of the TAM, an outline will be given in this section of the basic evaluation mechanism used to implement the TAM-Rewrite rules.

The TAM uses a frame based execution mechanism for the evaluation of rules. As is conventional, frames are held on a stack, with the current top of the stack referred to as the TOS. The reduction order in TAM-Rewrites is controlled by having specific activations of functions, which for the purposes of implementation are viewed as being analogous to a function call in a procedural language. For every activation $!head(a_1,...,a_k)$ there will be a *frame* on the stack containing details of which frame was current when the activation was initiated, a reference to the rules which should be tried to match the function *head/k*, and the actual parameters of the activation $a_1,...,a_k$. For each rule of the form $head(f_1,...,f_k)$:- *condition* ---> *body* there will be an associated sequence of T-Code instructions to perform the following operations:

- Alter the frame so that if the pattern matching or condition evaluation should fail then the next rule for *head/k* is attempted.
- Allocate logical variables (i.e. variables which do not appear in the head) in the graph with a pointer to them in the frame.
- Match the formal parameters $f_1,...,f_k$ of the rule head against the actual parameters $a_1,...,a_k$ of the function activated.
- Check that the *condition* holds.
- Construct the *body*, performing in the process rewrites on any function activated within the *body*.
- Return the *body*, indirecting the redex root only if executing a graph rewrite rule.

To give a flavour of the structure a T-Code program will take, Example 4.1 lists the T-Code for the TAM-Rewrite rules for generating Fibonacci numbers already given in Example 3.2. It is suggested that at present the reader should only note the general form the program takes, and return to review the example after having read the more detailed descriptions of the T-Code instructions in the next section.

Example 4.1 : T-Code instructions for fib rules.

```
fib(1) -t-> 1.                                      (1)
fib(2) -t-> 1.                                      (2)
fib(n) :- n>2 -t-> !add(!fib(!sub(n,1)),!fib(!sub(n,2))).   (3)
```

The T-Code for each rule is as follows:

(1)	`NEXT`	`1`	The environment should have just one actual parameter, so throw away anything else, and adjust frame so that T-Code for rule (2) is executed on fail.
	`ARG_COPY`	`1`	For matching the head against the actual parameters, take a copy of the first actual parameter and ...
	`C_INT`	`1`	... compare it with the number 1.
	`INT`	`1`	To build the body of the rule, construct a node with the integer 1 in it.
	`RETURN`		Return result on stack, but do not change redex root. Thus a term rewrite is performed, since only the context of the redex is updated.
(2)	`NEXT`	`1`	*Obvious similarity to rule 1*
	`ARG_COPY`	`1`	
	`C_INT`	`2`	

	INT	1	
	RETURN		
(3)	NEXT	1	
	INT_COPY	1	Check first argument is an integer, and fetch its value.
	PUSH	2	Push the value 2 onto the stack
	GRE		Compare the 2nd from TOS is greater than the TOS. This performs the condition evaluation n>2.
	PREPARE	fib	Get ready a frame to call fib/1 and build its arguments ...
	PREPARE	sub	... the only one of which is a call to sub/2, so get ready a frame for that, and build its arguments ...
	ARG_COPY	1	... the first of which is the first parameter of the current rule,
	INT	1	and the second is the number 1.
	ENACT	2	Make the last prepared frame (i.e. that for sub/2) the current frame with two parameters, and call the T-Code to evaluate the rule for sub/2.
	ENACT	1	The single argument to fib/1 is now evaluated, so make the frame for fib/1 the current one, and evaluate the T-Code for fib/1.
	PREPARE	fib	*Similar argument to that for above recursive call to* fib/1
	PREPARE	sub	
	ARG_COPY	1	
	INT	2	
	ENACT	2	
	ENACT	1	
	REDIRECT	add	The top two elements of the stack are the two arguments for add/2, so re-use the current frame to make a call to add/2.
(4)	OPEN		Final 'rule' for all rewrites causes activated expression to be returned unchanged.

At this point, the principal features to note in Example 4.1 are:

- The `NEXT` instruction which begins each rule resets the stack to contain just the actual parameters to the rule plus variables contained in the head, and uses information stored in the frame to set up the next rule to be evaluated should the current rule fail.

- Each term-rewrite rule ends with `RETURN` or `REDIRECT`, (a graph-rewrite rule would end with `WRETURN` or `REDIRECT`).

- There is always an implicit final rule containing a single `OPEN` instruction to return the activated function unchanged if all other rules have failed.

Similarities may be seen with various aspects of both the G-Machine and WAM described in Section 1.4; in particular the basic graph and stack manipulation operators are essentially those of the G-Machine, whilst the concepts from the WAM of choice-points, environments and the trail are also present. Such similarities will be remarked upon in the following section, which details the abstract machine and instruction set.

4.2 The Abstract Machine and Instruction Set

The TAM may be described by a tuple <C,S,G,T,R> where:

- C is the T-Code sequence being executed,
- S is the stack, containing frames for each rewrite rule,
- G is the current graph, stored in a heap,
- T is the trail of variable bindings made during execution,
- R is a set of registers:
  - F points to the current frame base
  - B is the last choicepoint

In a similar manner to the descriptions used for the G-Machine and WAM in Section 1.4, each T-Code instruction will be described in terms of *state transition rules* upon the tuple representing the machine state. When we speak of S or T we are referring to the top element of that structure, and recording the value of S or T records the position of the top of that structure. Thus if one records F:=S, and pushes three values onto S to get v_3:v_2:v_1:S, then at any future time one may obtain the values from the stack (provided they have not been popped) using the notation ...:v_3:v_2:v_1:F where the ... expresses that any number of other values may have been pushed on top of the values in the meantime. References to the graph G and register set R are placed in square brackets.

To obtain the actual value stored in a graph node n, rather than a pointer to it, the function contents/1 is used. Thus to obtain the values stored in node n we write contents(n). An exception to this rule will be made for code sequences, since it is clear that a code sequence pointer C placed in the code sequence position refers to the contents of the sequence pointed to.

The left hand side of the transition rules read the state of the machine, the right hand specifies the new state. For example, the rule <`INT` *i* ; C, S, G, T, R> ⇒ <C, *n*:S, G[*n*=INT *i*], T, R> states that if `INT` *i* is the first instruction of the current T-Code sequence, we update G to have a new node containing integer *i*, and push onto S a reference to that new node.

To help clarify whether we are naming a T-Code instruction, TAM-Rewrite construct or a variable of the execution rule, the following nomenclature will be used in this and the next chapter.

Name	*Usage*
`INSTR`	typewriter-style typeface for T-Code instructions
rewrite	sans-serif typeface for any TAM expression or a component of the abstract machine
variable	italicized typeface for parts of the rule which match general graphs or arguments to T-Code rules. In particular ...
n_1 n_2 ...	refer to any node in G
a_1 a_2 ...	refer to any nodes in G which are the arguments of a function which the rule pertains to.
i_1 i_2 ...	refer to any nodes in G which contain integers
f g	are functors
j k	are the arity of functions

Table 4.1 : Nomenclature of State Transition and Compilation Rules

Obtaining the T-Code to Execute a Function

The method by which T-Code sequences associated with a rule are stored and retrieved is left to be decided as part of the implementation. For the purposes of our abstract definition of the instruction set the following functions are defined, which return the T-Code sequence for rules with a particular functor *f/k* as their head.

code_for(*f/k*)	returns structure *s* containing all rules for *f/k*
1st(*s*)	returns T-Code sequence for the first rule of a rule structure
next(*s*)	returns the rule structure stripped of the first rule, or if no rules are left it returns the activated functor *f/k*

Identity Nodes

Whenever a graph node is referenced in a state transition rule it is assumed that all *indirection nodes* (IEXP) have been followed, so that when pushing the pointer onto the stack we push a pointer direct to the node that the indirection points to. Thus if the TOS n points to IEXP n_1, and n_1 points to INT 3, a reference to n will give the pointer to INT 3. This behaviour may be overridden by using the *actual node* operator $n, which semantically is equivalent to the $ operator of the TAM-Rewrite language. Thus for the example, $n would return a pointer to IEXP n_1.

For most rewrite rules, we will want to have the indirections followed, since we are usually only interested in what the current graph represents, not how it is represented. Thus most of the T-Code instructions work without using the $ operator. However, when dealing with TAM-Rewrites which use the $, := and ::= operators, we will see instructions which use the $ operator in their defining state transition rules.

4.2.1 Rewrite Evaluation

The evaluation of a function *f* with arity *k* which has been marked for rewriting using the TAM activation operator ! involves sequentially searching the list of rewrite rules associated with *f/k*, checking that the head pattern matches and ensuring that the condition is satisfied. If the condition is not satisfied, then the variables bound during the evaluation of the condition are restored to their previous state before attempting the next rule for the function. This is essentially the evaluation mechanism of Prolog, and the TAM uses a similar *frame* based (alternatively known as *environment* based) approach as the WAM. A frame is created for each attempt at rewriting a function, and is used to evaluate it. The frame so created must remain current whilst all instructions of the T-Code sequence associated with the rule are executed. Calls to subcomputations (i.e. activations of other functions) may temporarily suspend the frame whilst the T-Code for these functions is executed, but the old frame must be restored on returning from the subcomputation.

The first part of the frame is referred to as the *frame header block*, which contains such items as the information needed to continue execution once the associated function has been evaluated, and the next rule to try if the current one should fail. This is immediately followed by the *environment*, which contains the parameters of the function. Specifically, the frame header block contains eight entries, giving the start of the frame for rewrite function $f(a_1,...,a_k)$ the following entries:

- CF : T-Code sequence to execute if current rule does not match. This will usually be the next rule for *f/k*.
- FF : Frame to restore if current rule does not succeed, which usually is the current frame.
- BP : Value of the *choicepoint* register B that was current when rewriting for *f/k*

was initiated.

- VB : Record of the value of T (representing the status of variables bound) just before attempting to rewrite *f/k*. This is used to restore any variables recorded on the trail by a logical bind operation during the evaluation of the condition of one rule before attempting the evaluation of another.
- CS : T-Code sequence which followed the call to evaluate *f/k*, which is executed after the rewriting of *f/k* is complete.
- FS : Frame current at time of call to evaluate *f/k*, which is restored after the rewriting of *f/k* is complete.
- RL : The list of rules for *f/k* which we have not yet attempted to match against *f/k*.
- CL : Current called function, i.e. $f(a_1,...,a_k)$. Used to return the function unchanged if no rewrite is found to match.
- $a_1 \ldots a_k$: The actual arguments of *f/k*.
- $av_1 \ldots av_m$: The variables of *f/k* which appear as non-root nodes of the arguments of *f/k*.
- $v_1 \ldots v_p$: The free variables of *f/k*.

Attempting to rewrite a function must include creating a frame with the header block and actual parameters included. The execution of the pattern matching will then push the *av* variables onto the stack, and the *v* variables are added during execution of the condition. For clarity of presentation, the frame header block will be placed on the stack as a single entry in an ordered set {CF,FF,BP,VB,CS,FS,RL,CL}. Each transition rule will only name those entries that are used or changed, and thus an empty set {} indicates no access to the block. The entries of the block will be shaded in diagrams.

Frame Creation and Deletion

The instruction `EVAL` performs the task of creating such frames, expecting the function *f/k* that is to be evaluated on the TOS, and replacing it with the frame header block and actual arguments of *f/k* (with the first item of the frame header block taking the lowest position on S). The second rule of `EVAL` deals with the case where S has a non-functor node, and thus no action need be taken since the expression must be in normal form, since the TAM-Rewrite language only allows rewrite rules for functions.

`EVAL` <`EVAL` ; C, n:S, G[n=FUNC f n_1, n_1=CONS a_1 n_2,...,
n_k=CONS a_k n_{k+1}, n_{k+1}=NIL], T, R[F,B]>
$\Rightarrow$ <1st(code_for(f)),
a_k:...:a_1:{CF:=1st(code_for(f),FF:=S,BP:=B,VB:=T,
CS:=C,FS:=F,RL:=code_for(f),CL:=n}:S,
G,T,R[F:=S,B:=S]>

<EVAL ; C, n:S, G[n≠FUNC f n'], T, R> ⇒ <C, n:S, G, T, R>

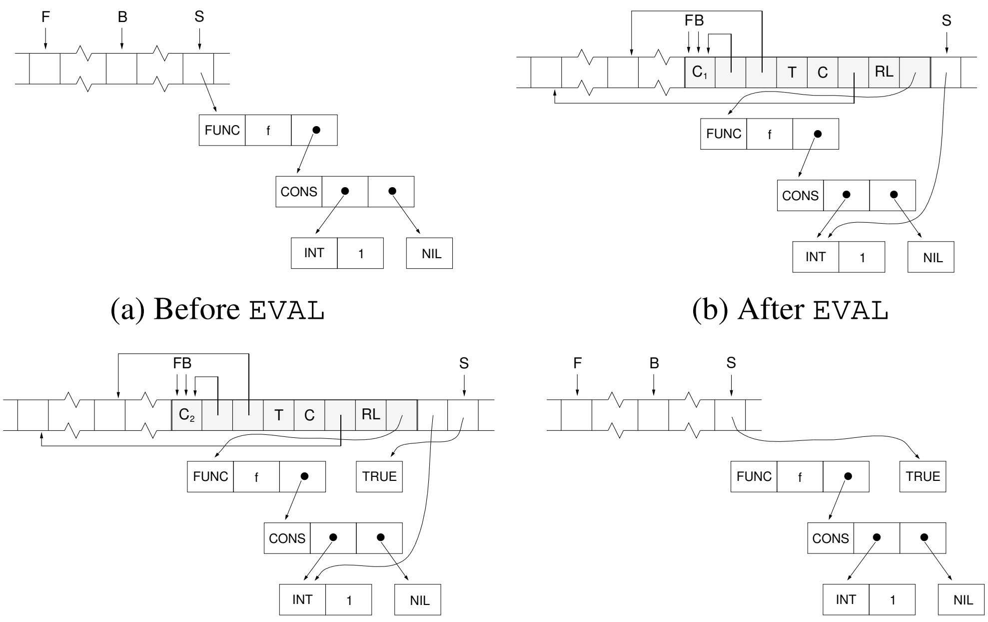
Example 4.2 : Use of EVAL to create frame for !f(1)

Assume that there are two rewrite rules for f/1 :

f(0) -t-> false. (1)

f(1) -t-> true. (2)

The machine state before execution of the EVAL would be <EVAL ; C, S, G, T, R[F,B]> with S and G possibly being as depicted in Figure 4.1 (a). After EVAL, the situation would be that shown in Figure 4.1 (b), where C_1 is the T-Code sequence for the first rule. Only after the execution of a NEXT instruction would this be updated to point at the T-Code for the second rule.



(a) Before EVAL (b) After EVAL

(c) Result of rewrite on TOS (d) After RETURN

Figure 4.1 : Various stages in the application of f(1) -t-> TRUE

Once a rule has been found which matches the graph activated by !, the contractum is built and returned as the result of the rewrite, replacing the frame created by EVAL. Two instructions are provided to perform this operation, the choice of which is used depending on the type of rewrite operation intended. RETURN is used to pass back the result of a term rewrite rule, and leaves the redex unchanged. It uses CS and FS to restore C and F, thus resuming execution at the instruction just after the EVAL which originally initiated the rewrite. WRETURN performs the same operation as RETURN, but in addition will overwrite the redex root with an indirection to the contractum root. The second rule from

WRETURN states that no such indirection is necessary if there was no actual redex — a situation that may exist due to certain optimizations that will be introduced later.

RETURN	<RETURN ; C, n:...:{BP,CS,FS}:F, G, T, R[F]>
	⇒ <CS, n:F, G, T, R[F:=FS,B:=BP]>
WRETURN	<WRETURN ; C, n:...:{BP,CS,FS,CL=h}:F, G, T, R[F]>
	⇒ <CS, n:F, G[h:=IEXP n], T, R[F:=FS,B:=BP]>
	<WRETURN ; C, n:...:{BP,CS,FS,CL=∅}:F, G, T, R[F]>
	⇒ <CS, n:F, G, T, R[F:=FS,B:=BP]>

Example 4.3 : Different results of term and graph rewriting on !f(1)

> Using the term rewrite rules of Example 4.2, Figure 4.1 (c) shows how the body of the rule has been used to construct the contractum TRUE. Since the rule is a term rewrite, the final action of rewriting will be to leave a pointer to the contractum root at the position on the stack occupied by the frame. In this manner we update the context of the activation from being a pointer to the redex root into a pointer to the contractum root, but leave all other pointers to the redex root unchanged. If the rules had been graph rewrites (i.e. had used -g->) then the final state would be that shown in Figure 4.2, where in addition to leaving a pointer on the stack, the root of the redex has been overwritten with an indirection to the contractum.

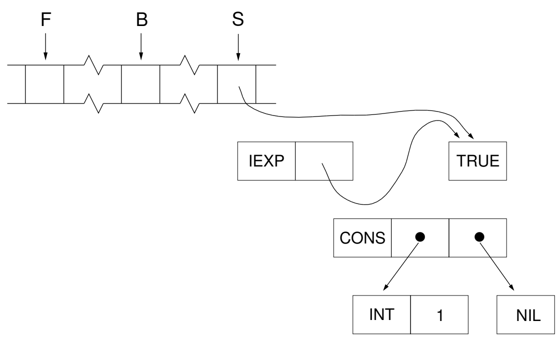


Figure 4.2 : Final stage of application of f(1) -g-> TRUE

If no rule matches the function marked for rewriting, then the function must be returned unchanged, in which case the reference to the function stored in the frame is used to replace the frame on S. This operation is carried out by the OPEN instruction, which behaves in a similar fashion to RETURN. However, certain optimizations that will be introduced later mean that there is not always an original version to return. This case is handled by the second rule for OPEN, which constructs the graph from the parameters held as part of the frame. Note that we do not need a special graph rewrite version (‘WOPEN’) since there is no need to overwrite the function if it has not changed.

OPEN

$$<\mathtt{OPEN}\ ;\ \emptyset(,\ a_k{:}\ldots{:}a_1{:}\{\mathrm{CL}{=}n\}{:}\mathrm{F},\ \mathrm{G}[n{\neq}\emptyset],\ \mathrm{T},\ \mathrm{R}[\mathrm{F}]>$$
$$\Rightarrow\ <\mathtt{RETURN}\ ;\ \emptyset,\ n{:}\{\mathrm{CL}{=}n\}{:}\mathrm{F},\ \mathrm{G},\ \mathrm{T},\ \mathrm{R}>$$
$$<\mathtt{OPEN}\ ;\ \emptyset,\ a_k{:}\ldots{:}a_1{:}\{\mathrm{CL}{=}\emptyset\}{:}\mathrm{F},\ \mathrm{G},\ \mathrm{T},\ \mathrm{R}[\mathrm{F}]>$$
$$\Rightarrow\ <\mathtt{RETURN}\ ;\ \emptyset,\ n{:}\{\mathrm{CL}{=}\emptyset\}{:}\mathrm{F},\ \mathrm{G}[n{=}\mathrm{FUNC}\ f\ n_1,\ n_1{=}\mathrm{CONS}\ a_1\ n_2,\ldots,\ n_k{=}\mathrm{CONS}\ a_k\ n_{k+1},\ n_{k+1}{=}\mathrm{NIL}],\ \mathrm{T},\ \mathrm{R}>$$

We may note the similarity of the `EVAL` and `RETURN` instructions to their namesakes in the G-Machine, except of course for the use of a frame based approach. The need for `WRETURN` is a result of the TAM's separation of the notion of term and graph rewrites, and the need for an `OPEN` instruction is the result of having several alternatives to try during pattern matching, with the possibility of failure causing none of them to hold.

Rule Variables

The variables of a rule may be divided into two classes, being:

- *head variables*, i.e. those which appear in the head of a rewrite rule. These will be matched against subgraphs in the redex, and thus need only a slot on the current frame to contain a pointer to the subgraph matched.
- *body variables*, i.e. those which appear in the condition or body of a rule, but not its head. These are logical variables, since they may persist outside the life-time of the rule frame, and must be allocated both a position in the frame from which they may be referenced and a variable node in the graph.

The head variables will require only that the space be left on the stack when creating a frame header, and the pattern matching process update the gaps left so that they contain pointers to the subgraphs of the redex matched. The body variables are created by a `VAR` instruction, which creates a node with tag `VAR` in the graph, and pushes a pointer to it onto the stack.

VAR

$$<\mathtt{VAR}\ ;\ \mathrm{C},\ \mathrm{S},\ \mathrm{G},\ \mathrm{T},\ \mathrm{R}>\ \Rightarrow\ <\mathrm{C},\ n{:}\mathrm{S},\ \mathrm{G}[n{=}\mathrm{VAR}],\ \mathrm{T},\ \mathrm{R}>$$

Optimizing Body-Root Activations

When the root of the body of a rewrite rule is itself the subject of an activation, i.e. we have the situation f(...) ---> !g(...), we say that the rule has a *body-root activation*. These may be treated in a special way, since there is no possibility of having to return f or use its parameters to build other graphs after rewriting g; the result of the rewrite of f will be either g if no rewrite for g can be found, or the body of a rewrite rule for g. The details of where to jump once g is complete are the same as for f, and thus we can re-use the current frame. To do this, we need to update the rule information held in the frame to be about g instead of f, and update the trail reference so that variables bound during the condition code for f are considered fixed as we try the different rules for g. The `REDIRECT` instruction performs this operation, needing the function name and number of arguments for the function as parameters. The arguments of the new function are assumed to have been placed on S before execution of the instruction.

`REDIRECT` f, k <`REDIRECT` f, k ; C,
a_{x+k}:...:a_{x+1}:a_x:...:a_1:{ }:F,G,T,R[F,B]>
⇒ <1st(code_for(f)),
a_k:...:a_1:{CF:=1st(code_for(f),VB:=T,
RL:=code_for(f),CL:=∅}:F, G, T, R[B:=F]>

Note how the backtrack point is reset to the current frame, thus dropping any backtrack points that may have been generated during evaluation of the condition for the current rule.

Rule (3) from Example 4.1 uses this optimization for the activation of add/2. Once rewriting has reached the point of activating the add/2, there can be no reference to the variables of fib/1 made again, and the result will be the addition of the two arguments to add/2.

Optimizing Non Body-Root Activations

When we have an subgraph of the body of a rewrite rule activated, it will be frequently found that there exists a matching rule for that subgraph. In this situation it is wasteful to construct a function in G for that subgraph only to immediately ‘unwind’ it back onto the stack as a frame which is successfully rewritten. The `PREPARE` and `ENACT` instructions avoid this inefficiency by preparing a new frame whilst the current frame is active (using `PREPARE`), constructing the arguments of the new (but not yet current) frame using the standard method for constructing body subgraphs, and then making the new frame the current one using `ENACT`.

`PREPARE` f <`PREPARE` f ; C, S, G, T, R[B,F]>
⇒ <C, {CF:=1st(code_for(f),FF:=S,BP:=B,VB:=T,
FS:=F,RL:=code_for(f),CL:=∅}:S, G, T, R[B:=S]>

`ENACT` k <`ENACT` k ; C, a_k:...:a_1:{CF}:B, G, T, R[B]>
⇒ <CF, a_k:...:a_1:{CS:=C}:S, G, T, R[F:=S,B:=S]>

Notice how the CL position for the potential redex is filled with ∅, indicating that there is no actual version of the potential redex in the graph. If the function associated with the frame it is to be returned as the result, which would be occur if no rules were found to match, then it must be constructed from the information stored in the frame. It is to serve this purpose that we provided the second rule in the definition of `OPEN`. The definition of the `WRETURN` instruction must also note that if no original graph exists then it can not be updated with an IEXP during a graph rewrite.

If we look again at rule (3) from Example 4.1, we can see that it contains four such activations, two for each of fib/1 and sub/2. Looking at the leftmost activation of fib/1, we see a `PREPARE` instruction followed immediately by another `PREPARE` to make the sub/2 activation. Other instructions then create the arguments for the call to sub/2; in this case it takes a copy of the argument of the fib/1 which is the current frame pushed onto the stack by `ARG_COPY`, and the integer 1 pushed onto the stack by `INT`. The `ENACT` instruction makes current the frame for sub/2, which leaves its result on top of the stack, in the correct position to be the single argument for fib/1 which the next `ENACT` instruction makes the current frame.

Looking at Example 4.1 as a whole, we see that there are no un-optimized activations in the bodies of the rules. This will frequently be the case in practice, since the only situation where a body activation can not be optimized is when it has been applied to a variable in the rule. At runtime the body variable may be bound to a function, and thus be eligible for rewriting, but at compile time we can not tell which frame and arguments to `PREPARE` or `REDIRECT`.

A similar optimization to that provided by `PREPARE` and `REDIRECT` has been suggested for the G-Machine [Pey87], where an environment can be created on the stack, and then placed into position by a SQUEEZE instruction, which slides a group of parameters down the stack. The frame based approach used in the TAM complicates matters, and requires different instructions to deal with different situations.

Sequencing of Rewrite Rules

The frame header block contains an entry CF for the T-Code sequence that should be executed if the pattern matching or condition should fail for the rule we are attempting. Thus the first instruction of each rule for a function should update CF to point to the T-Code for the next rule in the sequence for that function. This operation is performed by the `NEXT` instruction, which will also reset the stack so that the frame contains just the actual arguments and head variables of the function, thus discarding any material left on the stack from previous rules for the function, which may already have been attempted and failed.

`NEXT` i <`NEXT` i ; C, n_k:....:n_i:....:n_1:{RL}:F, G, T, R[F]>
⇒ <C, n_i:...:n_1:{CF:=1st(RL),RL:=next(RL)}:F, G, T, R>

This instruction may be seen to serve a similar function to the various *indexing* instructions such as TRY_ME_ELSE found in the WAM, except the WAM has the address of the fail sequence given at compile time rather than generated at run-time.

Terminating the Graph Rewriting Process

An instruction `END` is provided to terminate the attempts at rewriting the graph, and return the graph at TOS as the only element on the stack.

`END` <`END` ; ∅, n:S, G, T, R> ⇒ <∅, n, G, T, R>

4.2.2 Head Patterns

To facilitate the matching of actual arguments with the formal parameters of the rule, a set of *checking* instructions are provided. The '`C_`' group of instructions perform the checking of a graph in a 'top-down' manner. They assume that the argument to be checked is on the TOS, ensure that it is a certain class of node, and then either check the value field (for leaf nodes such as `INT`) or push the successor nodes onto the stack for further checking.

The `C_NODE`n instructions are used when we have a node in the pattern graph representing an operator (such as function application) with a certain tag and number of arguments. The instructions to match ground values (e.g. `C_INT`) or variables (e.g. `ARG_SET`) are used at the leaves of the pattern graph.

The checking that integers, strings and pointers are of a certain value is performed by `C_INT`, `C_STR` and `C_IMM` respectively. If the TOS is of the correct type and value, then the stack is popped and evaluation continues, otherwise the `FAIL` instruction is initiated.

`C_INT` i <`C_INT` i ; C, n:S, G[n=INT i], T, R> ⇒ <C, S, G, T, R>
<`C_INT` i ; C, n:S, G[n≠INT i], T, R> ⇒ <`FAIL` ; C, S, G, T, R>
`C_STR` s <`C_STR` s ; C, n:S, G[n=STR s], T, R> ⇒ <C, S, G, T, R>
<`C_STR` s ; C, n:S, G[n≠STR s], T, R> ⇒ <`FAIL` ; C, S, G, T, R>
`C_IMM` n <`C_IMM` n ; C, n:S, G[n], T, R> ⇒ <C, S, G, T, R>
<`C_IMM` n ; C, n_g:S, G[n≠n_g], T, R> ⇒ <`FAIL` ; C, S, G, T, R>

A match against a pattern variable is always going to succeed, so all that needs to be done is to set the position on the stack reserved for the particular local variable to be the subgraph matched, which is performed by the `ARG_SET` instruction.

`ARG_SET` i	<`ARG_SET` i ; C, $\$n_k{:}n_{k-1}{:}\ldots{:}n_i{:}\ldots{:}n_1{:}\{\}{:}$F, G, T, R[F]>
	$\Rightarrow$ <C, $n_{k-1}{:}\ldots{:}n_k{:}\ldots{:}n_1{:}\{\}{:}$F, G, T, R>

The following instructions are designed to provide the mechanism for searching and matching the structure of a graph. Each one checks the tag field of the node at TOS, and then pushes the one or two fields of the node onto the stack, which will then be inspected by other checking instructions, or used to bind local variables.

`C_NODE2` t	<`C_NODE2` t ; C, n:S, G[$n{=}t\ n_1$], T, R> $\Rightarrow$ <C, n_1:S, G, T, R>
	<`C_NODE2` t ; C, n:S, G[$n{\neq}t\ n_1$], T, R> $\Rightarrow$ <`FAIL` ; C, S, G, T, R>
`C_NODE3` t	<`C_NODE3` t ; C, n:S, G[$n{=}t\ n_1\ n_2$], T, R>
	$\Rightarrow$ <C, $n_1{:}n_2$:S, G, T, R>
	<`C_NODE3` t ; C, n:S, G[$n{\neq}t\ n_1\ n_2$], T, R>
	$\Rightarrow$ <`FAIL` ; C, S, G, T, R>

Note that no instruction is provided for checking nodes which just have a tag (namely those for TRUE, FALSE and NIL). This is due to the use of an implementation optimization where all instances of a node with a particular tag are represented by a single node created at compile time. Since the address of this node is known at compile time we can use `C_IMM` to check the address rather than looking at the actual value — the address corresponds to the only instance of a node with that tag.

An instruction we need to consider for use during pattern matching is the general TOS `COPY` instruction. This is used where the pattern conjunction operator (‘:’) means we need two instances of the item we are trying to match; one for each of the patterns of the operator.

`COPY`	<`COPY` ; C, $\$n$:S, G, T, R> $\Rightarrow$ <C, $n{:}n$:S, G, T, R>

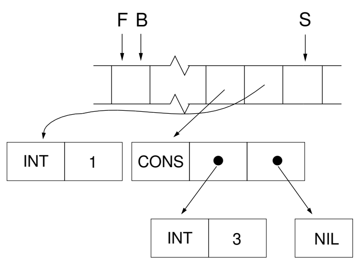
The range of instructions described above is sufficient for the TAM-Rewrite language since the patterns are in a sense first-order; that is we may not represent the name of a function or an operator by a variable in a pattern. For example, we may not use the pattern x y z to match 1+2 and get 1/x, +/y and 2/z. Example 4.4 describes how the structure for a list [x] and an integer 1 are pattern matched. The matching of the list_length/2 is implicit since the representation that the TAM uses for functors means that only the rules associated with a particular functor are ever attempted.

Example 4.4 : Pattern matching the rule head list_length([x],1)

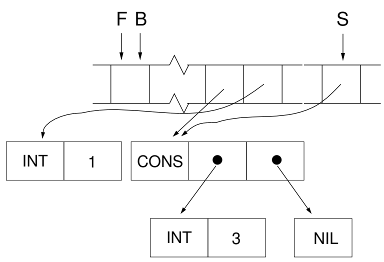
For this example we will consider the sequence of instructions necessary for the rule list_length([x],1) -t-> x, which are as follows:

`NEXT`	`3`	Just kept on the stack the two actual parameters, plus a slot for the variable x. Set-up the fail slot CF to point at the next instance of list_length/2.
`ARG_COPY`	`1`	Going to try and match the first argument of the rule...
`C_NODE3`	`CONS`	Check it is a CONS node, and push the arguments of the node onto S.
`ARG_SET`	`3`	The first argument of the `CONS` is the x variable of the rule, so set the entry for that variable on S to point at the subgraph that is on TOS.
`C_IMM`	`NIL`	The second argument of the `CONS` must be a NIL node.
`ARG_COPY`	`2`	Match the second argument of the rule ...
`C_INT`	`1`	Must be the integer 1.
`ARG_COPY`	`3`	Pattern matching is now complete, so build the body, which is just the local variable x.
`RETURN`		Return result on stack.

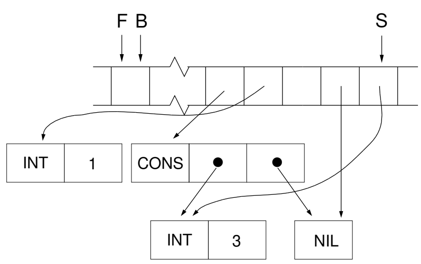
The following sequence of figures illustrates the first few steps of the pattern matching process for the function activation !list_length([3],1).



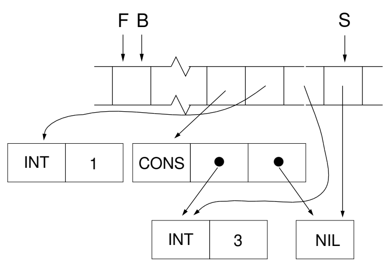
(a) Frame created for activation



(b) After `ARG_COPY 1`



(c) After `C_NODE3 CONS`



(d) After `ARG_SET 3`

Figure (a) shows the stack after the function activation, and the `NEXT` instruction, which leaves a free slot on the stack for the local variable `x`. In (b) the first parameter is copied onto the TOS using `ARG_COPY 1`. It is then checked by `C_NODE3 CONS` that it is a CONS node, which leaves the two arguments of the `CONS` on the stack (c). The first of these is what corresponds to the local variable x, which is set to the value (the number 3 in the example) and leaves the second argument of the `CONS` as TOS in (d).

The G-Machine does not specify pattern matching instructions, since its serves to implement supercombinator reduction. The WAM has the *get* instructions which check the type of the redex parameters, and so may be regarded as similar in action to the pattern matching operators presented above. Since the TAM does not perform unification in head pattern matching, there is no need for the complication of read and write modes found in the WAM *get* instructions.

4.2.3 Graph Construction

The instructions to perform graph construction can be viewed as performing a kind of inverse of the pattern matching process. Based on a pattern of what graph we wish to construct, we use instructions to build the leaves of the pattern graph placing the results on the stack, and then other instructions to build the constructor nodes of the graph with these leaves taken from the stack as arguments.

To build the leaves, a set of instructions analogous to the graph checking instructions are provided, with the situation for variables being somewhat more complicated. Normally if a variable is bound we wish to just use its value in graph construction (ignoring any IEXP nodes) in which case we use `ARG_COPY`. However, the use in the TAM rule of a $ to supply the actual variable node to form part of the new graph (even if it is bound via IEXP) needs the `ARG_DUP` instruction. Global variables are dealt with in a similar manner by `GBL_COPY` and `PUSH`.

`INT` i	$\langle \texttt{INT}\ i\ ; C, S, G, T, R\rangle \Rightarrow \langle C, n{:}S, G[n{=}\mathrm{INT}\ i], T, R\rangle$
`STR` s	$\langle \texttt{STR}\ s\ ; C, S, G, T, R\rangle \Rightarrow \langle C, n{:}S, G[n{=}\mathrm{STR}\ s], T, R\rangle$
`ARG_COPY` i	$\langle \texttt{ARG\_COPY}\ i\ ; C, n_k{:}\ldots{:}n_i{:}\ldots{:}n_1{:}\{\}{:}F, G, T, R[F]\rangle$ $\Rightarrow \langle C, n_i{:}n_k{:}\ldots{:}n_i{:}\ldots{:}n_1{:}\{\}{:}F, G, T, R\rangle$
`ARG_DUP` i	$\langle \texttt{ARG\_DUP}\ i\ ; C, n_k{:}\ldots{:}\$n_i{:}\ldots{:}n_1{:}\{\}{:}F, G, T, R[F]\rangle$ $\Rightarrow \langle C, n_i{:}n_k{:}\ldots{:}n_i{:}\ldots{:}n_1{:}\{\}{:}F, G, T, R\rangle$
`GBL_COPY` n	$\langle \texttt{GBL\_COPY}\ n\ ; C, n_k{:}\ldots{:}n_1{:}\{\}{:}F, G[n], T, R[F]\rangle$ $\Rightarrow \langle C, n{:}n_k{:}\ldots{:}n_1{:}\{\}{:}F, G, T, R\rangle$
`PUSH` n	$\langle \texttt{PUSH}\ n\ ; C, S, G[\$n], T, R\rangle \Rightarrow \langle C, n{:}S, G, T, R\rangle$

Again, the graph construction instructions are analogous to those for checking graph nodes. The `PUSH` instruction serves as '`NODE1`', since all it needs to do is push the address of the shared instance of a particular node with the tag onto the stack (see comment in the previous subsection for a justification of this procedure).

`NODE2` t <`NODE2` t ; C, n_1:S, G, T, R> $\Rightarrow$ <C, n:S, G[n=t n_1], T, R>

`NODE3` t <`NODE3` t ; C, n_2:n_1:S, G, T, R> $\Rightarrow$ <C, n:S, G[n=t n_1 n_2], T, R>

Optimization of Graph Construction

Since function application will play a central role in the creation of expression graphs, it is of obvious importance that we create such applications in an optimal manner. To avoid having a sequence of `CONS` instructions to build up the parameter list, and then a `PUSH` for the function name, we instead provide a single `APPLY` instruction. This will create the parameter list from the items assumed to be pushed onto the stack, and then apply the function name. The instruction takes the rule name and arity as its arguments.

`APPLY` f, k <`APPLY` f, k ; C, n_k:n_{k-1}:...:n_1:S, G, T, R>
$\Rightarrow$ <C, n:S, G[n=FUNC f p_1, p_1=CONS n_1 p_2, ... ,
p_k= CONS n_k p_{k+1}, p_{k+1}=NIL], T, R>

Another common structure is a `CONS` list, and to aid the compact encoding of list construction a `CONS` instruction is provided as a shorthand for `NODE3 CONS`, and a `NIL` instruction as a shorthand for `PUSH NIL`.

Similar optimizations may be made to the G-Machine [Pey87] where there is a MKAP n instruction serving the multiple application of APPLY, as well as the provision of CONS and NIL instructions.

4.2.4 Condition Expression Control

The condition part of a TAM-Rewrite rule is evaluated by a sequence of T-Code instructions for each term in the condition. If any condition fails then the operations of the `FAIL` instruction are executed, which uses the last choicepoint to restore the variable bindings to the state they were when the last choicepoint was created, and then jumps to the T-Code sequence for the alternative of the choicepoint. If all alternatives of the most recent choicepoint have been tried then the previous choicepoint is restored and the `FAIL` attempted again.

`FAIL` <`FAIL` ; C, ...:{CF=`FAIL`,FF,BP}:B, G, T, R[B]>
⇒ <`FAIL` ; C, B, G, T, R[B:=BP,F:=FF]>
<`FAIL` ; C, ...:{CF,FF,VB}:B, G, (v_1,p_1):...:(v_n,p_n):VB, R[F,B]>
⇒ <CF, ...:{}:B, G[v_1:=p_1,...,v_n:=p_n], VB, R[F:=FF]>

A sequence of T-Code instructions in a Prolog type rule (i.e. one of the form *head* :- *condition*) is terminated by the `SUCCEED` instruction, which acts like the `RETURN` instruction for rewrite rules, accept there is of course no contractum to build or return.

`SUCCEED` <`SUCCEED` ; C, ...:{CS,FS}:F, G, T, R[F]>
⇒ <CS, ...:{}:F, G, T, R[F:=FS]>

Disjunctions

To handle disjunctions within a condition code expression, we use the notion of a *choicepoint*. This is like a frame header block, except it does not need the information about what to do on success, nor does it have the variables repeated from the current frame. Thus it contains just the first four items found in the full frame header block. A choicepoint is created by the `OR` instruction, which takes as its argument the address to jump to on fail (i.e. the alternative of the choicepoint).

`OR C'` <`OR C'` ; C, S, G, T, R[F,B]>
⇒ <C, {CF:=C',FF:=F,BP:=B,VB:=T}:S,
G, T, R[B:=S]>

If we consider the TAM-Rewrite expression (condition$_1$;condition$_2$),condition$_3$ then the structure of the corresponding T-Code will be:

`OR` C_2
condition$_1$
`GOTO` C_3
C_2 : condition$_2$
C_3 : condition$_3$

The GOTO instruction has the obvious purpose of jumping to a new T-Code sequence, without making any record of how to return to the previous sequence.

`GOTO` C' <`GOTO` C' ; C, S, G, T, R> ⇒ <C', S, G, T, R>

Figure 4.3 illustrates the situation on the stack when we have just executed the `OR` instruction for the outline structure given above.

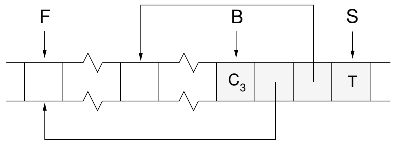


Figure 4.3 : Stack configuration for condition disjuncts

Negation

As in Prolog, a negation in the condition of a TAM-Rewrite language is implemented using *negation by failure*. The structure of the T-Code sequence for the TAM-Rewrite expression (\+ condition$_1$), condition$_2$ will be:

```
      OR C2
      condition1
      DROP_BTP
      FAIL
C2 :  condition2
```

The `DROP_BTP` instruction serves to remove the choice made by the last `OR` instruction of the current frame (ignoring any choicepoints set up during the evaluation of condition$_1$) and allows the following `FAIL` instruction to cause the whole condition to fail unless another choicepoint had been set up (as is the case in Figure 4.3). This gives us the intended semantics of negation by failure; that is if condition$_1$ should fail then \+ condition$_1$ will succeed, and otherwise \+ condition$_1$ will fail.

`DROP_BTP` <`DROP_BTP` ; C, ...:{FF,BP}:B, G, T, R[F≠FF,B]>
⇒ <`DROP_BTP` ; C, B, G, T, R[B:=BP]>
<`DROP_BTP` ; C, ...:{FF,BP}:B, G, T, R[F=FF,B]>
⇒ <C, B, G, T, R[B:=BP]>

We can not expect to find analogies to these instructions in the G-Machine, but do so in the WAM. The latter provides a FAIL instruction with similar behaviour to the T-Code namesake, and evaluates disjunctions and negation in a similar manner (though this is a extension to the language not found in the standard texts on the WAM).

Meta-Level Control

Two instructions are provided to correspond to call/1 and cut/0 predicates of the TAM, which in turn are semantically similar to the normal meta-level control operations found in Prolog. The `CALL` instruction passes the graph on the TOS to the part of the TAM compiler responsible for compiling the condition code of rules, and immediately executes the output. The `CUT` instruction removes all alternatives from the current frame, thus preventing any other rules being attempted for the Prolog clause. Note that this instruction is for use in condition rules only, and can not be used directly in the condition of a rewrite rule since the last instance of rules for a certain function must be `OPEN`, and not `FAIL`.

`CALL` <`CALL` ; C, n:S, G, T, R> ⇒ <compile_prolog(n) ; C, S, G, T, R>

`CUT` <`CUT` ; C, ...:{}:F, G, T, R[F]>
⇒ <C, ...:{CF:=`FAIL`}:F, G, T, R[B:=F]>

4.2.5 Condition Expression Evaluation

Equality Operations

The group of TAM operations = (unification), == (equality) and :== (pattern matching), which one may loosely describe as 'equality operations', all have a similar definition, based on that for the `UNIFY` instruction, with one or two rules removed or changed. The `UNIFY` instruction performs unification according to Robinson's algorithm (q.v. Section 2.2), but omits the computationally expensive occurs check.

`UNIFY` <`UNIFY` ; C, n_2:n_1:S, G[n_1=VAR, n_2], T, R>
⇒ <C, S, G[n_1:=n_2], (n_1,contents(n_1)):T, R> (1)

<`UNIFY` ; C, n_2:n_1:S, G[n_1, n_2=VAR], T, R>
⇒ <C, S, G[n_2:=n_1], (n_2,contents(n_2)):T, R> (2)

<`UNIFY` ; C, n_2:n_1:S, G[n_1=t,n_2=t], T, R>
⇒ <C, S, G, T, R> if $t \in$ class1 (3)

<`UNIFY` ; C, n_2:n_1:S, G[n_1=t n_{11}, n_2=t n_{21}], T, R>
⇒ <`UNIFY` ; C, n_{21}:n_{11}:S, G, T, R> if $t \in$ class2 (4)

<`UNIFY` ; C, n_2:n_1:S, G[n_1=t n_{11} n_{12}, n_2=t n_{21} n_{22}], T, R>
⇒ <`UNIFY` ; `UNIFY` ; C, n_{11}:n_{21}:n_{12}:n_{22}:S, G, T, R>
if $t \in$ class3 (5)

<`UNIFY` ; C, n_2:n_1:S, G[n_1=t i n_{12}, n_2=t i n_{22}], T, R>
⇒ <`UNIFY` ; C, n_{11}:n_{21}:n_{12}:n_{22}:S, G, T, R> if $t \in$ class8 (6)

<`UNIFY` ; C, n_2:n_1:S, G[n_1=t i, n_2=t i], T, R>
⇒ <C, S, G, T, R> if $t \in$ class10 (7)

<`UNIFY` ; C, n_2:n_1:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R> (8)

Where the classes define sets of possible node labels:

class1 = { TRUE, FALSE, []}
class2 = { NOT}

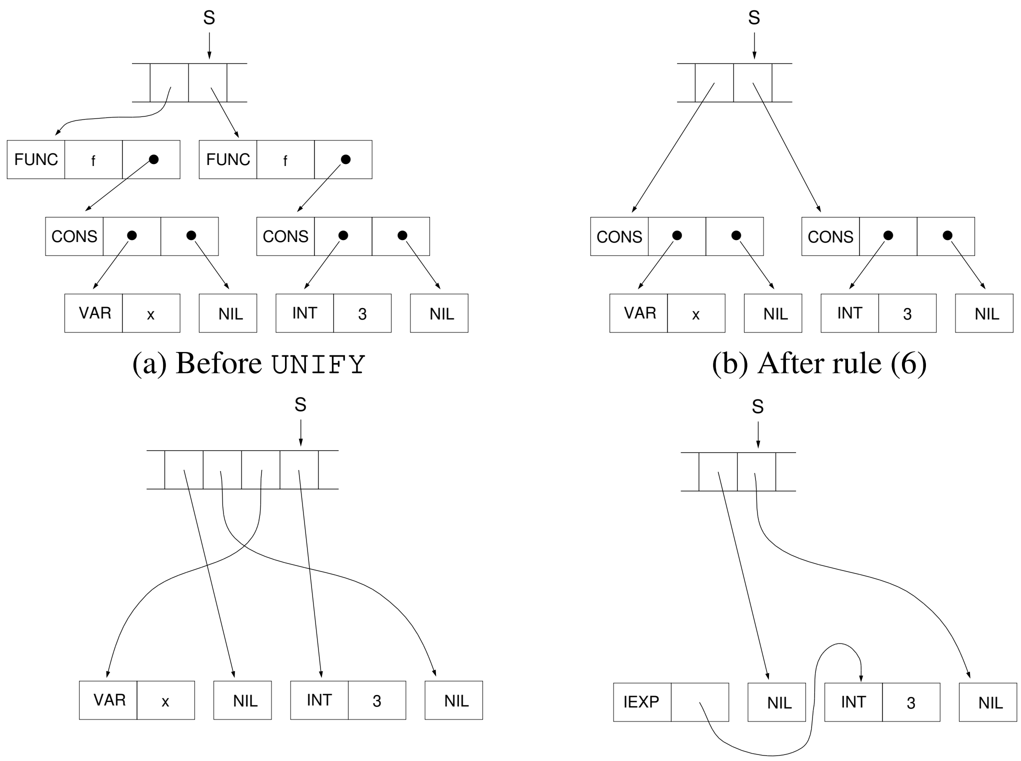
class3 = { =, ==, <, >, -g->, -t->, :- , *etc*}
class8 = { FUNC}
class10 = { INT, STR}

Example 4.5 : Evaluation of f(x)=f(3)

Before execution of the `UNIFY` instruction the graphs for f(x) and f(3) must be placed on the TOS as shown in Figure 4.4 (a). Rule (6) checks that the two have the same tag (FUNC) and functor before leaving the parameter lists of the two functions on the TOS and retrying the `UNIFY` in Figure 4.4 (b). This uses rule (5) to check that they have the same tag, and pushes both arguments of the node onto the stack in Figure 4.4 (c), ready to apply `UNIFY` twice. The result of the first `UNIFY` is shown in Figure 4.4 (d), where the variable has been indirected to the integer node, and the references to the two nodes popped from the stack. Unification will then succeed using rule (3).

Example 4.6 : Evaluation of f(x)==f(3)

Evaluation of the `EQL` instruction will proceed as for `UNIFY` in Example 4.5, except that at the stage shown in Figure 4.4 (c) rule (1) is not available for `EQL`, so the operation fails.



(a) Before `UNIFY` (b) After rule (6)

(c) After rule (5) (d) After rule (1)

Figure 4.4 : Sequence of states during `UNIFY` operation between f(x) and f(3)

For the other equality operators the following changes are necessary to the rules for UNIFY. To compare two expressions (The TAM operator ==) one simply removes the first two rules for binding variables to get the EQL instruction. To perform pattern matching (:==) one removes the second rule which gave UNIFY its bidirectional pattern matching semantics, and thus obtain the MATCH instruction.

The pattern matching process in the head of rules does not record bindings of variables on the trail (since the head variables are by definition not logical variables). To obtain the instruction PMATCH which performs this ‘non-backtrackable pattern matching’ with logical variables we must drop the second rule of UNIFY (as we did for MATCH), and modify the first to not record values on the trail:

PMATCH <PMATCH ; C, n_2:n_1:S, G[n_1=VAR, n_2], T, R>
⇒ <C, S, G[n_1:=n_2], T, R>

The UNIFY instruction serves a similar function to the various instructions in the *unify* family of the WAM, except being less optimized than the various specialized instructions that the WAM provides.

Library Predicates

A mechanism is provided for calling any function in a high level language, passing between zero (with LIB_CALL0) to five (with LIB_CALL5) root nodes of subgraphs in G, and expecting the result TRUE (in which case execution proceeds with the next T-Code instruction) or FALSE (in which case the FAIL instruction is executed). The following definition of LIB_CALL2 illustrates how these instructions expect the pointers to the parameters to be on the stack before execution, and indicate the arbitrary changes the library routine may make to the graph and trail by showing them in the consequence as G' and T'.

LIB_CALL2 *func* <LIB_CALL2 *func* ; C, n_2:n_1:S, G, T, R>
⇒ <C, S, G', T', R> if *func*(n_1,n_2)=TRUE
<LIB_CALL2 *func* ; C, n_2:n_1:S, G, T, R>
⇒ <FAIL ; ∅, S, G', T', R> if *func*(n_1,n_2)=FALSE

Whilst a general LIB_CALL *n* to call a library predicate with *n* arguments could be defined easily, its implementation would not be as efficient as the specialized instructions are for a particular number of arguments.

Arithmetic Operations

Several operators of the TAM conditional language can involve arithmetic expressions, such as the relational operators =:=, <, >, *etc*, and is operator. Such arithmetic in rule conditions could easily be performed by manipulating the graphical representation of the arithmetic expressions, but this would be inefficient compared with the more conventional technique of stack based arithmetic evaluation using integers. The T-Code thus provides a set of instructions to directly manipulate integers (as opposed to graphs containing integers) on the S stack, and to convert between representation of integers in S and G.

The S stack is used to store partial results of evaluation, and reverse Polish evaluation is used. For example, to evaluate 4–(5*x) the following steps are performed:

- push 4 onto the stack
- push the integer 5 onto the stack
- check the contents of the graph rooted at x is an integer, and push that integer onto the stack
- multiply the top two elements and replace them by the result
- add the two elements and replace them by the result

To perform the pushing of integers onto the stack we may use the `PUSH` instruction. Fetching the contents of any expression x will in general be achieved by using the usual sequence of T-Code instructions to construct x, and then applying the `INT_CONV` instruction to ensure that x is a single node containing an integer i, and replacing the TOS with i. If x is not an integer graph node then the condition has failed, and the normal action for `FAIL` should be undertaken.

`INT_CONV` <`INT_CONV` ; C, n:S, G[n=INT i], T, R> ⇒ <C, i:S, G, T, R>
<`INT_CONV` ; C, n:S, G[n≠INT i], T, R>
⇒ <`FAIL` ; ∅, S, G, T, R>

As an optimization of the procedure of finding the value of one of the arguments to the rule is an integer and then pushing its value onto the stack, the INT_COPY instruction avoids the intermediate step of constructing x on the stack by directly inspecting the argument in the current frame.

`INT_COPY` x <`INT_COPY` x ; C, n_k:...:n_x:...:n_1:{}:F, G[n_x=INT i], T, R[F]>
⇒ <C, i:n_k:...:n_x:...:n_1:{}:F, G, T, R>

<`INT_COPY` x ; C, n_k:…:n_x:…:n_1:{}:F, G[$n_x \neq$INT i], T, R[F]>

⇒ <`FAIL` ; C, n_k:…:n_x:…:n_1:{}:F, G, T, R>

Further instructions to deal with other situations (such as finding the integer value of a global variable) could easily be conceived if such situations were common, but experience has suggested that such usages are rare. The arithmetic operations are performed by instructions such as `SUB`, which assume the integer arguments are in place on TOS, and replace them with the result of the operation.

`SUB` <`SUB` ; C, i_2:i_1:S, G, T, R> ⇒ <C, $i_1 - i_2$:S, G, T, R>

The arithmetic relational predicates work by assuming that their arguments have already been placed (as integers) onto S, and cause a `FAIL` to be executed if the arithmetic relation does not hold. If the relation does hold then no action except popping the arguments from the stack need be performed. The following gives examples of how the greater than (`GRE`) and integer equality (`INT_EQL`) are defined.

`GRE` <`GRE` ; C, i_2:i_1:S, G, T, R> ⇒ <C, S, G, T, R> if $i_1 > i_2$

<`GRE` ; C, i_2:i_1:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R> if $i_1 \leq i_2$

`INT_EQL` <`INT_EQL` ; C, i_2:i_1:S, G, T, R>

⇒ <C, S, G, T, R> if $i_1 = i_2$

<`INT_EQL` ; C, i_2:i_1:S, G, T, R>

⇒ <`FAIL` ; ∅, S, G, T, R> if $i_1 \neq i_2$

The only situation where the stack representation of numbers must be converted into a graph representation is for the is and takes predicates. Thus instead of providing a general conversion instruction, two instructions are provided to evaluate these predicates in an optimized manner. For the expression *x* is *arith* we have three cases to consider:

- *x* is an integer, in which case a `FAIL` is executed if the value stored in *x* does not correspond to *arith*.
- *x* is a variable, in which case the variable is bound to a node in G representing *arith*, and the binding recorded on T.
- *x* is anything else, which indicates a type conflict (since an integer can not unify with strings, lists, functions, *etc*) and thus the `FAIL` instruction is executed.

`IS` <`IS` ; C, i:n:S, G[n=INT i], T, R> ⇒ <C, S, G, T, R>

<`IS` ; C, i:n:S, G[n=VAR], T, R>

⇒ <C, S, G[n:=IEXP n', n':=INT i],

(n,contents(n)):T, R>

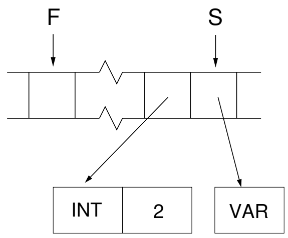
<`IS` ; C, i:n:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R>

The `TAKES` instruction only differs in not recording the binding on T, corresponding with its semantics that the variable x will not be restored to its previous value on any future backtracking.

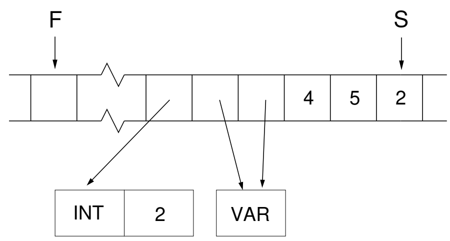
Example 4.7 : Execution of the T-Code to evaluate y is 4–(5*x)

The T-Code sequence for the evaluation is

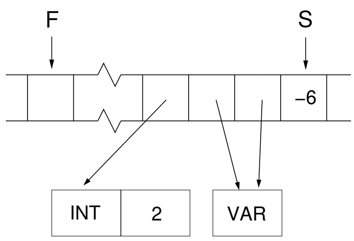
```
ARG_COPY 2
PUSH 4
PUSH 5
INT_COPY 1
MUL
SUB
IS
```



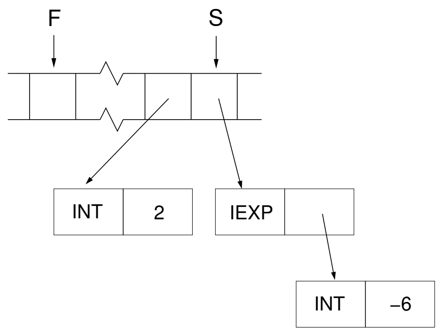
(a) Before any instruction



(b) Just before `MUL`



(c) Just before `IS`



(d) Just after `IS`

The sequence of diagrams above shows the various stages of execution of the T-Code sequence, assuming that there are no suspended procedure calls on the stack between the variable environment and the TOS. Figure (a) shows the two parameters x and y of the current rule at the top of the stack where x has been matched 2, and y with a free variable. Figure (b) shows the various numbers used in the calculation pushed onto the stack (note how the part used to evaluate the arithmetic is not stored as a graph, but placed directly on the stack). Figure (c) is after the arithmetic calculation is complete, and Figure (d) shows the result of the evaluation converted into a graph node, and assigned to the variable y.

The G-Machine provides a similar optimization [Pey87] to the normal graph rewriting instructions by providing instructions such as MKINT for `INT_CONV` and ADD* for `ADD`.

4.2.6 Manipulation of Variables

The operation of binding a variable to a graph (which may itself be another variable) is performed by the `BT_BIND` instruction if a reference to the variable is to be made on the trail T, or by `BIND` if a simple indirection without backtracking information is to be made.

`BIND`	<`BIND` ; C, n_2:\$$n_1$:S, G[$n_1$,$n_2$], T, R>
	⇒ <C, S, G[n_1:=IEXP n_2], T, R>
`BT_BIND`	<`BT_BIND` ; C, n_2:\$$n_1$:S, G[$n_1$,$n_2$], T, R>
	⇒ <C, S, G[n_1:=IEXP n_2], (n_1,contents(n_1)):T, R>

The entry on T records in a tuple the node n_1 being bound, and its contents before they are replaced by the IEXP tag and pointer. Finding the value of T for recording on the stack is done using the `LEVEL` instruction, and we use a previously recorded value of T from the stack in the `FREE` instruction to restore variables to their previous values.

`FREE`	<`FREE` ; C, t:S, G[t=INT T, n_0, ..., n_x], (n_k,p_k):...:(n_0,p_0):T, R>
	⇒ <C, S, G[contents(n_0):=p_0,...,contents(n_x):=p_x], T, R>
`LEVEL`	<`LEVEL` ; C, S, G, T, R> ⇒ <C, n:S, G[n:=INT T], T, R>

It is of interest to note that although the foregoing discussion refers to VAR nodes being the subject of this group of instructions, we might equally well apply them to any type of node. The usefulness of doing this has yet to be explored.

These instructions perform functions specific to graph rewriting of logic programming languages, and thus there are no similar instructions in the G-Machine or WAM.

Summary and Conclusions

The T-Code instruction set presented in this chapter supports the implementation of the TAM-Rewrite rule language presented in the previous chapter. An outline has been given of how TAM-Rewrite rules correspond to sequences of T-Code instructions, and an exact compilation method will be presented in the next chapter.

T-Code has been developed from G-Machine and the WAM. In particular, one may recognize that the graph construction and manipulation instructions of the G-Machine are similar to those provided in T-Code, but the facilities for handling variables and a general frame based approach are inherited from the WAM so as to provide the key features necessary for the implementation of logic programming languages. New features of the T-Code include the notion of term rewrites on graphs being distinct from graph rewrites, and the provision to handle as distinct cases the indirection function and what the indirection function points to.

The instruction set presented should be regarded as a first version of an abstract machine under development, since there are several obvious optimizations that may be applied without over complicating the situation. A few examples are:

- Making the compilation of rules generate the information for the T-Code sequence to follow on fail, since this is known at compile time. With the need to generate the rule instances at runtime removed we would not need to store the structure for the current rule in the frame header block. This would make the T-Code fall more in line with the WAM, but would make the addition of dynamic programming (the addition/removal of rewrite rules during rewriting) more complicated if such a capability were required.

- We may optimize the unification operation by providing a set of instructions dealing with special cases, as is the case in both the WAM and the T-Code's handling of pattern matching.

- The `CONS` and `VAR` instructions are obvious candidates for providing versions which perform multiple operations, such and `VAR` *n* producing *n* new variable nodes.

- The pattern matching process is very simplistic in nature. Many optimizations of pattern matching searches exist; for use in T-Code the WAM system, where the first argument is used as an index into an appropriate structure based on the first arguments of all the rules [Gab85], could be easily implemented.

These optimizations would improve performance to some extent, but which instructions need particular attention with respect to this criterion can only be ascertained by some assessment of a working version of the abstract machine used in practice. Thus the instruction set presented in this chapter was used for the first implementation of the TAM [McB92], with a view to that implementation being used for later assessment of the

T-Code performance.

5 Compilation of TAM-Rewrites to T-Code

This chapter describes the method by which a set of rules compile TAM-Rewrite rules into sequences of T-Code instructions. The description of the compilation is broken down to deal with each part of a TAM-Rewrite rule in turn, with some of the compilation rules listed. A full set of compilation rules is given in Appendix E. The compilation rules are then used, in conjunction with the state transition rules from the previous chapter, to outline the method by which we may prove that the TAM is a correct implementation of TAM-Rewrites as defined in Chapter 3. To avoid overloading of the word 'rule' we shall in this chapter sometimes refer to TAM-Rewrite rules as *rewrites* and compilation rules as *rules.*

The compilation rules are similar to those used for the G-Machine (q.v. Section 1.4.1), with the addition of a condition clause which has the same semantics as the condition clause of TAM-Rewrites. We use the nomenclature listed in Table 5.1 (which is simply a copy of Table 4.1 reproduced here for ease of reference), with all constructs of the compilation rules given in roman type.

Name	*Usage*
`INSTR`	typewriter-style typeface for T-Code instructions
rewrite	sans-serif typeface for any TAM expression or a component of the abstract machine
variable	italicized typeface for parts of the rule which match general graphs or arguments to T-Code rules. In particular ...
$n_1\ n_2$ ...	refer to any node in G
$a_1\ a_2$ ...	refer to any nodes in G which are the arguments of a function which the rule pertains to.
$i_1\ i_2$ ...	refer to any nodes in G which contain integers
$f\ \ g$	are functors
$j\ \ k$	are the arity of functions

Table 5.1 : Nomenclature of execution and compilation rules

Each instruction of a T-Code sequence produced by the compilation rules is separated by a semi-colon ';', with the empty set ∅ being used to represent a null output from the compilation. Thus the T-Code sequence `X ; ∅ ; Y` is equivalent to `X ; Y`. The address A of a instruction `I` is found by the construct `A:I`. In the head of compilation rules we may specify pattern classes, such as 'integer' and 'variable', rather like those of TAM-Rewrites (q.v. Chapter 3). The table_lookup/4 and length/2 predicates of the TAM-Rewrite condition language are also used.

Environments

As already explained in Chapter 4, the TAM uses a frame based execution mechanism, which entails storing the variables of a rule in an *environment*. When compiling rules the position that a variable will take in the environment at runtime must be known, so that the instructions generated reference the correct location. At compile time the environment will be stored in @environment as a cons list of pairs v::p, where v is the variable and p its position in the environment. With this structure we may use table_lookup/4 to find the position of a variable in the environment, and list append to add a variable to the environment. The following three predicates, defined in terms of TAM-Rewrite condition rules, provide for the searching and addition of a variable to @environment. We store the number of positions used in the runtime environment in the variable @entries, which in general will not be the same length as @environment since the TAM-Rewrite pattern conjunction operator allows two variables to reference the same object.

- To find the position of variable v in the environment, and if it is not present fail.

```
environment_lookup(v,position) :-
        table_lookup($v,position,@environment).
```

- To add a new variable v to the environment, and find its position.

```
environment_add(v,position) :-
        position is @entries+1,
        @environment:=($v::position)::environment,
        @entries := position.
```

- To search for, and if not present add a variable v to the environment, and return its position we may combine the above two definitions as detailed below. Note the use of cut/0 ensures that once a variable is found in the environment by the first rule, execution will never backtrack onto the second rule, which would cause the addition a new variable to the environment.

```
environment_lookup_and_add(v,position) :-
```

```
                    environment_lookup($v,position,@environment),
                    cut().
environment_lookup_and_add(v,position) :-
                    environment_add($v,position,@environment).
```

5.1 Compilation Rules from TAM Rewrites

The general form of a compilation rule is **S**[[*exp*]] a_1 :- *condition* → *T-Code*, where **S** specifies the name of a group of compilation rules that should be applied to a TAM-Rewrite expression *exp*. An additional argument a_1 may optionally be supplied, and the *condition* has the same semantics as a TAM-Rewrite condition, allowing manipulations of the environment to be performed before the sequence of instructions *T-Code* is produced. The compilation rules for TAM-Rewrites may be divided into five groupings as follows:

- **C** rules for compiling a term rewrite, graph rewrite or condition rule. The **C** rules use the other rule groups for the compilation of various parts of the rewrite. Associated with this group are the **V** rules for creating instructions which produce at runtime any free variables present in the rule.

- **H** and **Hd** rules for compiling the head pattern matching. These will produce an *empty output* (represented by ∅) for patterns containing only variables.

- **B** rules for the compilation of the rule body, which must produce some output since a rule body must be constructed (which at runtime may only entail copying one of the rule's arguments).

- **P** rules for compiling the condition expression of a rule. Associated with this group is the **A** rules for arithmetic expressions of the TAM-Rewrite arithmetic equality (e.g. is/2) and comparison operators. Given an empty condition the **P** rules produce an empty output.

- The **Q** rule for the compilation of a query.

The following subsections will deal with each of these groupings in turn. Before looking at these details, it is worth giving in Example 5.1 an overview of the compilation process. This revisits the TAM-Rewrite program in Example 3.2, and shows how the T-Code listed in Example 4.1 was produced from it. It is intended that the example should only be briefly glanced at before the following subsections have been read, and then returned to as a fully worked out example of the compilation of a group of rules. The convention used in the layout of example rewrite sequences, is to use vertically aligned arrows for showing each step as it effects the entire sequence of T-Code, and to indent when we wish to show just the results from applying the rules to one part of the previous step (as a kind of subcomputation).

Example 5.1 : Compilation to T-Code instructions of fib rules.

fib(1) -t-> 1. (1)

fib(2) -t-> 1. (2)

fib(n) :- n>2 -t-> !add(!fib(!sub(n,1)),!fib(!sub(n,2))). (3)

The application of the T-Code compilation system to each of the rules is itemized below.

(1) **C**[[fib(1) -t-> 1]]

→ **C**[[fib(1) :- succeed() -t-> 1]]
→ `NEXT 1` ; **H**[[1]] 1 ; **V**[[succeed()]] ; **V**[[1]] ;
  P[[succeed()]] ; **B**[[1]] ; `RETURN`

H[[1]] 1 → `ARG_COPY 1` ; **Hd**[[1]]
  → `ARG_COPY 1` ; `C_INT 1`

V[[succeed()]] → ∅

V[[1]] → ∅

P[[succeed()]] → ∅

B[[1]] → `INT 1`

→ `NEXT 1` ; `ARG_COPY 1` ; `C_INT 1` ; ∅ ; ∅ ; ∅ ;
  `INT 1` ; `RETURN`

A rule with an empty condition may be regarded as equivalent to a rule with succeed() as its condition. The head contains the pattern to match the integer 1, so the **H** rules must produce instructions which fetch the first argument and test its value. Since there are no free variables in the condition or body of the rule, the **V** rules when applied to those parts do not produce any instructions. A condition which just succeeds again needs no instructions to be generated by the **P** rules. The body is the graph containing the integer 1, so the **B** rules need only produce the single instruction which constructs that graph.

(2) **C**[[fib(2) -t-> 1]]

→ **C**[[fib(2) :- succeed() -t-> 1]]
→ `NEXT 1` ; **H**[[2]] 1 ; **V**[[succeed()]] ; **V**[[1]] ;
  P[[succeed()]] ; **B**[[1]] ; `RETURN`

→ ...

→ `NEXT 1` ; `ARG_COPY 1` ; `C_INT 2` ; ∅ ; ∅ ; ∅ ;
`INT 1` ; `RETURN`

The compilation of this rewrite has an obvious similarity with that for rewrite (1).

(3) **C**[[fib(n) :- n>2 -t-> !add(!fib(!sub(n,1)),!fib(!sub(n,2)))]]
→ `NEXT 1` ; **H**[[n]] 1 ; **V**[[n>2]] ;
V[[!add(!fib(!sub(n,1)),!fib(!sub(n,2)))]] ;
P[[n>2]] ;
B[[!fib(!sub(n,1))]] ; **B**[[!fib(!sub(n,2))]] ;
`REDIRECT 2 add`

H[[n]] 1 → ∅

V[[n>2]] → **V**[[n]] ; **V**[[2]]
→ ∅ ; ∅

V[[!add(!fib(!sub(n,1)),!fib(!sub(n,2)))]] → ... → ∅ ; ... ; ∅

P[[n>2]] → **A**[[n]] ; **A**[[2]] ; `GRE`
→ `INT_COPY 1` ; **A**[[2]] ; `GRE`
→ `INT_COPY 1` ; `PUSH 2` ; `GRE`

B[[!fib(!sub(n,1))]]
→ `PREPARE fib` ; **B**[[!sub(n,1)]] ; `ENACT 1`
B[[!sub(n,1)]] → `PREPARE sub` ; **B**[[n]] ;
B[[1]] ; `ENACT 2`
→ `PREPARE sub` ;
`ARG_COPY 1` ;
B[[1]] ; `ENACT 2`
→ `PREPARE sub` ;
`ARG_COPY 1` ;
`INT 1` ; `ENACT 2`
→ `PREPARE fib` ; `PREPARE sub` ;
`ARG_COPY 1` ;
`INT 1` ; `ENACT 2` ; `ENACT 1`

B[[!fib(!sub(n,2))]]
→ ...
→ `PREPARE fib` ; `PREPARE sub` ;
`ARG_COPY 1` ;
`INT 2` ; `ENACT 2` ; `ENACT 1`

→ `NEXT 1` ; ∅ ; ... ; ∅ ; `INT_COPY 1` ; `PUSH 2` ; `GRE` ;
`PREPARE fib` ; `PREPARE sub` ;

```
ARG_COPY 1;INT 1;ENACT 2;ENACT 1;
                    PREPARE fib;PREPARE sub;
                    ARG_COPY 1;INT 2;
                    ENACT 2;ENACT 1;REDIRECT 2 add
```

With an unrestricted pattern for this rewrite no instructions are needed to perform pattern matching, so the H rules produce no output. There are no free variables again, but there is a condition clause so the P rules produce instructions to compare the argument to the rule with the integer 2. The outermost add/2 function of the body is already catered for in the `REDIRECT` which ends the T-Code sequence for the rewrite, so we need only compile the arguments for add/2, which are themselves function activations.

5.1.1 Compilation of a TAM Rule (C[[ ]])

Term Rewrite Rules

Since rules with no condition can be considered as having just succeed() as the condition, there are only two cases to deal with when considering the compilation of term rewrite rules. The first is the optimized case where we have a body-root activation, whilst the second deals with the general case of any type of rule body. However the two cases only differ in the treatment of the body, and the instruction used to activate the rule, so we shall describe them together here.

C[[$f(a_1,a_2,...,a_k)$:- *condition* -t-> ! $g(b_1,b_2,...,b_j)$]]
:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k
→ `NEXT` n ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ;
V[[*condition*]] ; **V**[[$g(b_1,b_2,...,b_j)$]] ;
P[[*condition*]] ;
B[[b_1]] ; **B**[[b_2]] ; ... ; **B**[[b_j]] ; `REDIRECT` g , j

C[[$f(a_1,a_2,...,a_k)$:- *condition* -t-> *body*]]
:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k
→ `NEXT` n ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ;
V[[*condition*]] ; **V**[[*body*]] ; **P**[[*condition*]] ;
B[[*body*]] ; `RETURN`

The various components of the compilation rule work as follows:

- The condition of the compilation rule adds the arguments to @environment, in the positions they will be placed by a call at run time.

- The NEXT instruction is produced, which at runtime allocates sufficient space in the current frame to contain those parts of the environment which hold the arguments to the rule, and any variables contained within the head pattern. The argument to NEXT can only be calculated after the **H** compilation rules have been used.
- The **H** rules are used to produce instructions which pattern match each rule argument in a left to right ordering. Any variables not already present in the environment are added as part of the compilation. At runtime their position in the environment will have been allocated by a NEXT instruction, and filled by ARG_SET instructions. Only after having created the pattern matching instructions can we assign to *n* the number of slots needed in the runtime environment, since the **H** rules may increase this number.
- The *condition* and *body* of the rule are checked by use of the **V** rules for any free variables (i.e. those not appearing in the head), and VAR instructions added to create these variables.
- The **P** rules are used to produce instructions which check the condition.
- The body is compiled differently according to which case we are dealing with
  - the optimized case uses **B** rules to create instructions which will put each argument of root-body activation on the stack at runtime.
  - the general case uses **B** rules to put the whole body as one entry on the stack.
- The final instruction differs according to the case we are dealing with:
  - the optimized case uses REDIRECT to reuse the current frame
  - the general case uses RETURN to replace the current frame with the result of rewriting.

Graph Rewrite Rules

The rules to compile graph rewrites are identical, except that the second rule produces WRETURN instead of RETURN. At first it might be considered strange that the optimized case can be handled in the same way for both graph and term rewrite rules. This lack of any difference is due to the optimization applying only when the body is a newly created graph, and thus there can be only one pointer to it; that produced by the rewrite. As was discussed in Section 1.2.4, in this situation the effect of term and graph rewrites is identical, which would indicate that the implementation of the two rewrites should also be identical.

Condition Rules

The rules used during the evaluation of the condition clause of a rule take a similar form to that for a term rewrite rule, except that `SUCCESS` completes the evaluation of a condition rule, and there is no body to construct.

$$\begin{aligned}&\mathbf{C}[[\, f(a_1,a_2,\ldots,a_k) \text{ :- } \textit{condition.}\,]]\\ &\qquad\qquad\qquad \text{:- @environment:=}[a_1\text{::}1,a_2\text{::}1,\ldots,a_k\text{::}k]\text{, @entries:=}k\\ &\qquad\qquad\qquad \rightarrow \texttt{NEXT}\ \ n\ ;\ \mathbf{H}[[\, a_1\,]]\ ;\ \mathbf{H}[[\, a_2\,]]\ ;\ \ldots\ ;\ \mathbf{H}[[\, a_k\,]]\ ;\ \ n\text{:=@entries}\ ;\\ &\qquad\qquad\qquad\qquad\qquad \mathbf{V}[[\, \textit{condition}\,]]\ ;\ \mathbf{P}[[\, \textit{condition}\,]]\ ;\ \texttt{SUCCESS}\end{aligned}$$

Creating the Environment

So that at runtime we create all the free variables of a rule, we must at compile time check the condition and body graphs of a rule for such free variables. The **V** rules perform this task by a recursive descent over the condition and body graphs to search for variables, adding them to @environment. The rules assume that any variables belonging to the head (i.e. non-free variables) have already been inserted into @environment. Only **V** rule (5) produces an instruction, since a variable that has not already been inserted into @environment by rule (4) or the **H** rules requires a new free variable to be created at runtime. In this case, a `VAR` instruction is produced that at runtime will create in the graph a new free variable, and push onto the stack a pointer to that variable. This has the effect of adding the new free variable to the rule's runtime environment.

$$\mathbf{V}[[\, \textit{unary\_op}\ x\,]] \rightarrow \mathbf{V}[[\, x\,]] \tag{1}$$
$$\mathbf{V}[[\, x\ \textit{binary\_op}\ y\,]] \rightarrow \mathbf{V}[[\, x\,]]\ ;\ \mathbf{V}[[\, y\,]] \tag{2}$$
$$\mathbf{V}[[\, f(a_1,a_2,\ldots,a_k)\,]] \rightarrow \mathbf{V}[[\, a_1\,]]\ ;\ \mathbf{V}[[\, a_2\,]]\ ;\ \ldots\ ;\ \mathbf{V}[[\, a_k\,]] \tag{3}$$
$$\mathbf{V}[[\, v\text{:variable}\,]] \text{ :- environment\_lookup}(v,\_) \rightarrow \varnothing \tag{4}$$
$$\mathbf{V}[[\, v\text{:variable}\,]] \text{ :- environment\_add}(v,\_) \rightarrow \texttt{VAR} \tag{5}$$
$$\mathbf{V}[[\, x\,]] \rightarrow \varnothing \tag{6}$$

5.1.2 Rules for the Head Pattern Matching (H [[ ]])

As described in Chapter 4, the T-Code instructions to perform pattern matching work by using instructions to match the functor of a function, and push the arguments of the function onto the stack to be matched by other instructions. Thus the **Hd** rules for compilation of head pattern matching need to first insert the relevant instruction to match the principal functor of the rule's argument pattern, and then use further **Hd** rules to compile the matching of the pattern. However, before considering if we need to create instructions to perform pattern matching we must decide in the **H** rules if the pattern is just a variable (frequently the case in practice). Rule (1) below handles the case of a variable, which requires no pattern matching to be performed, since we will simply use the position of the argument in the environment to hold the variable's value. If the pattern is not a variable, then in rule (5) we insert an instruction to get a copy of the argument on the TOS, and then use the **Hd** rules to recursively descend over the pattern structure.

The situation is complicated by the presence of the pattern conjunction operation, which requires in rules (2–4) that any variables that form part of a pattern conjunction at the root of the argument are added to the environment as being at the position of the argument matching the pattern.

H[[*v*:variable]] *n* $\rightarrow \varnothing$ (1)

H[[*v*:variable : *x*:nonvariable]] *n*
:- @environment=(*v*::*n*)::@environment $\rightarrow$ **H**[[*x*]] (2)

H[[*x*:nonvariable : *v*:variable]] *n*
:- @environment=(*v*::*n*)::@environment $\rightarrow$ **H**[[*x*]] (3)

H[[*v*:variable : *w*:variable]] *n*
:- @environment=(*w*::*n*)::(*v*::*n*)::@environment $\rightarrow \varnothing$ (4)

H[[*x*]] *n* $\rightarrow$ `ARG_COPY` *n* ; **Hd**[[*x*]] (5)

The code for matching atoms and variables is produced by the following rules. Integers and strings are matched by specific instructions. To match the empty list, truth or falsity we need only check that the pointer is the same as that of the instance of these atoms known at compile time, since there is only ever one instance of these nodes.

Hd[[*i*:integer]] $\rightarrow$ `C_INT` *i*
Hd[[*s*:string]] $\rightarrow$ `C_STR` *s*
Hd[[[]]] $\rightarrow$ `C_IMM NIL`
Hd[[TRUE]] $\rightarrow$ `C_IMM TRUE`
Hd[[FALSE]] $\rightarrow$ `C_IMM FALSE`

Variables which are one of the arguments to a rule (and not a subterm of an argument) will have been correctly set up by the **C** rules. Variables which are a subterm of an argument will at the runtime have had a position allocated for them in the environment by `NEXT`, and now need an `ARG_SET` instruction created which will assign to that position the subgraph matched.

Hd[[*v*:variable]] :- environment_add(*v*,*pos*) $\rightarrow$ `ARG_SET` *pos*
Hd[[@*g*:variable]] $\rightarrow$ `PUSH` *g* ; `SWAP` ; `MATCH`

A pattern conjunction requires that we put a `COPY` instruction in, which at runtime duplicates the item we are attempting to match. The `COPY` should be followed with instructions to perform the two separate pattern matching operations.

Hd[[*x* : *y*]] → `COPY` ; **Hd**[[*x*]] ; **Hd**[[*y*]]

The matching of pattern classes is implemented using the same library predicates that are used in the condition checking. For instance, the *integer pattern results in a call to the library predicate `integer`.

Hd[[*integer]] → `LIB_CALL1 integer`

The descent over the pattern structure causes the instructions to be generated which check the node type and leave its arguments on the stack for checking by other instructions. Note that functions are treated rather crudely by creating a sequence of instructions to unwind the argument cons list.

Hd[[*unary_op x*]] → `C_NODE2` unary_op ; **Hd**[[*x*]]
Hd[[*x binary_op y*]] → `C_NODE3` binary_op ; **Hd**[[*x*]] ; **Hd**[[*y*]]
Hd[[$f(a_1,a_2,...,a_k)$]] → `C_NODE3 FUNC` ; `C_IMM` *f* ; **Hd**[[$(a_1,a_2,...,a_k)$]]

Example 5.2 shows how the compilation rules for head patterns are applied to produce the results already seen in Example 4.4. Note that list structures such as [x,y,...] and parameter lists [x,y,...] are merely syntactic sugar for structures of CONS nodes x::y::... . Since the cons operator :: is just an ordinary binary operator, it can be handled by general compilation rule for such operators.

Example 5.2 : Code generation for pattern matching of the rule head list_length([x],1)

The sequence of steps followed by the compilation rules for the two arguments of list_length([x],1) are listed below.

H[[[x]]] → `ARG_COPY 1` ; **Hd**[[[x]]]
→ `ARG_COPY 1` ; `C_NODE3 CONS` ; **Hd**[[x]] ; **Hd**[[[]]]
→ `ARG_COPY 1` ; `C_NODE3 CONS` ;
`ARG_SET 3` ; `C_IMM NIL`
H[[1]] → `ARG_COPY 2` ; **Hd**[[1]]
→ `ARG_COPY 2` ; `C_INT 1`

Before compilation of the head, the **C** rule will have created an environment [[x]::1,1::2]. Thus when the rule for **Hd**[[x]] executes, we call environment_add(x,*pos*) will return *pos* equal to 3 and add x to the environment to give [x::3,[x]::1,1::2]. Any further references to x in the rule thus look at the correct position in the environment as will be set at runtime by `ARG_SET`.

5.1.3 Rules for the Body (B[[ ]])

A rule body graph is constructed using instructions to create the leaf nodes (i.e. constants, variables and zero arity functions) which push pointers to them onto the stack, and then using other instructions to build the non-zero arity nodes. Thus the compiler must always produce the instructions to build the leaf nodes before the function nodes. The following rules produce the single instruction necessary to create a constant node. Note again that all instances of the empty list, truth and falsity are shared, and so we need only push the pointer to the shared instance which is known at compile time.

B[[*i*:integer]] → `INT` *i*
B[[*s*:string]] → `STR` *s*
B[[TRUE]] → `PUSH TRUE`
B[[FALSE]] → `PUSH FALSE`

Local variables need to have their position looked-up in the environment and an `ARG_DUP` or `ARG_COPY` instruction produced according to whether or not the *actual* (i.e. $) operator had been used on the variable. The location of global variables is known at compile time, so we use a `GBL_COPY` instruction to follow indirections, or if the *actual* operator is used just `PUSH` to push the pointer to the variable onto the stack.

B[[*v*:variable]] :- environment_lookup(*v*,*pos*) → `ARG_COPY` *pos*
B[[$*v*:variable]] :- environment_lookup(*v*,*pos*) → `ARG_DUP` *pos*
B[[@*v*]] → `GBL_COPY` *v*
B[[$@*v*]] → `PUSH` *v*

Lists are created by compiling the head, then the tail, and finally a `CONS` instruction to make the CONS node. A special instruction pushes a pointer onto the stack to the NIL node, which is shared like TRUE and FALSE, but has its own instruction to obtain the pointer (as opposed to using `PUSH NIL`) since it is frequently used.

B[[[]]] → `NIL`
B[[*x*::*y*]] → **B**[[*x*]] ; **B**[[*y*]] ; `CONS`

The compilation of function construction works by compiling the arguments first, and then producing the necessary instruction to create the function application.

$\mathbf{B}[[\ unary\_op\ x\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$; `NODE2` *unary_op*

$\mathbf{B}[[\ x\ binary\_op\ y\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$; $\mathbf{B}[[\ y\ ]]$; `NODE3` *binary_op*

$\mathbf{B}[[\ f(a_1,a_2,...,a_k)\ ]] \rightarrow \mathbf{B}[[\ a_1\ ]]$; $\mathbf{B}[[\ a_2\ ]]$; ... ; $\mathbf{B}[[\ a_k\ ]]$; `APPLY` f, k

The rewriting of a graph initiated by the activation operator ! in general is achieved by constructing the graph, leaving a pointer on top of the stack, then using the `EVAL` instruction to begin the attempt at rewriting. As discussed in Chapter 4, an optimization of this process is made when the activation operator is applied to a FUNC node, in which case we enclose the instructions to create the parameters of the function in a pair of instructions `PREPARE` and `ENACT`. The 'un-activated' operator # of course requires no instructions except those to create the graph for the expression it is applied to.

$\mathbf{B}[[\ !f(a_1,a_2,...,a_k)\ ]]$

$\rightarrow$ `PREPARE` f ; $\mathbf{B}[[\ a_1\ ]]$; $\mathbf{B}[[\ a_2\ ]]$; ... ; $\mathbf{B}[[\ a_k\ ]]$; `ENACT` k

$\mathbf{B}[[\ !\ x\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$; `EVAL`

$\mathbf{B}[[\ \#\ x\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$

5.1.4 Rules for the Condition (P [[ ]])

The code to execute rewrite rule conditions is produced after all variables of the rule have been created by the **C** and **V** rules. Thus the **P** rules assume a complete environment is present, and need only generate instructions to evaluate each literal of the condition in turn. Note that we expect each literal of a condition to leave the stack as it was before it executed, except calls to condition rules and disjunctions which may leave a choicepoint on the stack.

Connectives

The first rule deals with the case where we permit the condition of a rule to be empty. A conjunction is compiled by producing the code for the left conjunct first and then code for the right conjunct. Note that no special instruction is needed to 'execute' the conjunction; the linear sequencing of instructions does this for us. A disjunction requires that we produce an `OR` instruction with the address L_1 of the right disjunct before compiling the left disjunct. The left disjunct must be immediately followed by a `GOTO` to jump to the address L_2 immediately following the code sequence for the right disjunct.

$\mathbf{P}[[\ x\ ,\ y\ ]] \rightarrow \mathbf{P}[[\ x\ ]]$; $\mathbf{P}[[\ y\ ]]$

$\mathbf{P}[[\ x\ ;\ y\ ]] \rightarrow$ `OR` L_1 ; $\mathbf{P}[[\ x\ ]]$; `GOTO` L_2 ; $L_1{:}\mathbf{P}[[\ y\ ]]$; $L_2{:}\varnothing$

$\mathbf{P}[[\ \backslash+\ x\ ]] \rightarrow$ `OR` L_1 ; $\mathbf{P}[[\ x\ ]]$; `DROP_BTP` ; `FAIL` ; $L_1{:}\varnothing$

Predicates

Most built-in predicates are implemented using the `LIB_CALL` type of instruction, which requires the parameters of the predicate to be already in place on the stack. The rule for dealing with predicates of arity two is given below.

> **P**[[$f(a_1,a_2)$]] :- library_predicate($f(a_1,a_2)$) → **B**[[a_1]] ; **B**[[a_2]] ; `LIB_CALL2` *f*

The meta-level and variable control predicates must be handled in a special way. All of the fail/0, cut/0, call/1 and return_to_level/1 predicates translate directly into single instructions. The semantics of succeed/0 are such that we need produce no instructions, since implicitly the execution of an empty sequence of T-Code instructions (between two other instructions) 'succeeds' exactly once each time it is passed. For find_level/1 we use the `LEVEL` instruction along with `UNIFY`.

> **P**[[fail()]] → `FAIL`
> **P**[[succeed()]] → ∅
> **P**[[cut()]] → `CUT`
> **P**[[call(x)]] → **B**[[x]] ; `CALL`
> **P**[[find_level(x)]] → **B**[[x]] ; `LEVEL` ; `UNIFY`
> **P**[[return_to_level(x)]] → **B**[[x]] ; `FREE`

Condition rules (i.e. user defined predicates) are treated in the same way as activated body functions; enclosed in a pair of `PREPARE` and `ENACT` instructions are all the instructions to push the parameters of the predicate onto the stack.

> **P**[[$f(a_1,a_2,\ldots,a_k)$]] → `PREPARE` ; **B**[[a_1]] ; **B**[[a_2]] ; ... ; **B**[[a_k]] ; `ENACT` *k*

Due to their frequent use, the 'equality' operators are implemented directly by instructions, and not by a call to library routines. Note how we also automatically put the *actual* operator in front of a variable on the lefthand-side of an assignment, since it is unlikely that we would ever want to have the assignment overwrite the current binding of a variable instead of the actual variable!

> **P**[[*x compare_op y*]] :-
> table_lookup(*compare_op*,*op*,
> ['==':: `EQL`, '\==':: `NEQ`, '=':: `UNIFY`, ':==':: `MATCH`])
> → **B**[[x]] ; **B**[[y]] ; *op*

P[[*x*:variable *assign_op y*]] :-
table_lookup(*assign_op*,*op*,[':='::`BIND`, '::='::`BBIND`])
→ **B**[[$ *x*]] ; **B**[[*y*]] ; *op*

Arithmetic

Although the is/2, takes/2 and the arithmetic comparison operators are just another type of predicate, the way that we need to compile the arguments is quite different due to the optimized way in which arithmetic is handled. Any argument of these operators which is an arithmetic expression is compiled using **A** rules. The instructions so produced will manipulate integers directly on the stack, as opposed to the alternative of manipulating pointers to integer nodes in the graph. Thus in rule (8) where we come to compile a graph activation, we compile it as normal using the **B** rules, and then add an `INT_CONV` instruction to check that the result is a pointer to an integer in the graph, and replace that pointer with the actual integer on the stack. Integers are compiled in rule (6) to `PUSH` instructions, which push the integer directly onto the stack.

P[[*x* is *y*]] → **B**[[*x*]] ; **A**[[*y*]] ; `IS` (1)
P[[*x* takes *y*]] → **B**[[*x*]] ; **A**[[*y*]] ; `TAKES` (2)
P[[*x a_compare_op y*]] :-
table_lookup(*a_compare_op*,*op*,['=:='::`EQLN`, '=\='::`NEQN`,
'<'::`LES`, '=<'::`LEQ`, '>'::`GRE`, '>='::`GEQ`])
→ **A**[[*x*]] ; **A**[[*y*]] ; *op* (3)
A[[*v*:variable]] :- environment_lookup(*v*,*pos*) → `COPY_INT` *pos* (4)
A[[@*v*:variable]] → `GBL_COPY` *v* ; `INT_CONV` (5)
A[[*i*:integer]] → `PUSH` *i* (6)
A[[*x arith_op y*]] :-
table_lookup(*arith_op*,*op*,['+'::`ADD`,'–'::`SUB`,'/'::`DIV`,'*'::`MUL`])
→ **A**[[*x*]] ; **A**[[*y*]] ; *op* (7)
A[[! *x*]] → **B**[[*x*]] ; `INT_CONV` (8)

5.1.5 Queries (Q [[ ]])

A query q may be viewed as far as its execution is concerned to be rather like a rewrite rule * ---> q where * matches any graph and thus requires no pattern matching instructions. Thus the compilation of a query may be seen to be similar to compiling a body of a rewrite rule. However, we need to end it with a special instruction `END` which does not expect a calling frame to be available, but instead returns the result to the calling program. All the variables that are in the query graph are free, so we must set the environment to be empty before using **V** rules to create instructions that will produce all the variables at runtime.

Q[[x]] :- @environment=[], @entries=0 → **V**[[x]] ; **B**[[x]] ; `END`

5.2 Correctness of Compilation

This section describes an outline of a proof that the compilation of TAM-Rewrite rules into sequences of T-Code instructions results in a correct implementation of the TAM-Rewrite language specification of term and graph rewriting. This will involve showing that if we compile both a function activation and the rewrite rule for the function into T-Code instructions, and then execute those T-Code instructions, we obtain the same result as is dictated by the semantics of the TAM-Rewrite language.

Since the semantics of the TAM-Rewrite language and T-Code instructions are quiet complex, only some key aspects of the implementation will be dealt with in the proof sketch, but the techniques used could be extended to produce a complete proof if necessary. The proof method is the similar to that used for a simple procedural language in [Nie92], which should be consulted for a fuller explanation of proof methods that can be used for demonstrating the correctness of implementation for programming languages.

The proof sketch will involve comparing the semantics of the TAM-Rewrite language statements with that of the execution of compiled statements. In Chapter 3 no formal semantics of the TAM-Rewrite language was given, but for our purposes here we may say that there are two possible *configurations* of a graph G during rewriting:

- $< G_! , G >$ where $G_!$ is the list of activation nodes of G ordered by having the leftmost innermost activation first.
- G when there are no activations remaining.

Rewriting causes a series of configurations to be generated, the sequence being followed depending on the type of rewrite rule (if any) that matches the leftmost activation f obtained from matching $f::G'_!$ against $G_!$. The activation nodes $g_!$ of the body g may then be added to the remainder $G'_!$ by $g_!::G'_!$. (This introduces a slight confusion between single and sets of activation nodes which will be ignored).

f -g-> g: $< f::G'_! , G[f] > \Rightarrow < g_!::G'_! , G[f:=\text{IEXP}\ g, g] >$ (1)

f -t-> g: $< f::G'_! , G[h(\ldots,f,\ldots)] > \Rightarrow < g_!::G'_! , G[h(\ldots,g,\ldots), f, g] >$ (2)

no match: $< f::G'_! , G > \Rightarrow < G'_! , G >$ (3)

no activations: $< \varnothing , G > \Rightarrow G$ (4)

Rule (1) performs graph rewriting, by updating the graph with a copy of the body of the rule, and then indirecting from the body root to the contractum root. The new configuration has the new leftmost innermost node f taken from the set of activated nodes $G_!$, and the activation nodes $g_!$ put in its place. The term rewrite rule (2) works in a similar manner to the graph rewrite case, but only updates the context of the activation. Rule (3) strips a node of its activation if no rewrite rule matches, and rule (4) serves to terminate rewriting when no activation nodes remain.

The asking of a query ! $f(a_1,a_2,...,a_k)$ gives the overall transition

$$<\{ f(a_1,a_2,...,a_k)\} , G[f(a_1,a_2,...,a_k)] > \Rightarrow^* G[g]$$

where g is the normal form obtained by the use of leftmost innermost activations. This corresponds to the implementation performing the transition

$$< \mathbf{Q}[[\ ! f(a_1,a_2,...,a_k) \ ]], \{\}\text{:S, G, T, R[F=S,B=S]}> \Rightarrow^* <\varnothing, g, \text{G}[g], \text{T, R}>$$

That is to say, given the compiled version of the query as the T-Code to execute, and the machine in its initial configuration, the result is the normal form of the query created in the graph, with the TOS pointing to the root of the normal form.

Before giving a sketch of a proof that this equivalence between semantics of the TAM-Rewrites and T-Code execution of compiled TAM-Rewrites does exist, Example 5.3 describes how a specific case of graph construction can be shown to be correctly implemented. This will serve to motivate the generalized proof of correctness that follows.

Example 5.3 : The graph f(1,2) is correctly constructed by the implementation

Assuming that this graph is part of a query or body of a rewrite, the **B** compilation rules will be used, resulting in the sequence:

B[[f(1,2)]]
→ **B**[[1]] ; **B**[[2]] ; `APPLY` f , `2`
→ `INT 1` ; `INT 2` ; `APPLY` f , `2`

When executed, this gives us the sequence:

< `INT 1` ; `INT 2` ; `APPLY` f , `2`, S, G, T, R>
⇒ < `INT 2` ; `APPLY` f , `2`, n_1:S, G[n_1=INT 1], T, R>
⇒ < `APPLY` f , `2`, n_2:n_1:S, G[n_1=INT 1, n_2=INT 2], T, R>

$\Rightarrow$ < $\varnothing$, n:S, G[n=FUNC f a_1, a_1=CONS n_1 a_2, a_2=CONS n_2 a_3, a_3=NIL,
n_1=INT 1, n_2=INT 2], T, R>

i.e. the graph contains f(1,2) and the TOS has a pointer to f/2. We may write the overall transition as an execution of the call to the **B** compilation rules:

< **B**[[f(1,2)]], S, G, T, R> $\Rightarrow$* < $\varnothing$, n:S, G[n=f(1,2)], T, R>

In the example, it was shown that putting a call to a compilation scheme as the T-Code sequence to follow during execution, can be proved to produce the correct result. To formalize this process we need to introduce some equivalence rules between the execution of the calls to the compilation rules and execution of T-Code instructions. Hypothesis 5.1 is the obvious statement that if a TAM-Rewrite expression g compiles to some sequence *instr* of T-Code instructions, and then the state transition performed by the appropriate compilation scheme **Comp**[[g]] is the same as that for *instr*. Hypothesis 5.2 states that any sequence of T-Code instructions which executes in k steps may be decomposed into a pair of transitions of k_1 and k_2 steps, where $k=k_1+k_2$ and share an intermediate state in the execution sequence.

Hypothesis 5.1 : Compilation may be used instead of T-Code instructions

$$\frac{<\mathit{instr}\mathbf{C}\ ,\ \mathrm{S, G, T, R}> \Rightarrow^* <\varnothing\ ,\ \mathrm{S', G', T', R'}>}{<\mathbf{Comp}[[\ g\ ]]\ ,\ \mathrm{S, G, T, R}> \Rightarrow^* <\varnothing\ ,\ \mathrm{S', G', T', R'}>}\quad \mathit{if}\ \mathbf{Comp}[[\ g\ ]] \rightarrow \mathit{instr}\mathbf{C}$$

where *instr***C** is a sequence of T-Code instructions or calls to the compilation scheme, and **Comp** is a compilation scheme.

Hypothesis 5.2 : Sequences of configurations always may be decomposed

$$\frac{\begin{array}{ll} <\mathit{instr}_1\ ,\ \mathrm{S, G, T, R}> & <\mathrm{C}\ ,\ \mathrm{S', G', T', R'}> \\ \quad\Rightarrow^{k_1} <\varnothing\ ,\ \mathrm{S', G', T', R'}> & \quad\Rightarrow^{k_2} <\varnothing\ ,\ \mathrm{S'', G'', T'', R''}> \end{array}}{<\mathit{instr}_1\ ;\ \mathrm{C}\ ,\ \mathrm{S, G, T, R}> \Rightarrow^{k} <\varnothing\ ,\ \mathrm{S'', G'', T'', R''}>}$$

where $instr_1$ is a T-Code instruction, C is a sequence of T-Code instructions, and the number of steps k for the execution of $instr_1$; C is the sum k_1+k_2 of the two decompositions.

With Hypotheses 5.1 and 5.2 we are now in a position to show that a query and set of rule definitions in the TAM-Rewrite language are implemented correctly if they:

- have been compiled using the compilation rules (q.v. Appendix E), and

- executed using the state transition rules (q.v. Appendix D).

Proof Sketch 5.3 shows that TAM-Rewrite queries are implemented correctly if the **B** and **V** compilation schemes result in a correct implementation. A proof sketch for **B** rules is given in Proof Sketch 5.4.

Proof Sketch 5.3 : TAM graph rewrites are correctly implemented by compiled T-Code

Using the **Q** rule, asking a query $!\,f(a_1,a_2,...,a_k)$ relies on $\mathbf{V}[[\ !\,f(a_1,a_2,...,a_k)\ ]]$ and $\mathbf{B}[[\ !\,f(a_1,a_2,...,a_k)\ ]]$ generating the correct result. The sequence of operations is as follows:

$<\mathbf{Q}[[\ !\,f(a_1,a_2,...,a_k)\ ]], \{\}\text{:S, G, T, R[F=S,B=S]}>$

$\Rightarrow <\mathbf{V}[[\ !\,f(a_1,a_2,...,a_k)\ ]]\ ;\ \mathbf{B}[[\ !\,f(a_1,a_2,...,a_k)\ ]]\ ;\ \texttt{END}, \{\}\text{:S, G, T, R}>$ (1)

$\Rightarrow <\mathbf{B}[[\ !\,f(a_1,a_2,...,a_k)\ ]]\ ;\ \texttt{END},\ a_n{:}...{:}a_1{:}\{\}\text{:S, G, T, R}>$ (2)

$\Rightarrow <\texttt{END},\ g{:}a_n{:}...{:}a_1{:}\{\}\text{:S, G}[g]\text{, T, R}>$ (3)

$\Rightarrow <\emptyset,\ g,\ \text{G}[g]\text{, T, R}>$ (4)

(1) *Using* $\mathbf{Q}[[\ x\ ]]$:- @environment=[], @entries=0 $\rightarrow \mathbf{V}[[\ x\ ]]\ ;\ \mathbf{B}[[\ x\ ]]\ ;\ \texttt{END}$

(2) **V** *rules update the environment and create free variables. In this case they will create the n free variables that appear in* $f(a_1,a_2,...,a_k)$*. Proof of correctness omitted.*

(3) *Using Proof Sketch 5.4 that* **B** *rules implement graph construction correctly.*

(4) *Using* $<\texttt{END}\ ;\ \emptyset,\ n\text{:S, G, T, R}> \Rightarrow <\emptyset,\ n\text{, G, T, R}>$

Proof Sketch 5.4 will demonstrate that any graph g with activation markers is correctly constructed by the **B** rules. The proof sketch will show that if in an initial state we have the compiled code $\mathbf{B}[[\ g\ ]]$ for the graph as the T-Code sequence to follow, then after the execution sequence is completed the graph g will have been added to G, and a pointer to that graph placed on the TOS.

Proof Sketch 5.4 : TAM graph rewrite bodies are correctly constructed by compiled rules

We expect the compilation of a body graph to result at runtime in an instance of the graph being placed on the stack S. Thus we need to prove that

$<\mathbf{B}[[\ g\ ]]\ ;\ \text{C},\ a_k{:}...{:}a_1{:}\{\}\text{:F, G, T, R[F]}> \Rightarrow^* <\text{C},\ n{:}...{:}a_k{:}...{:}a_1{:}\{\}\text{:F, G}[n{=}g]\text{, T, R}>$

The proof sketch will work inductively on the shape of the execution of calls to the **B** compilation rules. This is done by first proving the correct implementation for those compilations which build the leaf nodes of graphs, and then proving constructor and function applications inductively. Note that except for the manipulation of the stack and trail, each part of the proof leaves the execution state unchanged, and thus Hypothesis 5.2 is easily applied where necessary.

Leaf Nodes

Integers

<**B**[[*i*:integer]] ; C, S, G, T, R>
⇒ <`INT` *i* ; C, S, G, T, R> (1)
⇒ <C, *n*:S, G[*n*=INT *i*], T, R> (2)

(1) *Using* **B**[[*i*:integer]] → `INT` *i and Hypothesis 5.1*
(2) *Using* <`INT` *i* ; C, S, G, T, R> ⇒ <C, *n*:S, G[*n*=INT *i*], T, R>

Variables

<**B**[[*v*:variable]] ; C, S, G, T, R>
⇒ <`ARG_COPY` *i* ; C, ...:n_k:...:n_i:...:n_1:{}:F, G, T, R[F]> (1)
⇒ <C, n_i:...:n_k:...:n_i:...:n_1:{}:F, G, T, R> (2)

(1) *Using* **B**[[*v*:variable]] :- environment_lookup(*v*,*pos*) → `ARG_COPY` *pos and Hypothesis 5.1*
(2) *Using* <`ARG_COPY` *i* ; C, n_k:...:n_i:...:n_1:{}:F, G, T, R[F]>
⇒ <C, n_i:n_k:...:n_i:...:n_1:{}:F, G, T, R>

Constructor Nodes

Unary Operators

<**B**[[*unary_op x*]] ; C, S, G, T, R>
⇒ <**B**[[*x*]] ; `NODE2` *unary_op* ; C, S, G, T, R> (1)
⇒* <`NODE2` *unary_op* ; C, n_x:S, G[n_x=*x*], T, R> (2)
⇒ <C, *n*:S, G[*n*=*unary_op* n_x], T, R> (3)

(1) *Using* **B**[[*unary_op x*]] → **B**[[*x*]] ; `NODE2` *unary_op and Hypothesis 5.1*
(2) *By induction on the structure of the execution of the* **B** *compilation*
(3) *Using* <`NODE2` *t* ; C, n_1:S, G, T, R> ⇒ <C, *n*:S, G[*n*=*t* n_1], T, R>

Binary Operators

<**B**[[*x binary_op y*]] ; C, S, G, T, R>
⇒ <**B**[[*x*]] ; **B**[[*y*]] ; `NODE3` *binary_op* ; C, S, G, T, R> (1)

⇒* <`NODE3` *binary_op* ; C, n_y:n_x:S, G[n_x=x,n_y=y], T, R> (2)

⇒ <C, n:S, G[n=*binary_op* n_x n_y], T, R> (3)

(1) *Using* **B**[[*x binary_op y*]] → **B**[[*x*]] ; **B**[[*y*]] ; `NODE3` *binary_op*
and Hypothesis 5.1
(2) *By induction on the structure of the execution of the two* **B** *compilations and Hypothesis 5.2*
(3) *Using* <`NODE3` *t* ; C, n_2:n_1:S, G, T, R> ⇒ <C, n:S, G[n=t n_1 n_2], T, R>

Function Nodes

Function Application

<**B**[[$f(a_1,a_2,...,a_k)$]] ; C, S, G, T, R>

⇒ <**B**[[a_1]] ; **B**[[a_2]] ; ... ; **B**[[a_k]] ; `APPLY` *f*, *k* ; C, S, G, T, R> (1)

⇒* <`APPLY` *f*, *k* ; C, a_n:...:a_2:a_1:S, G, T, R> (2)

⇒ <C, n:S, G[n=FUNC f p_1, p_1=CONS a_1 p_2, ... , p_k= CONS a_k p_{k+1}, p_{k+1}=NIL], T, R> (3)

(1) *Using* **B**[[$f(a_1,a_2,...,a_k)$]] → **B**[[a_1]] ; **B**[[a_2]] ; ... ; **B**[[a_k]] ; `APPLY` *f*, *k*
and Hypothesis 5.1
(2) *By induction on the structure of the execution of the* **B** *compilations and Hypothesis 5.2*
(3) *Using* <`APPLY` *f*, *k* ; C, n_k:n_{k-1}:...:n_1:S, G, T, R>
⇒ <C, n:S, G[n=FUNC f p_1, p_1=CONS n_1 p_2, ... ,
p_k= CONS n_k p_{k+1}, p_{k+1}=NIL], T, R>

Function Activation: Non optimized

<**B**[[! *g*]] ; C, S, G, T, R>

⇒ <**B**[[*g*]] ; `EVAL` ; C, S, G, T, R> (1)

⇒* <`EVAL` ; C, n_g:S, G[n_g=g], T, R[F,B]> (2)

EITHER

⇒ <1st(code_for(f)), a_k:...:a_1:{CF:=1st(code_for(f)),FF:=S,BP:=B,VB:=T, CS:=C,FS:=F,RL:=code_for(f),CL:=n}:S, G, T, R[F:=S,B:=S]> *if* n_g=$f(a_1,a_2,...,a_k)$ (3a)

⇒ <`NEXT` *n* ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ; **V**[[*condition*]] ; **V**[[*body*]] ; **P**[[*condition*]] ; **B**[[*body*]] ; `RETURN`, a_k:...:a_1:{}:F, G, T, R[F]> (4a)

⇒* < **B**[[*body*]] ; `RETURN`, a_k:...:a_1:{}:F, G, T', R[F]> (5a)

⇒* < `RETURN`, n_{body}:a_k:...:a_1:{BP,CS,FS}:F, G[n_{body}=*body*], T', R> (6a)

$\Rightarrow$ < CS, n_{body}:S, G, VB, R[F:=FS,B:=BP]> (7a)

OR

$\Rightarrow$ < C, n_g:S, G, T, R> *if* $n_g \neq f(a_1,a_2,...,a_k)$ (3b)

(1) *Using* **B**[[! *x*]] $\rightarrow$ **B**[[*x*]] ; `EVAL` *and Hypothesis 5.1*

(2) *By induction on the structure of the execution of the* **B** *compilation*

(3a) *Using* <`EVAL` ; C, n:S, G[n=FUNC f n_1, n_1=CONS a_1 n_2,...,
n_k=CONS a_k n_{k+1}, n_{k+1}=NIL], T, R[F,B]>
$\Rightarrow$ <1st(code_for(f)),
a_k:...:a_1:{CF:=1st(code_for(f),FF:=S,BP:=B,VB:=T,
CS:=C,FS:=F,RL:=code_for(f),CL:=n}:S,
G,T,R[F:=S,B:=S]>

(4a) *The T-Code for the rewrite rule* $f(a_1,a_2,...,a_k)$ *:- condition ---> body or the* `OPEN` *instruction will be returned by the call* 1st(code_for(f)). *This will have been generated by the* **C** *compilation rules — the rule for a term rewrite with no body-root activation is chosen here, the proof for the other rules would take a similar form.*

Using **C**[[$f(a_1,a_2,...,a_k)$:- *condition* -t-> *body*]]
:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k
$\rightarrow$ `NEXT` n ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ;
V[[*condition*]] ; **V**[[*body*]] ; **P**[[*condition*]] ;
B[[*body*]] ; `RETURN`

(5a) *The proof of correctness for the* **H**, **V** *and* **P** *rules is omitted. They serve to perform pattern matching, create new logical variables, and condition evaluation. Thus after their evaluation the environment (in general) will have additional entries.*

(6a) *By induction on the structure of the execution of the* **B** *compilation*

(7a) *Using* <`RETURN` ; C, n:...:{BP,CS,FS}:F, G, T, R[F]>
$\Rightarrow$ <CS, n:F, G, T, R[F:=FS,B:=BP]>

Note that only the context of the activation is updated, since we only place a pointer to the contractum root on the stack, and do not update the node pointed to by the CL *entry in the frame with an indirection to the contractum root.*

(3b) *Using* <`EVAL` ; C, n:S, G[$n \neq$FUNC f n'], T, R> $\Rightarrow$ <C, n:S, G, T, R>

Function Activation: Non body-root optimization

<**B**[[! $f(a_1,a_2,...,a_k)$]] ; C, S, G, T, R>

$\Rightarrow$ <`PREPARE` f ; **B**[[a_1]] ; **B**[[a_2]] ; ... ; **B**[[a_k]] ; `ENACT` k ; C, S, G, T, R> (1)

⇒ <**B**[[a_1]] ; **B**[[a_2]] ; ... ; **B**[[a_k]] ; `ENACT` n ; C,
{CF:=1st(code_for(f),FF:=S,BP:=B,VB:=T,
FS:=F,RL:=code_for(f),CL:=∅}:S, G, T, R> (2)

⇒* <`ENACT` k ; C, a_k:...:a_2:a_1:∅:{}:S, G, T, R> (3)

⇒ <1st(code_for(f)), a_k:...:a_2:a_1:∅:{CS:=C}:S, G, T, R> (4)

⇒* <C, n_{body}:S, G, T', R> (5)

(1) *Using* **B**[[!$f(a_1,a_2,...,a_k)$]] →
`PREPARE` f ; **B**[[a_1]] ; **B**[[a_2]] ; ... ; **B**[[a_k]] ; `ENACT` k
and Hypothesis 5.1

(2) *Using* <`PREPARE` f ; C, S, G, T, R[B,F]>
⇒ <C, {CF:=1st(code_for(f),FF:=S,BP:=B,VB:=T,
FS:=F,RL:=code_for(f),CL:=∅}:S, G, T, R[B:=S]>

(3) *By induction on the structure of the execution of each of the* **B** *compilations and Hypothesis 5.2*

(4) *Using* <`ENACT` k ; C, a_k:...:a_1:{CF}:B, G, T, R[B]>
⇒ <CF, a_k:...:a_1:{CS:=C}:S, G, T, R[F:=S,B:=S]>

(5) *As from step* (3a) *of non-optimized case*

Proof Sketch 5.4 does not demonstrate that function activation is applied in a leftmost innermost ordering, but that this is so should be readily apparent from the fact that the **B** compilation rules always place the instructions to build the leftmost innermost part of the graph first.

Summary and Conclusions

In this chapter a set of compilation rules for the TAM-Rewrite language has been presented which converts rewrites into sequences of T-Code instructions. Also, an outline has been given of how the compilation rules, taken in conjunction with the state transition rules for T-Code, may be proved to produce a semantically correct implementation of TAM-Rewrites. The compilation rules have been implemented as a ‘C’ program, and forms part of the TAM-Rewrite language implementation [McB92].

The compilation algorithm should be regarded as a ‘first’ implementation, in which attention has been focused on producing correct T-Code programs, and not on the optimization of the code produced. There are thus numerous simple optimizations which may be considered for inclusion in a future implementation, some of which we might mention here are:

- The handling of free variable creation is rudimentary. Firstly, the `VAR` instruction could be extended to handle creating several variables at once (as opposed to having a series of instructions each creating a single variable). Secondly, a VAR instruction only needs to be placed just before the literal of a condition that uses it. This would generate more efficient code, since such variables would not be created if an earlier literal failed. Thirdly, as with the WAM, some variables need no representation in the graph. If the first use of a free variable is in the left hand side of an indirection then it only needs a position in the stack environment to point at the indirection's right hand side.

- The optimization of body-root activations in rewrite rules has a parallel in condition rules, where we can use the well known technique of *tail recursion* to optimize the last rule for a functor. When the final rule for a functor ends with a call to another rule, then the frame could be reused as the frame for the predicate being called.

- With the current compilation scheme, any information on the stack present when a rule commits is kept there. Normally it is just the variables, but can include choice points left over from the condition evaluation. This is all wasted space, and the automatic insertion of a type of CUT instruction which reclaimed this space would improve memory usage efficiency.

As was remarked in the conclusions for Chapter 4, which particular aspect of the TAM that needs attention with respect to improving performance can only be fully assessed after some practical experience has been gained in using the TAM as an implementation vehicle for logic programming. As it stands, the compilation scheme produces code which corresponds to the current definition of the T-Code instruction set, and can be shown to be a correct implementation of term and graph rewriting.

Summary & Conclusions

This thesis has presented the basis for making practical implementations of logic programming in graph rewrite systems. In particular, the following topics have been covered:

- The use of GRS as defined in [Bar87a] for the implementation of logic programming was studied, and shown to lead to some problems when executing logical disjunctions. Some enhancements to GRS to provide *unification*, *backtracking* of variable bindings and the notion of a *term rewrite on graphs* have been introduced to overcome these problems. Although unification in GRS has been studied before [Gla88b,Pap89], the new method of explicitly treating the logical variable as a fixed location in the graph allows for the use of backtracking on variable bindings. Thus the notion of indirection functions being used to implement rewriting becomes significant for the correctness of implementation. The unification algorithm used by a built-in unify operator has been proved to be a correct implementation of the well-known Robinson's algorithm.

  The notion of a term rewrite on graphs was in effect provided by the Dactl0 language [Gla87], but at the time it was not apparent what use the operation would be, and the feature was dropped from latter versions of the language [Gla88a,Gla91].

- The *TAM-Rewrite* compiler target language has been described, being based on GRS with the enhancements detailed in the previous item. Having some similarities with Dactl [Gla88a], it includes full control over the binding, trailing and backtracking of logical variables, and the choice for every rewrite being a term or graph rewrite. Although it allows for the specification of which functions should be considered for rewriting using activation markings, it does not provide such elaborate control over the process as there is in Dactl. This is due to the TAM-Rewrite language being aimed at a fast sequential machine implementation, whilst Dactl is aimed at providing the necessary control for efficient parallel evaluation.

- An abstract machine called *TAM* has been defined, to support the efficient implementation of TAM-Rewrites on a sequential machine architecture. The TAM's T-Code instruction set was defined using state transition rules like those used for the G-Machine [Joh84,Pey87]. A set of compilation rules between TAM-Rewrites and T-Code was provided, and an outline proof was given that the compilation rules and state transition rules together provide a correct implementation of TAM-Rewrites.

The TAM, and the compilation from TAM-Rewrites to T-Code, has been implemented in 'C' [McB92], and used for the implementation of Prolog and the PLL in TAM-Rewrites that has been presented within this thesis. We may use some benchmark figures from the implementation of Prolog to gain a little insight to the benefits (or otherwise) of using graph rewriting as opposed to term rewriting. The comparison between term and graph rewriting may be done by timing the execution times for some of the simple programs listed in Appendix B, and then re-timing them with all the graph rewrite rules for user defined predicates replaced by term rewrite rules. The implementation of the user defined rules then becomes essentially a tree rewriting one (but the use of variables and the way in which we handle alternative goals is still done by graph rewriting).

	Time in s for -g->	*Time in s for* -t->	*Time in s for 'C' PLL*
fib(10,x)	0.57	0.57	0.35
permute([1,2,3,4,5],x)	0.33	0.41	0.74
reverse([1,2,...,20],x)	0.51	0.51	0.55

The third column of the table above gives timings for version 0.43 of the first 'C' implementation of the PLL, and indicates that the use of the TAM-Rewrite language has not incurred any overheads compared with a direct implementation of rewriting in 'C'. The first and second columns give timings for version 0.16 of the TAM-Rewrite system. The table indicates that the fib/2 and reverse/2 predicates do not execute any faster when implemented as a graph rewrite as opposed to a term rewrite. This is due the the deterministic nature of the program — only one clause of a disjunction ever succeeds, and so the computation of the clause that succeeds has no other clauses which can share results via graph rewriting. Looking at the Prolog version of the *permute* program, repeated below from Appendix B, we can see that the call to *permute* in the second clause will succeed with several alternative solutions, calling *insert* with each. When using graph rewriting, after the first call to permute/2, the insert/2 will already have replaced by its body, and thus we only need repeat the evaluation of its body for the alternative solutions to permute/2. Thus we achieve a benefit from the use of graph rewriting, which is reflected in the timings.

permute([],[]).

permute([xh|xt],y):-

permute(xt,pxt),

insert(xh,pxt,y).

The implementation [M^cB92] only interprets T-Code instructions, as opposed to compiling them into a native machine code of some machine architecture. Thus it is not yet possible to make a direct comparison between PLL and Prolog implementations in the TAM and those made using more conventional techniques. However, it is apparent that we can not expect an implementation in TAM-Rewrites to provide the same performance as WAM-type methods as used in Prolog, due to two factors:

- The TAM-Rewrite implementation will have an overhead of creating graphs to store the current program state, such as the sequencing of disjunctions.

- Since the arguments of functions must be created in the graph before it is rewritten, there will often be a wasted effort in producing graphs for conjunctions, since once a term has rewritten to FALSE the result of the conjunction is also FALSE. For example, if we are rewriting and(a,and(b,c)) and a fails, then we have produced the graph and(b,c) unnecessarily. By contrast the graph creation that is needed in the WAM (for variables and structures) is conducted only when the predicate that uses that graph is executed.

Set against these problems of efficiency, the use of graph rewriting makes a language's implementation specification much nearer to being its operational semantics. This has the advantages that the correct implementation of the language is easier to verify, and that the development of new language implementations will be much more rapid.

Future development of the work presented in this thesis will follow several themes:

- The TAM-Rewrite language provides the necessary constructs to support constraint logic programming. An implementation of one such language, for example CHIP [Hen89], would test these capabilities in practice. As part of this work, some study would need to be conducted into the correctness of using the trail and backtracking facilities of the TAM-Rewrite language on non-variable nodes. Although these operations already exist in the language, the impact that their use has will need consideration if we are to maintain the soundness of approach to the implementation of logic programming.

- The implementation of the TAM in 'C' [McB92] should be extended to generate native machine code, as opposed to the current practice of interpreting T-Code instructions. In addition to the obvious benefit of improving the performance of TAM-Rewrite programs, this would allow for a comparison of performance between implementations of logic programming in TAM-Rewrites and more standard implementations.

- Further development of the control mechanisms used in the TAM-Rewrite language might be investigated, to provide methods by which we can avoid unnecessary graph construction during rewriting. This may use as a starting point Dactl's notion of rewriting a function when only some of its arguments have been rewritten, and modify it to create arguments only if the function can not be rewritten without their values.

- Profiling of the T-Code programs for the implementations of the PLL and Prolog presented in this thesis would allow for an assessment to be made on which T-Code instructions need attention with respect to increase in execution efficiency. Results can be imported from the development in the G-Machine [Pey87], and new abstract machines which support functional programming with logic programming extensions, such as the machine proposed by Lock [Loc90].

In this thesis it has been shown how we may adapt existing techniques from logic programming to produce a modified notion of graph rewriting more suited to the implementation of logic programming. With the above mentioned future developments in mind, it can be seen that the work set out in this thesis makes a sound basis for the further exploration of how we can implement logic programming languages by graph rewriting.

A TAM-Rewrite Language

TAM Syntax

The following modified BNF expressions give the syntactic definition of a TAM expression. Expressions in brackets '{}' are optional, and if followed by an asterisk '*' may be repeated any number of times.

Rules

```
<rule>         ::= <head> :- <condition> -t-> <body>.
<rule>         ::= <head> :- <condition> -g-> <body>.
<rule>         ::= <head> :- <condition>.
<head>         ::= <function>
<condition>    ::= <condition> , <condition> | <condition> ; <condition> |
                   \+ <condition> | <graph> <c_binary_op> <graph> |
                   <graph> is <arith> |
                   <graph> takes <arith> | <function>
<arith>        ::= <integer> | <graph> | <arith> <a_binary_op> <arith>
<a_binary_op>  ::= + | – | * | /
<c_binary_op>  ::= =.. | = | := | :== | ::= | =:= | =\= | == | \== | > | < | =< | >= |
                   @> | @< | @=< | @>=
<body>         ::= <graph>
```

Graphs

```
<graph>        ::= <terminal> | <function> | <list> | <condition> |
                   <graph> <binary_op> <graph> |
                   <unary_op> <graph>
<function>     ::= <identifier>({ <graph> })
<list>         ::= [ { <graph> } ]
<unary_op>     ::= \+ | ! | # | $
<binary_op>    ::= :- | ---> | -t-> | -g-> | :: | : | is | takes | , | ; |
                   <c_binary_op> | <a_binary_op>
```

Terminals

<terminal> ::= <truth_value> | <integer> | <string> |
<identifier> | <pattern_class>

<truth_value> ::= TRUE | FALSE

<integer> ::= { + | – } digit { digit }*

<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

<string> ::= " {<*any ASCII character*>}* "

<identifier> ::= <alpha_char> { <alpha_char> | <digit> | _ }*

<alpha_char> ::= <*any alphabet character*>

<pattern_class> ::= *integer | *number | *list | *var | *nonvar | *atom

Precedence of TAM Operators

The following table lists the TAM operators in decreasing order of precedence (i.e. those at the top of the table bind their operands tighter than those lower down the table). The *level* entry in the table at present serves only to give a numeric reference to the precedence. In the future new predicates may be defined which would manipulate the precedence of operators, as well as define new operators, in a similar manner to that allowed in Prolog.

<table>
<tr><th>Level</th><th>Operator</th><th>Associativity</th></tr>
<tr><td>50</td><td>! $ #</td><td>right to left</td></tr>
<tr><td>400</td><td>+ –</td><td>left to right</td></tr>
<tr><td>500</td><td>* /</td><td>left to right</td></tr>
<tr><td>700</td><td>=:= =\= == \== > < =< >= @> @< @=< @>=</td><td>left to right</td></tr>
<tr><td>700</td><td>=.. = := :== ::= is takes</td><td>right to left</td></tr>
<tr><td>900</td><td>\+ ::</td><td>right to left</td></tr>
<tr><td>950</td><td>: |</td><td>right to left</td></tr>
<tr><td>1000</td><td>,</td><td>right to left</td></tr>
<tr><td>1050</td><td>-></td><td>right to left</td></tr>
<tr><td>1100</td><td>;</td><td>right to left</td></tr>
<tr><td>1200</td><td>:-</td><td>right to left</td></tr>
<tr><td>1400</td><td>---> -t-> -g-></td><td>right to left</td></tr>
<tr><td>2000</td><td>.</td><td>left to right</td></tr>
</table>

Semantics of Condition Expressions

A TAM-Rewrite rule condition is made up of a sequence of literals formed of the followed connectives:

A , B	Succeeds if A succeeds, and then B succeeds.
A ; B	Succeeds if either A succeeds or B succeeds.
\+ A	Succeeds if A does not succeed.

A literal may be any one of the following built-in predicates:

A=B	Unify the graph A with the graph B
A==B	Graphs A and B are identical, ignoring any indirection nodes contained within either graph. `(x+1)==(x+1)` `p(1)==!p(1)   %  succeeds if p(1) is in normal form`
A\==B	Graphs A and B are not identical.
A@<B	Graph A precedes B in the canonical ordering. Canonical ordering is defined as the alphanumeric ordering of the ASCII string of each graph as produced by write/1.
A@>B	Graph A follows B in the canonical ordering.
A@=<B	Graph A precedes or is identical to graph B in the canonical ordering.
A@>=B	Graph A follows or is identical to graph B in the canonical ordering.
A:=B	Without following any indirection nodes from A, overwrite the root node of A with an indirection node pointing to B.
A::=B	As for :=/2, except record previous values stored in the root of A on the trail for future backtracking.
A:==B	Match graph B against the pattern graph A. Pattern matching is a restricted kind of unification, where variables in B may not match non-variables in A.
A is B	Unify the graph A with the graph representing the evaluation of arithmetic expression B, which may be made up of numbers, arithmetic operators and graphs which rewrite to numbers before is/2 is evaluated. Any node activation which does not result in a number as its normal form will result in the is predicate failing at run-time. If A was a variable, then it is placed on the trail, so that this operator is reversed by backtracking. `x is 2+3+7`

	cubic is x*!square(x)
A takes B	As for is/2, except A is not placed on the trail.
A=:=B	A and B are both numbers, and A is equal to B.
A=\=B	A and B are both numbers, and A is not equal to B.
A<B	A and B are both numbers, and A is less than B.
A>B	A and B are both numbers, and A is greater than B.
A=<B	A and B are both numbers, and A is less than or equal to B.
A>=B	A and B are both numbers, and A is greater than or equal to B.
A=..L	Unify graph A with the list L, where the first element of L is the root label of A, and successive elements are the parameters of A if it is a FUNC node, or the successors of the root constructor if not a FUNC node.
	f(x,y) =.. ["f",x,y] x+y =.. ["+",x,y]
atom(A)	A is a string node.
atomic(A)	A is a string or numeric node.
call(E)	Add expression E as part of the condition expression at the place where the call statement occurs.
compound(G)	Graph G is a compound expression, namely a function, constructor operator or CONS node.
copy_graph(G,C)	Produce in C a version of graph G which contains no IEXP nodes. This will ensure that the graph in C will not change its value on any backtracking operation (such as a call to return_to_level/1), since the IEXP nodes usually represent bound variables.
cut()	Remove all further instances of rules from the pattern matching process for the function currently being considered.
fail()	Backtrack immediately.
find_level(L)	Records in L a value which represents the current state of the logical variables in the graph.
functor(G,N,A)	The name of function G is unified with N, and the arity with A. Binary operators are treated as functions of arity two, and unary operators as arity one.
	functor(f(1,2,3),X,Y) ---> X="f",Y=3 functor(5+A,X,Y) ---> X="+",Y=2
integer(I)	I is an integer.

length(L,N)	List L has length N. On calling, either L or N must be ground.
list(L)	L has a CONS or NIL node as its root.
make_free_var(V)	Make node V a free variable, by overwriting the label of the root of V with the name VAR. Invariably V should be prefixed with the $ operator to ensure that the actual node of a bound variable is passed, and not the value it is bound to. x=2,make_free_var($ x) ---> TRUE % x left free
nl()	Force a new line on the current output device.
nonvar(X)	X is not a free variable.
number(N)	N is a numeric node (equivalent to integer/1 in any integer implementation).
printf(S,A_1,A_2,...)	S is a string which when printed, will substitute the *n*th occurrence of '%' by A_n. The control characters (such as \n for newline) supported by the 'C' language may be used. The number of A arguments supplied may be between 0 and 4 inclusive. printf("Welcome to %\nTime is now % o'clock\n","Peter",12) ---> Welcome to Peter Time is now 12 o'clock
return_bindings(V,F1,F2,R)	Return in structure R the bindings of the variables in list V, where each variable is paired with its binding by functor F2, and the pairings are built together using the arity two functor F1. Note the use of the $ operator in the example, so that we give the predicate the actual node in the graph representing the variable, and not what it may happen to be bound to by following an IEXP node. % X=1 and Y=2 before call return_bindings([$X,$Y],or(_,_),equal(_,_),R) ---> R=or(equal(X,1),equal(Y,2))
return_to_level(L)	Return all variables to their values which held when L was recorded using find_level/1. Only variables made using logical bind operations will be affected by this operation.
succ(A,B)	B is the successor to integer A. On calling, either A or B must be ground.
succeed()	Succeed exactly once.

table_lookup(I,V,T)
T is a list of CONS nodes, where the left of each CONS node is searched for I, and if equal, the right of that CONS node is unified with V. If unification fails on the first match found for I, or if no match is found for I, then the predicate fails.

table_lookup(2,X,[1::"red",2::"green",3::"blue"]) ---> X="green"
table_lookup(1,"bus",[1::"car",2::"train",1::"bus"]) ---> FALSE

time(T) T is unified with the current system time. Interpretation of values is implementation dependent.

var(V) V is a free variable.

write(A) Print A on the current output device.

B TAM-Rewrite Rules to Implement Prolog

Handling of Query and Results

```
#define visible_vars              % Declare global variable

prolog_query(query,var_list) :-
        @visible_vars:=var_list
        -t-> !and(query,end_of_resolution()).

end_of_resolution() :-
        display_vars(@visible_vars)
        -t-> FALSE.

display_vars([]):-
        printf("==========\n").
display_vars([x|tail]) :-
        printf("%\n",x),
        display_vars(tail).
```

SLD Resolution

```
and(x,y) -t-> !try_and(!x,y).

try_and(FALSE,x) -t-> FALSE.
try_and(TRUE,x) -t-> !x.
try_and(or(x,tail),y) :-
        find_level(var_level),
        tail:=y
        -t-> !backtrack(x,var_level).
try_and(x,y) :-
        write("failing on "),
        write(x),nl()
        -t-> FALSE.               % closed world assumption
```

```
#rewrite_head or/2              % declare or/2 as being functor we can activate
```

Backtracking

```
cut(m) :- m:=TRUE -t-> TRUE.

backtrack([],var_level) -t-> FALSE.
backtrack([mark(m)|b],var_level) :-
        nonvar(m),
        make_free_var($m)
        -t-> FALSE.
backtrack([mark(m)|b],var_level) -t-> !backtrack(b,var_level).
backtrack([a|b],var_level) :-
        res := !a,
        return_to_level(var_level)
        -t-> !make_or(res,!backtrack(b,var_level)).

make_or(FALSE,x) -t-> x.
make_or(TRUE,x) -t-> TRUE.
```

Negation By Failure

```
not(x) :-
        find_level(vars)
        -t-> !try_not(!x,vars).

try_not(TRUE,vars) :-
        return_to_level(vars)
        -t-> FALSE.
try_not(x,vars) :-
        return_to_level(vars)
        -t-> TRUE.
```

Prolog Built-In Predicates

```
eq(a,b) :- a=b -t-> TRUE.
eq(a,b) -t-> FALSE.
gt(a,b) :- a > b -t-> TRUE.
gt(a,b) -t->  FALSE.
lt(a,b) :- a < b -t-> TRUE.
```

```
lt(a,b) -t->  FALSE.
p_is(a,b) :- a is !arith(b) -t-> TRUE.
p_is(a,b) -t-> FALSE.
p_write(a) :- write(a) -t-> TRUE.
p_nl(a) :- nl() -t-> TRUE.

% Rules to perform arithmetic

arith(a+b) :- c is !arith(a)+!arith(b) -t-> c.
arith(a-b) :- c is !arith(a)-!arith(b) -t-> c.
arith(a*b) :- c is !arith(a)*!arith(b) -t-> c.
arith(a/b) :- c is !arith(a)/!arith(b) -t-> c.
arith(a) -t-> a.
```

Examples of User Defined Predicates

A selection of familiar Prolog predicates is listed here with the equivalent compiled form in TAM-Rewrites. The translation is a fairly direct one, with Prolog predicates translated to functions according to Table B.1.

Prolog	Translated Form in TAM
a,*b*	and(*a*,*b*)
\+ *a*	not(*a*)
a;*b*	or([and(*a*,tail),and(*b*,tail)],tail)
a=*b*	eq(*a*,*b*)
a<*b*	lt(*a*,*b*)
a>*b*	gt(*a*,*b*)
a is *b*	p_is(*a*,*b*)
write(*a*)	p_write(*a*)
nl	p_nl()
p(...):-*a*. *p*(...):-*b*.	*p*(...) -g-> or([and(*a*,tail),and(*b*,tail)],tail)
p(...):-a_1,!,a_2. *p*(...):-*b*.	*p*(...) -g-> or([and(a_1,and(cut(m),and(a_2,tail))),mark(m),and(*b*,tail)],tail)

Table B.1 : Abstract translation rules for converting Prolog into TAM-Rewrites

The translation of predicates and disjunctions extends in an obvious manner for the situation where we have multiple disjuncts. The rules for predicates assume that any non-variable in the head has already been replaced by a variable plus an explicit unification in the body to the original non-variable; i.e. *p*(*nonvar*) :- ... becomes *p*(*var*) :- *var*=*nonvar*, ...

Fibonacci Numbers

Prolog	TAM-Rewrite
fib(1,1). *fib(2,1).* *fib(x,y):-* *x>2,* *xp is x−1,* *xpp is x−2,* *fib(xp,yp),* *fib(xpp,ypp),* *y is yp+ypp.*	fib(x,y) -g-> or([and(eq($x,1),and(eq($y,1),tail)), and(eq($x,2),and(eq($y,1),tail)), and(gt($x,2),and(p_is(xp,$x-1), and(p_is(xpp,$x-2), and(fib(xp,yp), and(fib(xpp,ypp), and(p_is($y,yp+ypp),tail))))))], tail).

Fibonacci Numbers using Cuts

Prolog	TAM-Rewrite
cfib(1,y):-!,y=1. *cfib(2,y):-!,y=1.* *cfib(x,y):-* *xp is x−1,* *xpp is x−2,* *cfib(xp,yp),* *cfib(xpp,ypp),* *y is yp+ypp.*	cfib(x,y) -g-> or([and(eq($x,1),and(cut(m1),and(eq($y,1),tail))), mark(m1), and(eq($x,2),and(cut(m2),and(eq($y,1),tail))), mark(m2), and(p_is(xp,$x-1),and(p_is(xpp,$x-2), and(cfib(xp,yp),and(cfib(xpp,ypp), and(p_is($y,yp+ypp),tail)))))], tail).

List Concatenation

```
append([],y,z).
append([xh|xt],y,[xh|zt]):-
        append(xt,y,zt).
```

```
append(x,y,z) -g->
   or(  [ and(eq($x,[]),and(eq($y,$z),tail)),
          and(eq($x,[xh|xt]),and(eq($z,[xh|zt])
                ,and(append(xt,$y,zt),tail)))],
        tail).
```

Prolog *TAM-Rewrite*

List Reversal

```
reverse([],[]).
reverse([xh|xt],y):-
        reverse(xt,rxt),
        append(rxt,[xh],y).
```

```
reverse(x,y) -g->
   or(  [ and(eq($x,[]),and(eq($y,[]),tail)),
          and(eq($x,[xh|xt]),and(reverse(xt,rxt),
                and(append(rxt,[xh],$y),tail)))],
        tail).
```

Prolog *TAM-Rewrite*

List Permutation

```
permute([],[]).
permute([xh|xt],y):-
        permute(xt,pxt),
        insert(xh,pxt,y).
```

```
permute(x,y) -g->
   or(  [ and(eq($x,[]),and(eq($y,[]),tail)),
          and(eq($x,[xh|xt]),and(permute(xt,pxt),
                and(insert(xh,pxt,$y),tail)))],
        tail).
```

Prolog *TAM-Rewrite*

List Insertion

```
insert(x,list,[x|list]).
insert(x,[lh|lt],[lh|xlt]):-
        insert(x,lt,xlt).
```

```
insert(x,list,y) -g->
   or(  [ and(eq($y,[$x|$list]),tail),
          and(eq($list,[lh|lt]),and(insert($x,lt,xlt),
                and(eq($y,[lh|xlt]),tail)))],
        tail).
```

Prolog *TAM-Rewrite*

C TAM-Rewrite Rules to Implement the PLL

Handling of Query and Results

```
#define delay_or             % Define global variables used in rules
#define visible_vars
#define debug
#define inside_not

pll_query(f,var_list) :-
        @visible_vars:=var_list,
        @delay_or:=[],
        @inside_not:=FALSE,
        @debug:="off"
        -t-> !encapsulate(f,[]).

encapsulate(f,tail) :-
        find_level(btp),
        @delay_or:=tail,
        fprime:=!f,
        fprime:=!rewrite_delay_or(@delay_or,fprime),
        return_to_level(btp)
        -t-> fprime.
```

Disjunctions

```
or(x,y) :- @delay_or:=[x,y|@delay_or] -t-> TRUE.

rewrite_delay_or(x,FALSE) -t-> FALSE.
rewrite_delay_or([x,y|tail],TRUE)
        -t-> !lr_or(!encapsulate(x,tail),encapsulate(y,tail)).
rewrite_delay_or([x,y|tail],c)
        -t-> !lr_or(!encapsulate(and(x,c),tail),encapsulate(and(y,c),tail)).
```

```
rewrite_delay_or([],TRUE) :-
        return_bindings(@visible_vars,and(a,b),eq(c,d),vars),
        (@debug==“off”;printf(“Bindings found are %\n”,vars))
        -t-> vars.
rewrite_delay_or([],x) :-
        return_bindings(@visible_vars,and(a,b),eq(c,d),vars),
        (@debug==“off”;
           printf(“Bindings found are % with partial result %\n”,vars,x))
        -t-> !make_and(x,vars).
```

Conjunctions

```
and(x,y) -t-> !s1(!x,y,-1).
s1(TRUE,y,l) -t-> !y.
s1(FALSE,y,l) -t-> FALSE.
s1(x,y,l) :- find_level(new_level),new_level=\=l -t-> !s2(x,!y,new_level).
s1(x,y,l) -t-> and(x,y).
s2(x,TRUE,l) -t-> !x.
s2(x,FALSE,l) -t-> FALSE.
s2(x,y,l) :- find_level(new_level),new_level=\=l -t-> !s1(!x,y,new_level).
s2(x,y,l) -t-> and(x,y).
```

Simplification of Conjunctions and Disjunctions

```
lr_and(TRUE,x) -t-> !x.
lr_and(FALSE,x) -t-> FALSE.
lr_and(x,y) -t-> !make_and(x,!y).

lr_or(TRUE,x) -t-> TRUE.
lr_or(FALSE,x) -t-> !x.
lr_or(x,y) -t-> !make_or(x,!y).

make_and(TRUE,x) -t-> x.
make_and(FALSE,x) -t-> FALSE.
make_and(x,TRUE) -t-> x.
make_and(x,FALSE) -t-> FALSE.
make_and(x,y) -t-> and(x,y).
```

```
make_or(TRUE,x) -t-> TRUE.
make_or(FALSE,x) -t-> x.
make_or(x,TRUE) -t-> TRUE.
make_or(x,FALSE) -t-> x.
make_or(x,y) -t-> or(x,y).
```

Negation

```
not(TRUE) -t-> FALSE.
not(FALSE) -t-> TRUE.
not(and(a,b)) -t-> !or(not(a),not(b)).
not(not(a)) -t-> !a.
not(or(a,b)) -t-> !and(not(a),not(b)).
not(a) :-
        tmp_inside_not:=@inside_not,
        tmp_delay_or:=@delay_or,
        @inside_not:=TRUE,
        new_a:=!encapsulate(a,[]),
        @inside_not:=tmp_inside_not,
        @delay_or:=tmp_delay_or,
        \+ new_a==a
        -t-> !not(new_a).
```

Quantification

```
all(x,f) -t-> !not(some(x,not(f))).

some(x,a) :-
        @inside_not==FALSE
        -t-> !a.
some(x,or(a,b))
        -t-> !or(some(x,a),some(x,b)).
some(x,a) :-
        tmp:=@delay_or,
        new_a:=!encapsulate(a,[]),
        @delay_or:=tmp
        -t-> !make_some(x,new_a).
```

```
make_some(x,TRUE) -t-> TRUE.
make_some(x,FALSE) -t-> FALSE.
make_some(x,a) :- nonvar(x) -t-> a.
make_some(x,a) -t-> some(x,a).
```

Equality

```
eq(x,y) -t-> !make_eq(!x,!y).

make_eq(z,add(x,y)) -t-> !make_add(z,x,y).
make_eq(z,sub(x,y)) -t-> !make_sub(z,x,y).
make_eq(z,mul(x,y)) -t-> !make_mul(z,x,y).
make_eq(z,div(x,y)) -t-> !make_div(z,x,y).
make_eq(add(x,y),z) -t-> !make_add(z,x,y).
make_eq(sub(x,y),z) -t-> !make_sub(z,x,y).
make_eq(mul(x,y),z) -t-> !make_mul(z,x,y).
make_eq(div(x,y),z) -t-> !make_div(z,x,y).
make_eq(x,y) :- y=x -t-> TRUE.
make_eq(x,y) -t-> FALSE.
```

Arithmetic

```
add(x,y) -t-> !make_add(!x,!y).

make_add(x,y) :- z takes x+y -t-> z.
make_add(x,y) -t-> add(x,y).

make_add(x,y,z) :- yval is x-z -t-> !make_eq(yval,y).
make_add(x,y,z) :- zval is x-y -t-> !make_eq(zval,z).
make_add(x,y,z) -t-> eq(x,add(y,z)).

sub(x,y) -t-> !make_sub(!x,!y).
make_sub(x,y) :- z takes x-y -t-> z.
make_sub(x,y) -t-> sub(x,y).

make_sub(x,y,z) :- yval is x+z -t-> !make_eq(yval,y).
make_sub(x,y,z) :- zval is y-x -t-> !make_eq(zval,z).
make_sub(x,y,z) -t-> eq(x,sub(y,z)).
```

```
mul(x,y) -t-> !make_mul(!x,!y).
make_mul(x,y) :- z takes x*y -t-> z.
make_mul(x,y) -t-> mul(x,y).

make_mul(x,y,z) :- yval is x/z -t-> !make_eq(yval,y).
make_mul(x,y,z) :- zval is x/y -t-> !make_eq(zval,z).
make_mul(x,y,z) -t-> eq(x,mul(y,z)).

div(x,y) -t-> !make_div(!x,!y).
make_div(x,y) :- z takes x/y -t-> z.
make_div(x,y) -t-> div(x,y).

make_div(x,y,z) :- yval is x*z -t-> !make_eq(yval,y).
make_div(x,y,z) :- zval is y/x -t-> !make_eq(zval,z).
make_div(x,y,z) -t-> eq(x,div(y,z)).
```

Condition Operators

```
less(x,y) :- x<y -t-> TRUE.
less(x:*integer,y:*integer) -t-> FALSE.

greater(x,y) :- x>y -t-> TRUE.
greater(x:*integer,y:*integer) -t-> FALSE.
```

Examples of User Defined Rules

Fibonacci Numbers

define fib(x,y) guard x ∨ y tobe
x=1 ∧ y=1
∨ *x=2 ∧ y=1*
∨ *some (xp,xpp,yp,ypp) x>2 ∧ xp=x−1 ∧ xpp=x−2 and*
fib(xp,yp) ∧ fib(xpp,ypp) ∧ y=yp+ypp

PLL

```
fib(x,y) :- nonvar(x);nonvar(y) -g->
    !or( and(eq(x,1),eq(y,1)),
         or( and(eq(x,2),eq(y,1)),
              some(xp,some(xpp,some(yp,some(ypp,
                  and(greater(x,2),and(eq(xp,sub(x,1)),
                  and(eq(xpp,sub(x,2)), and(fib(xp,yp),
                  and(fib(xpp,ypp),eq(y,add(yp,ypp)))))))))))
           )
       ).
```

TAM-Rewrite

List Concatenation

define append(x,y,z) guard [x,y] $\vee$ *[z] tobe*
x=[] $\wedge$ *y=z*
$\vee$ *some (xh,xt,zt) x=xh::xt* $\wedge$ *z=xh::zt* $\wedge$ *append(xt,y,zt)*

PLL

```
append(x,y,z) :- nonvar(x),nonvar(y);nonvar(z) -g->
    !or( and(eq($x,[]),eq($y,$z)),
         some(xh,some(xt,some(zt,
                  and(eq($x,[xh|xt]),and(eq($z,[xh|zt]),
                          append(xt,$y,zt))))))).
```

TAM-Rewrite

List Reversal

define reverse(x,y,z) guard [x] $\vee$ *[z] tobe*
x=[] $\wedge$ *y=[]*
$\vee$ *some (xh,xt,rxt) x=xh::xt* $\wedge$
reverse(xt,rxt) $\wedge$ *append(rxt,[xh],y)*

PLL

```
reverse(x,y) :- nonvar(x);nonvar(y) -g->
    !or( and(eq($x,[]),eq($y,[])),
         some(xh,some(xt,some(rxt,
             and(eq($x,[xh|xt]),
                  and(reverse(xt,rxt),append(rxt,[xh],$y))))))).
```

TAM-Rewrite

List Permutation

define permute(x,y,z) guard [x] ∨ [z] tobe
x=[] ∧ y=[]
∨ some (xh,xt,pxt) x=xh::xt ∧
permute(xt,pxt) ∧ insert(xh,pxt,y)

PLL

```
permute(x,y) :- nonvar(x);nonvar(y) -g->
    !or(  and(eq($x,[]),eq($y,[])),
          some(xh,some(xt,some(pxt,
                  and(eq($x,[xh|xt]),and(permute(xt,pxt),
                          insert(xh,pxt,$y))))))).
```

TAM-Rewrite

List Insertion

define insert(x,list,y) guard [x,list] ∨ [y] tobe
y=x::list
∨ some (lh,lt,xlt) list=lh::lt ∧ insert(x,lt,xlt) ∧ y=lh::xlt

PLL

```
insert(x,list,y) :- nonvar(x),nonvar(list);nonvar(y) -g->
   !or(   eq($y,[$x|$list]),
          some(lh,some(lt,some(xlt,
                  and(eq($list,[lh|lt]),and(insert($x,lt,xlt),
                          eq($y,[lh|xlt]))))))).
```

TAM-Rewrite

D State Transition Rules for T-Code Instructions

Rewrite Evaluation

`EVAL` <`EVAL` ; C, n:S, G[n=FUNC $f\ n_1$, n_1=CONS $a_1\ n_2$,...,
n_k=CONS $a_k\ n_{k+1}$, n_{k+1}=NIL], T, R[F,B]>
$\Rightarrow$ <1st(code_for(f)),
a_k:...:a_1:{CF:=1st(code_for(f),FF:=S,BP:=B,VB:=T,
CS:=C,FS:=F,RL:=code_for(f),CL:=n}:S,
G,T,R[F:=S,B:=S]>

<`EVAL` ; C, n:S, G[n≠FUNC $f\ n'$], T, R> $\Rightarrow$ <C, n:S, G, T, R>

`PREPARE` f <`PREPARE` f ; C, S, G, T, R[B,F]>
$\Rightarrow$ <C, {CF:=1st(code_for(f),FF:=S,BP:=B,VB:=T,
FS:=F,RL:=code_for(f),CL:=$\emptyset$}:S, G, T, R[B:=S]>

`ENACT` k <`ENACT` k ; C, a_k:...:a_1:{CF}:B, G, T, R[B]>
$\Rightarrow$ <CF, a_k:...:a_1:{CS:=C}:S, G, T, R[F:=S,B:=S]>

`REDIRECT` f, k <`REDIRECT` f, k ; C,
a_{x+k}:...:a_{x+1}:a_x:...:a_1:{ }:F,G,T,R[F,B]>
$\Rightarrow$ <1st(code_for(f)),
a_k:...:a_1:{CF:=1st(code_for(f),VB:=T,
RL:=code_for(f),CL:=$\emptyset$}:F, G, T, R[B:=F]>

`RETURN` <`RETURN` ; C, n:...:{BP,CS,FS}:F, G, T, R[F]>
$\Rightarrow$ <CS, n:F, G, T, R[F:=FS,B:=BP]>

`WRETURN` <`WRETURN` ; C, n:...:{BP,CS,FS,CL=h}:F, G, T, R[F]>
$\Rightarrow$ <CS, n:F, G[h:=IEXP n], T, R[F:=FS,B:=BP]>

<`WRETURN` ; C, n:...:{BP,CS,FS,CL=$\emptyset$}:F, G, T, R[F]>
$\Rightarrow$ <CS, n:F, G, T, R[F:=FS,B:=BP]>

`END` <`END` ; $\emptyset$, n:S, G, T, R> $\Rightarrow$ <$\emptyset$, n, G, T, R>

`OPEN` <`OPEN` ; $\emptyset$(, a_k:...:a_1:{CL=n}:F, G[n≠$\emptyset$], T, R[F]>
$\Rightarrow$ <`RETURN` ; $\emptyset$, n:{CL=n}:F, G, T, R>

<`OPEN` ; $\emptyset$, a_k:...:a_1:{CL=$\emptyset$}:F, G, T, R[F]>
$\Rightarrow$ <`RETURN` ; $\emptyset$, n:{CL=$\emptyset$}:F,
G[n=FUNC $f\ n_1$, n_1=CONS $a_1\ n_2$,...,
n_k=CONS $a_k\ n_{k+1}$, n_{k+1}=NIL], T, R>

`NEXT` *i*	<`NEXT` *i* ; C, n_k:....:n_i:....:n_1:{RL}:F, G, T, R[F]> ⇒ <C, n_i:....:n_1:{CF:=1st(RL),RL:=next(RL)}:F, G, T, R>

Head Patterns

`C_NODE2` *t*	<`C_NODE2` *t* ; C, *n*:S, G[n=t n_1], T, R> ⇒ <C, n_1:S, G, T, R>
	<`C_NODE2` *t* ; C, *n*:S, G[n≠t n_1], T, R> ⇒ <`FAIL` ; C, S, G, T, R>
`C_NODE3` *t*	<`C_NODE3` *t* ; C, *n*:S, G[n=t n_1 n_2], T, R> ⇒ <C, n_1:n_2:S, G, T, R>
	<`C_NODE3` *t* ; C, *n*:S, G[n≠t n_1 n_2], T, R> ⇒ <`FAIL` ; C, S, G, T, R>
`C_INT` *i*	<`C_INT` *i* ; C, *n*:S, G[n=INT i], T, R> ⇒ <C, S, G, T, R>
	<`C_INT` *i* ; C, *n*:S, G[n≠INT i], T, R> ⇒ <`FAIL` ; C, S, G, T, R>
`C_STR` *s*	<`C_STR` *s* ; C, *n*:S, G[n=STR s], T, R> ⇒ <C, S, G, T, R>
	<`C_STR` *s* ; C, *n*:S, G[n≠STR s], T, R> ⇒ <`FAIL` ; C, S, G, T, R>
`C_IMM` *n*	<`C_IMM` *n* ; C, *n*:S, G[*n*], T, R> ⇒ <C, S, G, T, R>
	<`C_IMM` *n* ; C, n_g:S, G[n≠n_g], T, R> ⇒ <`FAIL` ; C, S, G, T, R>

Stack Manipulation

`PUSH` *n*	<`PUSH` *n* ; C, S, G[\$*n*], T, R> ⇒ <C, *n*:S, G, T, R>
`COPY`	<`COPY` ; C, \$*n*:S, G, T, R> ⇒ <C, *n*:*n*:S, G, T, R>
`SWAP`	<`SWAP` ; C, n_1:n_2:S, G, T, R> ⇒ <C, n_2:n_1:S, G, T, R>
`ARG_SET` *i*	<`ARG_SET` *i* ; C, \$$n_k$:$n_{k-1}$:....:$n_i$:....:$n_1$:{}:F, G, T, R[F]> ⇒ <C, n_{k-1}:....:n_k:....:n_1:{}:F, G, T, R>
`ARG_COPY` *i*	<`ARG_COPY` *i* ; C, n_k:....:n_i:....:n_1:{}:F, G, T, R[F]> ⇒ <C, n_i:n_k:....:n_i:....:n_1:{}:F, G, T, R>
`ARG_DUP` *i*	<`ARG_DUP` *i* ; C, n_k:....:\$$n_i$:....:$n_1$:{}:F, G, T, R[F]> ⇒ <C, n_i:n_k:....:n_i:....:n_1:{}:F, G, T, R>
`GBL_COPY` *n*	<`GBL_COPY` *n* ; C, n_k:....:n_1:{}:F, G[*n*], T, R[F]> ⇒ <C, *n*:n_k:....:n_1:{}:F, G, T, R>

Graph Construction

`INT` *i*	<`INT` *i* ; C, S, G, T, R> ⇒ <C, *n*:S, G[n=INT i], T, R>
`STR` *s*	<`STR` *s* ; C, S, G, T, R> ⇒ <C, *n*:S, G[n=STR s], T, R>
`PUSH` *n*	<`PUSH` *n* ; C, S, G[*n*], T, R> ⇒ <C, \$*n*:S, G, T, R>
`VAR`	<`VAR` ; C, S, G, T, R> ⇒ <C, *n*:S, G[n=VAR], T, R>
`NODE2` *t*	<`NODE2` *t* ; C, n_1:S, G, T, R> ⇒ <C, *n*:S, G[n=t n_1], T, R>
`NODE3` *t*	<`NODE3` *t* ; C, n_2:n_1:S, G, T, R> ⇒ <C, *n*:S, G[n=t n_1 n_2], T, R>

`APPLY` f, k <`APPLY` f, k ; C, n_k:n_{k-1}:...:n_1:S, G, T, R>
⇒ <C, n:S, G[n=FUNC $f\ p_1$, p_1=CONS $n_1\ p_2$, ... ,
p_k= CONS $n_k\ p_{k+1}$, p_{k+1}=NIL], T, R>

`CONS` <`CONS` ; C, n_2:n_1:S, G, T, R> ⇒ <C, n:S, G[n=CONS $n_1\ n_2$], T, R>

`NIL` <`NIL` ; C, S, G, T, R> ⇒ <C, n:S, G[n=NIL], T, R>

Conditional Expression Control

`OR C'` <`OR C'` ; C, S, G, T, R[F,B]>
⇒ <C, {CF:=C',FF:=F,BP:=B,VB:=T}:S,
G, T, R[B:=S]>

`DROP_BTP` <`DROP_BTP` ; C, ...:{FF,BP}:B, G, T, R[F≠FF,B]>
⇒ <`DROP_BTP` ; C, B, G, T, R[B:=BP]>
<`DROP_BTP` ; C, ...:{FF,BP}:B, G, T, R[F=FF,B]>
⇒ <C, B, G, T, R[B:=BP]>

`GOTO` C' <`GOTO` C' ; C, S, G, T, R> ⇒ <C', S, G, T, R>

`FAIL` <`FAIL` ; C, ...:{CF=`FAIL`,FF,BP}:B, G, T, R[B]>
⇒ <`FAIL` ; C, B, G, T, R[B:=BP,F:=FF]>
<`FAIL` ; C, ...:{CF,FF,VB}:B, G, (v_1,p_1):...:(v_n,p_n):VB, R[F,B]>
⇒ <CF, ...:{}:B, G[v_1:=p_1,...,v_n:=p_n], VB, R[F:=FF]>

`SUCCEED` <`SUCCEED` ; C, ...:{CS,FS}:F, G, T, R[F]>
⇒ <CS, ...:{}:F, G, T, R[F:=FS]>

`CALL` <`CALL` ; C, n:S, G, T, R> ⇒ <compile_prolog(n) ; C, S, G, T, R>

`CUT` <`CUT` ; C, ...:{}:F, G, T, R[F]>
⇒ <C, ...:{CF:=`FAIL`}:F, G, T, R[B:=F]>

Conditional Expression Evaluation

`UNIFY` <`UNIFY` ; C, n_2:n_1:S, G[n_1=VAR, n_2], T, R>
⇒ <C, S, G[n_1:=n_2], (n_1,contents(n_1)):T, R> (1)
<`UNIFY` ; C, n_2:n_1:S, G[n_1, n_2=VAR], T, R>
⇒ <C, S, G[n_2:=n_1], (n_2,contents(n_2)):T, R> (2)
<`UNIFY` ; C, n_2:n_1:S, G[n_1=t,n_2=t], T, R>
⇒ <C, S, G, T, R> if t∈class1 (3)
<`UNIFY` ; C, n_2:n_1:S, G[n_1=t n_{11}, n_2=t n_{21}], T, R>
⇒ <`UNIFY` ; C, n_{21}:n_{11}:S, G, T, R> if t∈class2 (4)
<`UNIFY` ; C, n_2:n_1:S, G[n_1=$t\ n_{11}\ n_{12}$, n_2=$t\ n_{21}\ n_{22}$], T, R>
⇒ <`UNIFY` ; `UNIFY` ; C, n_{11}:n_{21}:n_{12}:n_{22}:S, G, T, R>
if t∈class3 (5)

<`UNIFY` ; C, n_2:n_1:S, G[n_1=t i n_{12}, n_2=t i n_{22}], T, R>

⇒ <`UNIFY` ; C, n_{11}:n_{21}:n_{12}:n_{22}:S, G, T, R> if $t \in$ class8 (6)

<`UNIFY` ; C, n_2:n_1:S, G[n_1=t i, n_2=t i], T, R>

⇒ <C, S, G, T, R> if $t \in$ class10 (7)

<`UNIFY` ; C, n_2:n_1:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R> (8)

Where: class1 = { TRUE, FALSE, []}

class2 = { NOT}

class3 = { =, ==, <, >, -g->, -t->, :- , *etc*}

class8 = { FUNC}

class10 = { INT, STR}

`EQL` *as for* `UNIFY` *but excluding the rules* (1) *and* (2)

`MATCH` *as for* `UNIFY` *but excluding rule* (2)

`PMATCH` *as for* `UNIFY` *but excluding rule* (2), *and modifying rule* (1) *to*

<`PMATCH` ; C, n_2:n_1:S, G[n_1=VAR, n_2], T, R>

⇒ <C, S, G[n_1:=n_2], T, R>

`LIB_CALL0` *func* <`LIB_CALL0` *func* ; C, S, G, T, R>

⇒ <C, S, G', T', R> if *func*()=TRUE

<`LIB_CALL0` *func*: C, S, G, T, R>

⇒ <`FAIL` ; ∅, S, G', T', R> if *func*()=FALSE

`LIB_CALL1` *func* <`LIB_CALL1` *func* ; C, n_1:S, G, T, R>

⇒ <C, S, G', T', R> if *func*(n_1)=TRUE

<`LIB_CALL1` *func*: C, S, G, T, R>

⇒ <`FAIL` ; ∅, S, G', T', R> if *func*(n_1)=FALSE

`LIB_CALL2` *func* <`LIB_CALL2` *func* ; C, n_2:n_1:S, G, T, R>

⇒ <C, S, G', T', R> if *func*(n_1,n_2)=TRUE

<`LIB_CALL2` *func* ; C, n_2:n_1:S, G, T, R>

⇒ <`FAIL` ; ∅, S, G', T', R> if *func*(n_1,n_2)=FALSE

`LIB_CALL3` *func* <`LIB_CALL3` *func* ; C, n_3:n_2:n_1:S, G, T, R>

⇒ <C, S, G', T', R> if *func*(n_1,n_2,n_3)=TRUE

<`LIB_CALL3` *func*: C, S, G, T, R>

⇒ <`FAIL` ; ∅, S, G', T', R> if *func*(n_1,n_2,n_3)=FALSE

`LIB_CALL4` *func* <`LIB_CALL4` *func* ; C, n_4:n_3:n_2:n_1:S, G, T, R>

⇒ <C, S, G', T', R> if *func*(n_1,n_2,n_3,n_4)=TRUE

<`LIB_CALL4` *func*: C, S, G, T, R>

⇒ <`FAIL` ; ∅, S, G', T', R> if *func*(n_1,n_2,n_3,n_4)=FALSE

`LIB_CALL5` *func* <`LIB_CALL5` *func* ; C, n_5:n_4:n_3:n_2:n_1:S, G, T, R>

⇒ <C, S, G', T', R> if *func*(n_1,n_2,n_3,n_4,n_5)=TRUE

<`LIB_CALL5` *func*: C, S, G, T, R>
⇒ <`FAIL` ; ∅, S, G', T', R> if $func(n_1,n_2,n_3,n_4,n_5)$=FALSE

Arithmetic Operations

`INT_COPY` *x*	<`INT_COPY` *x* ; C, n_k:...:n_x:...:n_1:{}:F, G[n_x=INT *i*], T, R[F]>
	⇒ <C, *i*:n_k:...:n_x:...:n_1:{}:F, G, T, R>
	<`INT_COPY` *x* ; C, n_k:...:n_x:...:n_1:{}:F, G[n_x≠INT *i*], T, R[F]>
	⇒ <`FAIL` ; C, n_k:...:n_x:...:n_1:{}:F, G, T, R>
`IS`	<`IS` ; C, *i*:*n*:S, G[*n*=INT *i*], T, R> ⇒ <C, S, G, T, R>
	<`IS` ; C, *i*:*n*:S, G[*n*=VAR], T, R>
	⇒ <C, S, G[*n*:=IEXP *n′*, *n′*:=INT *i*], (*n*,contents(*n*)):T, R>
	<`IS` ; C, *i*:*n*:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R>
`TAKES`	<`TAKES` ; C, *i*:*n*:S, G[*n*=INT *i*], T, R> ⇒ <C, S, G, T, R>
	<`TAKES` ; C, *i*:*n*:S, G[*n*=VAR], T, R>
	⇒ <C, S, G[*n*:=IEXP *n′*, *n′*:=INT *i*], T, R>
	<`TAKES` ; C, *i*:*n*:S, G, T, R> ⇒ <`FAIL`:[], S, G, T, R>
`INT_CONV`	<`INT_CONV` ; C, *n*:S, G[*n*=INT *i*], T, R> ⇒ <C, *i*:S, G, T, R>
	<`INT_CONV` ; C, *n*:S, G[*n*≠INT *i*], T, R>
	⇒ <`FAIL` ; ∅, S, G, T, R>
`ADD`	<`ADD` ; C, i_2:i_1:S, G, T, R> ⇒ <C, i_1+i_2:S, G, T, R>
`SUB`	<`SUB` ; C, i_2:i_1:S, G, T, R> ⇒ <C, i_1-i_2:S, G, T, R>
`MUL`	<`MUL` ; C, i_2:i_1:S, G, T, R> ⇒ <C, $i_1\times i_2$:S, G, T, R>
`DIV`	<`DIV` ; C, i_2:i_1:S, G, T, R> ⇒ <C, $i_1\div i_2$:S, G, T, R>
`LEQ`	<`LEQ` ; C, i_2:i_1:S, G, T, R> ⇒ <C, S, G, T, R> if $i_1\le i_2$
	<`LEQ` ; C, i_2:i_1:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R> if $i_1>i_2$
`LES`	<`LES` ; C, i_2:i_1:S, G, T, R> ⇒ <C, S, G, T, R> if $i_1<i_2$
	<`LES` ; C, i_2:i_1:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R> if $i_1\ge i_2$
`GRE`	<`GRE` ; C, i_2:i_1:S, G, T, R> ⇒ <C, S, G, T, R> if $i_1>i_2$
	<`GRE` ; C, i_2:i_1:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R> if $i_1\le i_2$
`GEQ`	<`GEQ` ; C, i_2:i_1:S, G, T, R> ⇒ <C, S, G, T, R> if $i_1\ge i_2$
	<`GEQ` ; C, i_2:i_1:S, G, T, R> ⇒ <`FAIL` ; ∅, S, G, T, R> if $i_1<i_2$
`INT_EQL`	<`INT_EQL` ; C, i_2:i_1:S, G, T, R>
	⇒ <C, S, G, T, R> if $i_1=i_2$
	<`INT_EQL` ; C, i_2:i_1:S, G, T, R>
	⇒ <`FAIL` ; ∅, S, G, T, R> if $i_1\ne i_2$
`INT_NEQ`	<`INT_NEQ` ; C, i_2:i_1:S, G, T, R>
	⇒ <C, S, G, T, R> if $i_1\ne i_2$
	<`INT_NEQ` ; C, i_2:i_1:S, G, T, R>

⇒ <`FAIL` ; ∅, S, G, T, R> if i_1=i_2

Manipulation of Variables

`BIND` <`BIND` ; C, n_2:$$n_1$:S, G[$n_1$,$n_2$], T, R>
⇒ <C, S, G[n_1:=IEXP n_2], T, R>

`BT_BIND` <`BT_BIND` ; C, n_2:$$n_1$:S, G[$n_1$,$n_2$], T, R>
⇒ <C, S, G[n_1:=IEXP n_2], (n_1,contents(n_1)):T, R>

`FREE` <`FREE` ; C, t:S, G[t=INT T, n_0, ..., n_x], (n_k,p_k):...:(n_0,p_0):T, R>
⇒ <C, S, G[contents(n_0):=p_0,...,contents(n_x):=p_x], T, R>

`LEVEL` <`LEVEL` ; C, S, G, T, R> ⇒ <C, n:S, G[n:=INT T], T, R>

E Compilation Rules for TAM-Rewrites

Compilation of a TAM Rule (C[[ ]])

Term Rewrite Rules

C[[$f(a_1,a_2,...,a_k)$ -t-> *body*]]
→ **C**[[$f(a_1,a_2,...,a_k)$:- succeed() -t-> *body*]]

C[[$f(a_1,a_2,...,a_k)$:- *condition* -t-> ! $g(b_1,b_2,...,b_j)$]]
:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k
→ `NEXT` n ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ;
V[[*condition*]] ; **V**[[$g(b_1,b_2,...,b_j)$]] ;
P[[*condition*]] ;
B[[b_1]] ; **B**[[b_2]] ; ... ; **B**[[b_j]] ; `REDIRECT` g , j

C[[$f(a_1,a_2,...,a_k)$:- *condition* -t-> *body*]]
:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k
→ `NEXT` n ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ;
V[[*condition*]] ; **V**[[*body*]] ; **P**[[*condition*]] ;
B[[*body*]] ; `RETURN`

Graph Rewrite Rules

C[[$f(a_1,a_2,...,a_k)$ -g-> *body*]]
→ **C**[[$f(a_1,a_2,...,a_k)$:- succeed() -g-> *body*]]

C[[$f(a_1,a_2,...,a_k)$:- *condition* -g-> ! $g(b_1,b_2,...,b_j)$]]
:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k
→ `NEXT` n ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ;
V[[*condition*]] ; **V**[[$g(b_1,b_2,...,b_j)$]] ;
P[[*condition*]] ;
B[[b_1]] ; **B**[[b_2]] ; ... ; **B**[[b_j]] ; `REDIRECT` g , j

C[[$f(a_1,a_2,...,a_k)$:- *condition* -t-> *body*]]
:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k
→ `NEXT` n ; **H**[[a_1]] ; **H**[[a_2]] ; ... ; **H**[[a_k]] ; n:=@entries ;
V[[*condition*]] ; **V**[[*body*]] ; **P**[[*condition*]] ;
B[[*body*]] ; `WRETURN`

Condition Rules

$\mathbf{C}[[\ f(a_1,a_2,...,a_k)$:- *condition.*]]

:- @environment:=[a_1::1,a_2::1,...,a_k::k], @entries:=k

→ `NEXT` n ; $\mathbf{H}[[\ a_1\ ]]$; $\mathbf{H}[[\ a_2\ ]]$; ... ; $\mathbf{H}[[\ a_k\ ]]$; n:=@entries ;

$\mathbf{V}[[$ *condition* $]]$; $\mathbf{P}[[$ *condition* $]]$; `SUCCESS`

Creating the Environment

$\mathbf{V}[[$ *unary_op* $x\ ]] \rightarrow \mathbf{V}[[\ x\ ]]$

$\mathbf{V}[[\ x$ *binary_op* $y\ ]] \rightarrow \mathbf{V}[[\ x\ ]]$; $\mathbf{V}[[\ y\ ]]$

$\mathbf{V}[[\ f(a_1,a_2,...,a_k)\ ]] \rightarrow \mathbf{V}[[\ a_1\ ]]$; $\mathbf{V}[[\ a_2\ ]]$; ... ; $\mathbf{V}[[\ a_k\ ]]$

$\mathbf{V}[[\ v$:variable]] :- environment_lookup(v,_) → ∅

$\mathbf{V}[[\ v$:variable]] :- environment_add(v,_) → `VAR`

$\mathbf{V}[[\ x\ ]] \rightarrow \varnothing$

Body Rules (B[[ ]])

$\mathbf{B}[[\ i$:integer]] → `INT` i

$\mathbf{B}[[\ s$:string]] → `STR` s

$\mathbf{B}[[$ TRUE]] → `PUSH TRUE`

$\mathbf{B}[[$ FALSE]] → `PUSH FALSE`

$\mathbf{B}[[\ v$:variable]] :- environment_lookup(v,pos) → `ARG_COPY` pos

$\mathbf{B}[[$ \$$v$:variable]] :- environment_lookup(v,pos) → `ARG_DUP` pos

$\mathbf{B}[[$ @v]] → `GBL_COPY` v

$\mathbf{B}[[$ \$@v]] → `PUSH` v

$\mathbf{B}[[$ []]] → `NIL`

$\mathbf{B}[[\ x::y\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$; $\mathbf{B}[[\ y\ ]]$; `CONS`

$\mathbf{B}[[$ *unary_op* $x\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$; `NODE2` *unary_op*

$\mathbf{B}[[\ x$ *binary_op* $y\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$; $\mathbf{B}[[\ y\ ]]$; `NODE3` *binary_op*

$\mathbf{B}[[\ !f(a_1,a_2,...,a_k)\ ]]$

→ `PREPARE` f ; $\mathbf{B}[[\ a_1\ ]]$; $\mathbf{B}[[\ a_2\ ]]$; ... ; $\mathbf{B}[[\ a_k\ ]]$; `ENACT` k

$\mathbf{B}[[\ f(a_1,a_2,...,a_k)\ ]] \rightarrow \mathbf{B}[[\ a_1\ ]]$; $\mathbf{B}[[\ a_2\ ]]$; ... ; $\mathbf{B}[[\ a_k\ ]]$; `APPLY` f, k

$\mathbf{B}[[\ !\,x\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$; `EVAL`

$\mathbf{B}[[\ \#\,x\ ]] \rightarrow \mathbf{B}[[\ x\ ]]$

Condition Rules (P [[ ]])

$\mathbf{P}[[\ x\ ,\ y\ ]] \rightarrow \mathbf{P}[[\ x\ ]]$; $\mathbf{P}[[\ y\ ]]$

$\mathbf{P}[[\ x\ ;\ y\ ]] \rightarrow$ `OR` L_1 ; $\mathbf{P}[[\ x\ ]]$; `GOTO` L_2 ; L_1:$\mathbf{P}[[\ y\ ]]$; L_2:∅

P[[\+ *x*]] $\to$ `OR` L_1 ; **P**[[*x*]] ; `DROP_BTP` ; `FAIL` ; L_1:$\emptyset$

P[[fail()]] $\to$ `FAIL`

P[[succeed()]] $\to$ $\emptyset$

P[[cut()]] $\to$ `CUT`

P[[call(*x*)]] $\to$ **B**[[*x*]] ; `CALL`

P[[find_level(*x*)]] $\to$ **B**[[*x*]] ; `LEVEL` ; `UNIFY`

P[[return_to_level(*x*)]] $\to$ **B**[[*x*]] ; `FREE`

P[[*x* is *y*]] $\to$ **B**[[*x*]] ; **A**[[*y*]] ; `IS`

P[[*x* takes *y*]] $\to$ **B**[[*x*]] ; **A**[[*y*]] ; `TAKES`

P[[*x a_compare_op y*]] :-

table_lookup(*a_compare_op*,*op*,['=:='::`EQLN`, '=\='::`NEQN`,

'<'::`LES`, '=<'::`LEQ`, '>'::`GRE`, '>='::`GEQ`])

$\to$ **A**[[*x*]] ; **A**[[*y*]] ; *op*

A[[*v*:variable]] :- environment_lookup(*v*,*pos*) $\to$ `COPY_INT` *pos*

A[[@*v*:variable]] $\to$ `GBL_COPY` *v* ; `INT_CONV`

A[[*i*:integer]] $\to$ `PUSH` *i*

A[[*x arith_op y*]] :-

table_lookup(*arith_op*,*op*,['+'::`ADD`,'–'::`SUB`,'/'::`DIV`,'*'::`MUL`])

$\to$ **A**[[*x*]] ; **A**[[*y*]] ; *op*

A[[! *x*]] $\to$ **B**[[*x*]] ; `INT_CONV`

P[[*f*()]] :- library_predicate(*f*()) $\to$ `LIB_CALL0` *f*

P[[$f(a_1)$]] :- library_predicate($f(a_1)$) $\to$ **B**[[a_1]] ;`LIB_CALL1` *f*

P[[$f(a_1,a_2)$]] :- library_predicate($f(a_1,a_2)$) $\to$ **B**[[a_1]] ; B[[a_2]] ; `LIB_CALL2` *f*

P[[$f(a_1,a_2,a_3)$]] :- library_predicate($f(a_1,a_2,a_3)$)

$\to$ **B**[[a_1]] ; **B**[[a_2]] ; **B**[[a_3]] ; `LIB_CALL3` *f*

P[[$f(a_1,a_2,a_3,a_4)$]] :- library_predicate($f(a_1,a_2,a_3,a_4)$)

$\to$ **B**[[a_1]] ; **B**[[a_2]] ; **B**[[a_3]] ; **B**[[a_4]] ; `LIB_CALL4` *f*

P[[$f(a_1,a_2,a_3,a_4,a_5)$]] :- library_predicate($f(a_1,a_2,a_3,a_4,a_5)$)

$\to$ **B**[[a_1]] ; **B**[[a_2]] ; **B**[[a_3]] ; **B**[[a_4]] ;

B[[a_5]] ; `LIB_CALL5` *f*

P[[$f(a_1,a_2,\ldots,a_k)$]]

$\to$ `PREPARE` ; **B**[[a_1]] ; **B**[[a_2]] ; ... ; **B**[[a_k]] ; `ENACT` *k*

P[[*x compare_op y*]] :-

table_lookup(*compare_op*,*op*,

['=='::`EQL`, '\=='::`NEQ`, '='::`UNIFY`, ':=='::`MATCH`])

$\to$ **B**[[*x*]] ; **B**[[*y*]] ; *op*

P[[*x*:variable *assign_op y*]] :-

table_lookup(*assign_op*,*op*,[':='::`BIND`, '::='::`BBIND`])

$\rightarrow$ **B**[[\$ *x*]] ; **B**[[*y*]] ; *op*

Head Pattern Matching Rules (H [[ ]])

H[[*v*:variable]] *n* $\rightarrow \emptyset$

H[[*v*:variable : *x*:nonvariable]] *n*

:- @environment=(*v*::*n*)::@environment $\rightarrow$ **H**[[*x*]]

H[[*x*:nonvariable : *v*:variable]] *n*

:- @environment=(*v*::*n*)::@environment $\rightarrow$ **H**[[*x*]]

H[[*v*:variable : *w*:variable]] *n*

:- @environment=(*w*::*n*)::(*v*::*n*)::@environment $\rightarrow \emptyset$

H[[*x*]] *n* $\rightarrow$ `ARG_COPY` *n* ; **Hd**[[*x*]]

Hd[[*i*:integer]] $\rightarrow$ `C_INT` *i*

Hd[[*s*:string]] $\rightarrow$ `C_STR` *s*

Hd[[[]]] $\rightarrow$ `C_IMM NIL`

Hd[[TRUE]] $\rightarrow$ `C_IMM TRUE`

Hd[[FALSE]] $\rightarrow$ `C_IMM FALSE`

Hd[[*v*:variable]] :- environment_add(*v*,*pos*) $\rightarrow$ `ARG_SET` *pos*

Hd[[@*g*:variable]] $\rightarrow$ `PUSH` *g* ; `SWAP` ; `MATCH`

Hd[[*x* : *y*]] $\rightarrow$ `COPY` ; **Hd**[[*x*]] ; **Hd**[[*y*]]

Hd[[*integer]] $\rightarrow$ `LIB_CALL1 integer`

Hd[[*string]] $\rightarrow$ `LIB_CALL1 string`

Hd[[*atom]] $\rightarrow$ `LIB_CALL1 atom`

Hd[[*var]] $\rightarrow$ `LIB_CALL1 var`

Hd[[*nonvar]] $\rightarrow$ `LIB_CALL1 nonvar`

Hd[[*unary_op x*]] $\rightarrow$ `C_NODE2` unary_op ; **Hd**[[*x*]]

Hd[[*x binary_op y*]] $\rightarrow$ `C_NODE3` binary_op ; **Hd**[[*x*]] ; **Hd**[[*y*]]

Hd[[$f(a_1,a_2,...,a_k)$]] $\rightarrow$ `C_NODE3 FUNC` ; `C_IMM` *f* ; **Hd**[[$(a_1,a_2,...,a_k)$]]

Queries (Q [[ ]])

Q[[*x*]] :- @environment=[], @entries=0 $\rightarrow$ **V**[[*x*]] ; **B**[[*x*]] ; `END`

Acknowledgements

I would first like to thank my PhD Supervisor Chris Hankin, who over several years has not only provided much helpful advice, but also broadened my outlook on the subject, and spent a great deal of time in careful checking of the many draft versions on this thesis. I am also in debt to Guido Jouret for proof reading and commenting upon the submitted version, and to my examiners Peter Burton and John Glauert for suggestions for improvements in the final version.

This work was conducted whilst I was working under Dov Gabbay as a Research Assistant at Imperial College, partly funded by the EC as part of the Esprit *TEMPORA* project number E2469. I would like to thank all my colleagues at Imperial College, and in particular Dov Gabbay and Richard Owens for providing a stimulating environment in which to conduct research. My work on the use of graph rewriting to implement logic programming began whilst I was employed by ICL, working with Ed Babb and David Cooper, whom I would like to acknowledge for their contributions to discussions on the subject.

Finally I will like to be let known my gratitude to my parents, John and Shirley, and brother James for giving me much personal support during my period of study.

Bibliography

[Aug84] *A Compiler for Lazy ML,*
L.Augustsson,
In *Proceedings of the ACM Symposium on Lisp and Functional Programming*,
ACM, August 1984.

[Bab91] *An Incremental Pure Logic Language with Constraints and Classical Negation*,
Ed Babb,
In *Logic Programming: Expanding the Horizons,*
Ed. Tony Dodd, Richard Owens & Steve Torrance,
Intellect Books/Ablex Publishing Corporation 1991.

[Bar87a] *Term Graph Rewriting*,
H.P.Barendregt *et al*,
PARLE : Parallel Architectures and Languages Europe,
Springer Verlag LNCS 259, 1987.

[Bar87b] *Towards an Intermediate Language based on Graph Rewriting*,
H.P.Barendregt *et al*,
PARLE : Parallel Architectures and Languages Europe,
Springer Verlag LNCS 259, 1987.

[Bel86] *The Relation Between Logic and Functional Languages: A Survey,*
Marco Bellia & Giorgio Levi,
Journal of Logic Programming, Vol 3, pp217-236, 1986

[Fie88] *Functional Programming,*
Anthony Field & Peter Harrison,
Addison-Wesley, 1988

[Gab85] *A Tutorial on the Warren Abstract Machine for Computational Logic,*
John Gabriel, Tim Lindholm, E.L.Lusk & R.A.Overbeek,
Argonne National Laboratory,
Argonne, Illinois, June 1985.

[Gab89] *The Declarative Past and Imperative Future,*
Dov Gabbay,
Temporal Logic in Specification: Altrincham Workshop 1987,
LNCS 398 pp 409-448, Springer-Verlag, 1989.

[Gla87] *Dactl: A Computational Model and Compiler Target Language based on Graph Reduction,*
J.R.W.Glauert, J.R.Kennaway & M.R.Sleep,
ICL Technical Journal, Vol. 5,Issue 3, May 1987.

[Gla88a] *Final Specification of Dactl,*
J.R.W.Glauert, J.R.Kennaway, M.R.Sleep & G.W.Somner,
Univ. of East Anglia, Internal Report SYS-C88-11.

[Gla88b] *A Parallel Implementation of GHC,*
J.R.W.Glauert & G.A.Papadopoulos,
In *Proceedings of the International Conference on Fifth Generation Computer Systems,*
ICOT, Tokyo, 1988.

[Gla91] *Dactl: An Experimental Graph Rewriting Language,*
J.R.W.Glauert, J.R.Kennaway, I.King & M.R.Sleep,
Univ. of East Anglia, Internal Report.

[Han91] *Static Analysis of Term Graph Rewriting Systems,*
Chris Hankin,
PARLE '91 Conference Proceedings,
Springer-Verlag LNCS, 1991.

[Hen89] *Constraint Satisfaction in Logic Programming,*
Pascal Van Hentenryck,
MIT Press, 1989.

[Hog91] *Essentials of Logic Programming,*
Christopher Hogger,
OUP, Graduate Texts in Computer Science 1.

[Joh84] *Efficient Compilation of Lazy Evaluation,*
T.Johnsson,
In *Proceedings of the ACM Conference on Compiler Construction,*
Montreal, pp.58-69, June 1984.

[Ken84] *An Outline of Some Results of Staples on Optimal Reduction Orders in Replacement Systems*,
J.R.Kennaway,
Univ. of East Anglia,
Internal Report CSA/19/1984.

[Klo92] *Term Rewriting Systems*,
J.Klop,
In *Handbook of Logic in Theoretical Computer Science*, *Vol 1*,
Ed. S.Abramsky, D.Gabbay & T.Maibaum,
OUP, 1992.

[Knu71] *Simple Word Problems in Universal Algebras,*
Donald Knuth & Peter Bendix,
In *Proceedings of Conference on Computing Problems in Abstract Algebra,*
Ed. Leech,
Pergamon Press, 1970.

[Koo91] Efficient Graph Rewriting using the Annotated Functional Strategy,
P.W.M.Koopman, J.E.W.Smetsers, M.C.J.D.van Eekelen & M.J.Plasmeijer,
Proceedings of the Semagraph '91 Symposium,
Nijmegen Technical Report No. 91-25, 1991.

[Loc90] *An Abstract Machine for the Implementation of Functional Logic Programming Languages,*
Hendrik Lock,
GMD National Research Lab. at the University of Karlsruhe,
Vincenz Prießnitz Straße 1,
D-7500 Karlsruhe 1, FRG,
July 1990.

[McB88] *The PLL User's Guide*,
Peter McBrien,
Internal Report,
ICL, SSC, Bracknell, Berks.

[McB91a] *Implementing Logic Languages by Graph Rewriting*,
Peter McBrien,
In *Logic Programming: Expanding the Horizons,*
Ed. Tony Dodd, Richard Owens & Steve Torrance,
Intellect Books/Ablex Publishing Corporation 1991.

[McB91b] *Implementing Logical Variables and Disjunctions in Graph Rewrite Systems,*
Peter McBrien,
Proceedings of the Semagraph '91 Symposium,
Nijmegen Technical Report No. 91-25, 1991.

[McB92] *TAM-Rewrite Language: Implementation in 'C',*
Peter McBrien,
Imperial College, Dept. of Computing, 1992.

[Nie92] *Semantics with Applications,*
Hanne Riis Nielson & Flemming Nielson,
Wiley, 1992.

[Pap89] *A Fine-Grain Parallel Implementation of PARLOG,*
TAPSOFT '89,
Barcelona, Spain,
Springer Verlag, LNCS 352, pp 313-327, 1989.

[Pey87] *The Implementation of Functional Programming Languages*,
Simon Peyton Jones,
Prentice Hall, 1987.

[Raa90] *Graph-Based Proof Procedures for Horn Clauses,*
Stan Raatz,
Progress in Computer Science and Applied Logic,
Birkhaüser, 1990.

[Ric71] *The Portability of the BCPL Compiler,*
Martin Richards,
Software – Practice and Experience,
Vol. 1, No. 2, pp.135-146.

[Ric89] *Clausal Form Logic,*
Tom Richards,
Addison Wesley, 1989.

[Sha87] *Concurrent Prolog: Collected Papers,*
Ed: Ehud Shapiro,
MIT Press, 1987.

[Sta80a] *Computation on Graph-Like Expressions*,
John Staples,
Theoretical Computer Science,
Vol 10, pp171-185, 1980, North-Holland.

[Sta80b] *Optimal Evaluation of Graph-Like Expressions*,
John Staples,
Theoretical Computer Science,
Vol 10, pp297-316, 1980, North-Holland.

[Ste86] *The Art of Prolog,*
Leon Sterling & Ehud Shapiro,
MIT Press, 1986.

[Tre76] *An Introduction to Data Structures with Applications*,
J.P.Trembley and P.G.Sorenson,
McGraw-Hill, 1976.

[War83] *An Abstract Prolog Instruction Set,*
David H.D. Warren,
SRI International Technical Note 309, October 1983.

Index

W