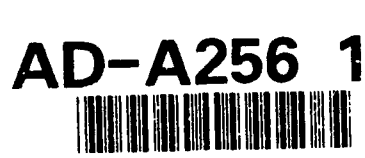

Compiling Prolog to Standard ML: Some Optimizations

Luke Hornof
September 9, 1992
CMU-CS-92-166

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Using a high level language to develop a Prolog system offers many advantages. Development time is decreased, maintainability is increased, and modularity can be utilized. We used Standard ML to develop such a system due to its many powerful features, such as its efficient garbage collection and strong type checking. Of particular consideration was to determine the practicality of this high level system—whether or not a reasonable performance could be obtained. This paper focuses on the optimizations implemented which increase the performance of the system. They include Indexing, Last Call Optimization, Garbage Reduction, and String Compare. All of the combined greatly enhance the performance of the system, which approaches that of a low-level implementation for certain types of programs.

This report was submitted in partial fulfillment of the requirements for the Senior Honors Research Program in the School of Computer Science at Carnegie Mellon University.

Keywords: Prolog, WAM, Standard ML, programming languages, optimizing compilers, compilers

1. Introduction

Prolog is a language for programming in logic. Logic programming offers an alternative to conventional programming. Instead of the usual von Neumann based model, it is derived from an abstract model. A logic program expresses knowledge about a problem with a set of logical axioms. This program can then be executed by providing it with a logical statement to be proved—the goal. The execution solves the problem by proving or disproving the goal statement, given the assumptions the logic program contains[10].

Therefore it should not be surprising that compiling Prolog programs into efficient object code is quite different from compiling languages like C or Pascal. In 1983, David H. D. Warren designed an abstract machine, called the *Warren Abstract Machine* (WAM), to be used as an efficient target for Prolog compilers [11]. The clearest description of the WAM is given by Aït-Kaci [1]. The reader may find Aït-Kaci's tutorial and an additional paper [6] as useful background material to help understand this paper.

The WAM has become the standard model for compiling Prolog, and logic programming languages in general. It can be divided into two parts—the instruction set and the memory architecture. The instruction set provides operations for building and destructuring Prolog data structures, allocating and binding variables, unification, backtracking, and search control. The memory architecture consists of registers, a heap, a trail stack, and an environment stack. It is designed to be implemented in a low-level language such as C, or directly simulated in machine code. Both of these approaches produce the high performance necessary for a Prolog system to be usable.

Nevertheless, using low-level languages has its disadvantages. First of all, certain features of the WAM are difficult to implement. For example, it is hard to keep track of environments when implementing backtracking, and allocating complex data structures on the heap can become quite tricky and detailed. Second, lack of modularity and type safety increases development time. Also, Prolog programs generate garbage on the heap, so a garbage collector must be written. But perhaps the biggest disadvantage is that logic programming languages are still evolving and changing, and making frequent modifications to experimental low-level systems is hard.

Programming in a modern high-level language to a large extent avoids these problems, although a high-level implementation of the WAM might be too slow for practical use.

The goal of our research is to find out whether it is possible to implement a Prolog system in the high-level language Standard ML [8] with comparable performance to the lower-level language implementations. We chose Standard ML because recent progress in its compilation led us to believe that it might be competitive with lower-level languages for a Prolog system. We expected this approach to offer the software engineering advantages mentioned above, such as shortened development time, ease of maintainability, and modularity, as well as a few disadvantages, such as a slight deviation from the WAM.

Another question which we kept in mind was whether there were any SML features, such as its strong type checking or its functional nature, which would make it difficult or even impossible to implement some of the more complicated Prolog features, such as assert/retract or meta-programming.

The first step in writing this system involved working out a specific description of the how we would model the WAM in SML. By taking our time and paying close attention to details, our original design proved to be effective. By following it, we fairly easily implemented Core Prolog

with no major problems. After that, we added cuts, arithmetic, control functions, file I/O, and meta-programming. The optimizations which were also added include indexing, last call, string compare, and garbage reduction. All of these were added without altering our original design much, although meta-programming and string compare both required some minor changes.

Section 2 describes the design we followed when developing our system. Our first goal was concerned with determining whether our method of compiling Prolog to SML was possible. Therefore we were more interested in obtaining a working version, rather than a highly optimized one. Section 3 then explains in detail the elements of which this original system is composed. Sections 3.1–3.3 introduce the basics: terms, queries, and programs. Then in Sections 3.4–3.6 we add argument registers, flat resolution, and the final additions needed to complete Core Prolog. We were quite pleased with this original system and wanted to continue further development. We divided the project into two independent parts. Bárbara Moura worked on extending the core system. Her results can be found in a paper she is currently writing [9]. I concentrated on implementing optimizations which would increase the overall performance. These optimizations are explained in Section 4, and include Indexing, Last Call Optimization, Garbage Reduction, and String Compare. In Section 5 we present our results and analyze them. Finally, Section 6 contains the conclusion to the performance aspect of our project.

2. Compiling Prolog into Standard ML

We chose SML for many reasons. This section will explain our approach to writing the system in SML. Its polymorphic type system provides complete type safety in a way which is convenient for programmers. Also, higher order functions allow control flow to proceed via success continuations. Further, the language supports a module system which provides type-safe separate compilation and parameterized modules. In addition, the code compiled by the Standard ML of New Jersey compiler [2] is highly efficient, particularly in garbage collection and continuation-passing. We wanted our Prolog system to benefit from all of the advantages of using this high-level language.

As we will see in Section 3, we utilize SML's efficient features in our system in the following way. We use SML *datatypes* to represent the WAM's terms, which can then be stored easily on SML's *heap*. *References* are used to implement substitution during unification. Although SML is a mostly functional programming language, it still allows side-effects, such as the ones created by these references. Also, SML *functions* are defined to correspond directly to WAM instructions. These functions are then compiled, which makes their execution more efficient than interpreting WAM instructions as is commonly done. This also makes reading the ML code produced by our compiler very much like reading the WAM object code produced by a conventional Prolog compiler. These functions can be passed as *success continuations* and used to implement backtracking. This makes it particularly simple to keep track of environments, unlike in a low-level implementation. And finally, we made use of the ML *modules* so that we could develop separate components of the Prolog compiler and the WAM, and then easily combine them.

We also take advantage of several of SML's built in features. These tools are needed in the Prolog system. Since we do not have to develop them ourselves, we further decrease our development time. For example, we do not have to write our own parser for the compiler—we can simply use MLyacc and MLlex to automatically generate it for us. And instead of taking care of memory management ourselves, we have the SML/NJ implementation's highly efficient garbage collection system. Utilizing these efficient and powerful tools eliminates the need for ı ɹnr

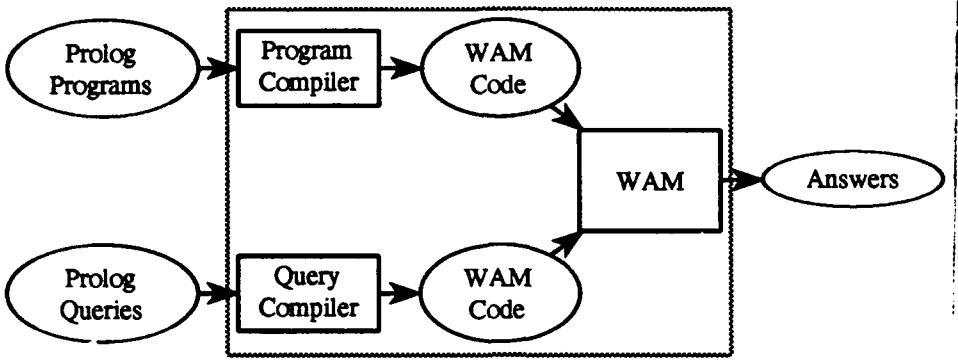

Figure 1: This figure shows the structure of our system. Note that the objects inside square are written in SML.

The structure of our system is a bit complicated. The code for the runtime system, our version of the WAM written in SML, is included and explained in this paper. Also included are many examples of the SML WAM code which is produced by our compiler. Determining the practicality of using SML for this WAM code is an important aspect of our research. The runtime system is defined once, before compilation, and never changes. The SML WAM code is generated at runtime, and can be thought of as input to the WAM. The actual "Prolog to SML" compiler is also written in SML. This paper does not address this compiler, but understanding its place in the system is helpful since we will be making references to it.

A diagram illustrating the relationships between these different components can be seen in Figure 1. From this picture we can see how Prolog programs and queries entered by the user are compiled into their respective SML WAM codes. This two codes, written in SML, are the input to the WAM, which is also written in SML. The WAM then executes program and query WAM code and returns the appropriate results to the user.

3. A small subset of Prolog (L_0)

We closely followed Aït-Kaci's tutorial of the WAM [1] when developing our system. In a fashion similar to his, we will first describe a small subset of Prolog and then add to it in subsequent sections. The initial subset described in this subsection includes terms, queries, and programs, and is referred to as L_0 by Aït-Kaci in his tutorial.

A Prolog program establishes relationships between logical objects. Once that is done, the program is presented with a goal to prove in the form of a query. If the query does not unify with the program, then the goal fails, and the user is notified of this. If they do unify, success is indicated by returning the bindings of query variables unified in the process.

In this first language, L_0, failure during unification raises an ML exception, which is handled by printing out the string "no". Otherwise the query and the program unify, and the list of bound query variables is printed out.

3.1. Term Representation

There are two types of terms, structures and variables. A structure is of the form $f(t_1, \ldots, t_n)$ where f is the structure's *name*, n is the number of subterms the structure contains (known as its *arity*), and the t_i's are the *subterms*. Structures are distinguished by their unique *functor*, which is the concatenation of the name and arity of the form f/n. By convention, functor names begin with a lower case letter. Structures whose arity is zero have no subterms, and are referred to as *constants*.

A variable is initially unbound when it is created. During the execution of a Prolog program it may be bound to another term, either structure or variable. The convention for variables is that they begin with an upper case letter. The information needed to store structure and variable terms in SML is represented by the following `datatype` definition (comments are enclosed by `(* *)`).

```
datatype term =                          (* terms *)
    STR of string * int * term list      (* structure *)
  | REF of termref ref                   (* variable *)
and termref =                            (* term reference *)
    UNBOUND                              (* unbound variable *)
  | BOUND of term                        (* bound variable *)
```

This datatype defines a term to be either a structure (`STR`) or a variable (`REF`). The structure term consists of a string for the name, an integer for the arity, and a list of subterms for its arguments. The variable term is an SML reference, which allows variables to be bound. This reference, called `termref`, is either `UNBOUND` or `BOUND` to another term.

With this definition, we can see how a Prolog term is represented in SML. A simple term such as $foo(h(X), W, g(W, X))$ would become this SML structure:

```
STR ("foo",3,[STR ("h",1,[REF (ref UNBOUND)]),
              REF (ref UNBOUND),
              STR ("g",2,[REF (ref UNBOUND),REF (ref UNBOUND)])]).
```

The structure terms, such as *foo*, *h*, and *g*, are created with their name, arity, and list of subterms. The variables terms, in this case X and W, are initialized with unbound values, `REF (ref UNBOUND)`. It is important to realize, though, that the X's and W's in the term $foo(h(X), W, g(W, X))$ are shared, even though this is not apparent in the printed representation. To see this, we could print out the term unambiguously, as follows:

```
X3 = REF (ref UNBOUND)                   (* X *)
X4 = REF (ref UNBOUND)                   (* W *)
STR ("foo",3,[STR ("h",1,[X3]),          (*  foo(h(X),   *)
              X4,                        (*      W,      *)
              STR ("g",2,[X4,X3])])      (*      g(W,X). *)
```

Though `X2` and `X4` have the same initial values, this representation shows that they both occur twice in the term. A change made to either variable will cause a change to both of its occurrences in the term.

3.2. Query Representation

A query is represented by an SML function named `query` which builds a structure term (and its subterms) from the given information and puts it into a variable named `x1`. It then runs the program, by executing another SML function named `p`, on this newly built `x1` term. The execution of this program will then decide the the results of the query.

In order to transform the Prolog query into this SML function, we have implemented two simple WAM instructions which help build the query term. They are `put_structure` and `put_variable`. The former instruction creates SML structures when they are encountered in the query, and the latter is used to create variables. Their definitions are as follows:

```
fun put_structure (f, arity) vars = STR (f, arity, vars)

fun set_variable () = REF (ref UNBOUND)
```

The first instruction, `put_structure`, creates a `STR` structure by taking the structure's functor and its list of subterms, and returning them in the form defined by the structure term datatype. The other instruction, `set_variable`, returns `REF (ref UNBOUND)`, the value with which new variables are initialized to when they are first encountered.

The query compiler produces a function named `query` which consists of a sequence of these instructions. When executed, the appropriate term structure is created and stored in `x1` on the heap, and the program is called. The SML code which follows is what the compiler produces for the query $foo(h(X), W, g(W, X))$. The term which is produced and stored in `x1` by the execution of this code is the *foo* structure term previously described.

```
(* foo(h(X),W,g(W,X)) *)
fun query () =
    let
        val x3 = set_variable ()
        val x2 = put_structure ("h", 1) [x3]
        val x4 = set_variable ()
        val x5 = put_structure ("g", 2) [x4, x3]
        val x1 = put_structure ("foo", 3) [x2, x4, x5]
    in
        p(x1)
    end
```

3.3. Program Representation

A program is represented by an SML function named `p` which performs operations on a query term already built and stored on the heap. This program is what determines whether the query fails or succeeds. There are three operations which the SML program function can perform on the query term. The first is `get_structure`, which is used each time a structure is encountered in the program. The next is `unify_variable`, called the first time a variable occurs. The last function, `unify_value`, is used for any repeated occurrence of a variable. These functions are defined below.

```
datatype mode = READ | WRITE

fun get_structure (f1, arity1, Xi) =
    let val addr = deref(Xi)
    in
        case (addr) of
            (REF (r as ref UNBOUND)) =>
                let fun for(0) = nil
                      | for(n) = (REF (ref UNBOUND))::(for (n-1))
                    val lst1 = for (arity1)
                in
                    (r := BOUND (STR (f1, arity1, lst1));
                     (lst1, WRITE))
                end
          | (STR (f2, arity2, lst2)) =>
                if (f1 = f2) andalso (arity1 = arity2) then
                    (lst2, READ)
                else
                    raise DoesNotUnify "get_structure: STR"
          | (_) => raise DoesNotUnify "get_structure: else"
    end

fun unify_variable lst = lst

fun unify_value (Xi, s::S, mode) =
    ((case (mode) of
          READ => unify(Xi, s)
        | WRITE => bind(s, Xi));
     S)
  | unify_value (_, nil, _) = raise InternalError "unify value"
```

The first instruction, **get_structure**, takes three arguments. They are a structure's name, *f1*, its arity, *arity1*, and an argument register, X_i. The purpose of this instruction is to determine if the structure defined by *f1* and *arity1* unifies with the contents of the argument register passed in. To do so, it first dereferences (explained in detail at the end of this section) the argument register and does one of two things. If the dereferenced X_i is an unbound variable, unification succeeds by simply binding this variable to a structure term with the structure's name, arity, and a list of unbound variable subterms. This list of subterms and the mode **WRITE** are returned, which will be used by subsequent instructions to continue processing the program. **WRITE** mode indicates that the subterms in the list are unbound.

On the other hand, if the dereferenced X_i is a second structure, *f2*, *arity2*, and *lst2*, then the two structure's functors are compared for equality. If either the names or the arities differ, unification fails and the exception **DoesNotUnify** is raised. Otherwise, the second structure's list of subterms, *lst2*, is returned with a mode of **READ**. This mode is chosen since *lst2*'s variables may already be bound.

The last two instructions are much shorter and simpler. The first time a register argument is seen, **unify_variable** is called. This function merely returns the list of terms passed in. The way

in which this is used is to allow the SML WAM code to remove the head of the list and set a variable to its value. We will see how this happens when we look at our examples.

Repeated occurrences of a variable use the instruction **unify_value**. It takes an argument register (X_i), the list of subterms (where s is the head of the list and S is the tail), and the mode. This function uses the mode to decide what to do. If **get_structure** set a mode of **WRITE**, then s is known to be unbound and **unify_value** simply binds s to X_i. If **READ** mode was set, then s may already be bound, in which case full unification must be performed. In either case, the tail of the list of subterms is returned.

Using these three functions, a program term is compiled into the SML function p which takes the query term as input and returns it after performing its specific sequence of operations to it. A program, such as $foo(Z, h(Y), g(Z, h(a)))$, would be compiled into the following SML function.

```
(* foo(Z,h(Y),g(Z,h(a))) *)
fun p x1 =
    let
        val (S, mode) = get_structure ("foo", 3, x1)    (* x1 = foo *)
        val (x2::S) = unify_variable S                  (* Z *)
        val (x3::S) = unify_variable S                  (* x3 *)
        val (x4::S) = unify_variable S                  (* x4 *)
        val (S, mode) = get_structure ("h", 1, x3)      (* x3 = h *)
        val (x5::S) = unify_variable S                  (* Y *)
        val (S, mode) = get_structure ("g", 2, x4)      (* x4 = g *)
        val S = unify_value (x2, S, mode)               (* Z *)
        val (x6::S) = unify_variable S                  (* x6 *)
        val (S, mode) = get_structure ("h", 1, x6)      (* x6 = h *)
        val (x7::S) = unify_variable S                  (* x7 *)
        val (S, mode) = get_structure ("a", 0, x7)      (* x7 = a *)
    in
        x1
    end
```

Looking at this example, we see how these runtime functions are used to create an SML function which represents the Prolog program. Each structure has its corresponding **get_structure** instruction, which returns a list of its subterms and a mode. Each of these instructions is then followed by unify instructions, one for each subterm in the list. It can be seen here how the list passed to **unify_variable** is returned and split into two parts. A variable is set to the head, and a new S is set to the tail. Since the **unify_value** instruction removes the head of the list itself, it also sets S to the tail of the subterm list.

Another example which may be easier to follow involves relationships between relatives. We will start out with a simple program, *parent*(*margaret*, *tommy*), which will be added to in subsequent sections. This program consists of one literal, which is read, "*margaret* is the *parent* of *tommy*." Here is the SML function, p, generated to represent this program.

```
(* parent(margaret,tommy) *)
fun p x1 =
```

```
let
    val (S, mode) = get_structure ("parent", 2, x1)     (* x1 = parent *)
    val (x2::S) = unify_variable S                      (* x2 *)
    val (x3::S) = unify_variable S                      (* x3 *)
    val (S, mode) = get_structure ("margaret", 0, x2)   (* x2 = margaret *)
    val (S, mode) = get_structure ("tommy", 0, x3)      (* x3 = tommy *)
in
    x1
end
```

Three three main WAM runtime functions use several utility functions. Variable binding is done with **bind**, which takes an unbound term and another term, and binds the first to the second. Dereferencing terms is done with **deref**, which calls itself recursively until either a structure or an unbound variable term is found. The function **samePointer** determines whether two variable terms refer to the same variable.

Unification, the main processing tool in logic programming, is done with a function named **unify**. It takes two terms and binds one to the other if either is an unbound variable. If they are both structures, then it checks if their names and arities are equal, and then unifies corresponding subterms. Unification fails if any part of this process fails. The SML code which defines these utility functions is as follows.

```
fun bind (REF (r as ref UNBOUND), s) = (r := BOUND s)
  | bind (_) = raise InternalError "bind"

fun deref (a as (STR (_,_,_))) = a
  | deref (a as (REF (ref UNBOUND))) = a
  | deref (REF (ref (BOUND t))) = deref t

fun samePointer (REF r1, REF r2) = (r1 = r2)
  | samePointer (_) = false

fun unify (a1, a2) =
    let val d1 = deref a1
        and d2 = deref a2
    in
        (* checks if pointers point to different locations in memory *)
        if samePointer (d1, d2) then
            ()
        else
            case (d1, d2) of
                (REF (r as (ref UNBOUND)), d2) => (r := BOUND d2;
                                                   ())
              | (d1, REF (r as (ref UNBOUND))) => (r := BOUND d1;
                                                   ())
              | (STR (f1, n1, x1), STR (f2, n2, x2)) =>
                    if (f1 = f2) andalso (n1 = n2) then
                        let fun for(nil, nil) = ()
```

```
                        | for(h1::t1, h2::t2) =
                          if (unify(h1, h2) = ()) then
                              for(t1, t2)
                          else
                              raise DoesNotUnify "Diff args"
                        | for (_) = raise InternalError "Unify"
                    in
                        for(x1, x2)
                    end
                else raise DoesNotUnify "Diff functor/arity"
            | (_) => raise InternalError "STR"
    end
```

3.4. Argument Registers (L_1)

We will now make two changes and extend L_0 to a second language referred to as L_1 by Aït-Kaci. To understand these changes, it should be noted that a *literal* is a term whose functor is a predicate, while *terms* are arguments to the predicate. Whereas L_0 only allowed one literal to be defined per program (by a function named p), L_1 will allow more than one. In order to do this, we name the SML functions with the name and arity of the Prolog literal it represents.

The second change which L_1 makes is instead of building the whole query structure in x1, each literal argument is built in a separate argument register. These argument registers are labeled $a_1, a_2, \ldots, a_n$, where n is the arity of the literal. The argument registers for the query $foo(h(X), W, g(W, X))$ would be created with the following values:

```
(* foo(h(X),W,g(W,X)) *)
    a1 = STR ("h",1,[REF (ref UNBOUND)])
    a3 = REF (ref UNBOUND)
    a4 = STR ("g",2,[REF (ref UNBOUND),REF (ref UNBOUND)])
```

The query function, still named query as in L_0, is slightly different due to these two changes. The L_1 code for the query $foo(h(X), W, g(W, X))$ is listed below.

```
(* foo(h(X),W,g(W,X)) *)
fun query () =
    let
        val x2 = set_variable ()
        val a1 = put_structure ("h", 1) [x2]
        val a3 = set_variable ()
        val x5 = set_variable ()
        val a4 = put_structure ("g", 2) [x5, x2]
    in
        (foo3(a1, a3, a4);
         Top.Term.STR ("foo", 3, [a1, a3, a4]))
    end
```

The changes in this new code can easily be seen. First of all, there is no `put_structure` instruction needed for the literal's name, *foo*. Instead, the function `foo3` is called with arguments `a1`, `a3`, and `a4`. After this call, the value is returned to the top level is the *foo* structure with the new values of the argument registers, used to print out substitutions.

The program code for L_1 also looks slightly different. The SML functions generated now have the names of the literals they represent, and the runtime instructions which perform the operations on the argument registers have changed. The new code for the same program, $foo(Z, h(Y), g(Z, h(a)))$, is listed below.

```
(* foo(Z,h(Y),g(Z,h(a))) *)
fun foo3(a1, a2, a3) =
    let
        val (S, mode) = get_structure ("h", 1, a2)      (* a2 = h *)
        val (x4::S) = unify_variable S                  (* Y *)
        val (S, mode) = get_structure ("g", 2, a3)      (* a3 = g *)
        val S = unify_value (a1, S, mode)               (* Z *)
        val (x5::S) = unify_variable S                  (* x5 *)
        val (S, mode) = get_structure ("h", 1, x5)      (* x5 = h *)
        val (x6::S) = unify_variable S                  (* x6 *)
        val (S, mode) = get_structure ("a", 0, x6)      (* x6 = a *)
    in
        ()
    end
```

The name of the function is now `foo3`, and takes it takes three arguments, `a1`, `a2`, and `a3`. There is no `get_structure` for *foo* followed by `unify_variable` instructions for each of the subterms. The rest of the code is very similar to that of L_0, except `x1` is no longer returned since the changes made to the terms on the heap will be reflected in the changes made to the argument registers.

Since the main advantage L_1 has over L_0 is allowing a program to have more than one literal, let us extend our "relatives" example to see how this is done. In this new program, we add the literal *brother*(*bob*, *margaret*) to *parent*(*margaret*, *tommy*). Each function now has a different name, and each argument has the form a_i. The resulting SML code which is now generated can be seen below.

```
(*
 * parent(margaret,tommy).
 * brother(bob,margaret).
 *)
fun parent2(a1, a2) =
    let
        val (S, mode) = get_structure ("margaret", 0, a1)   (* a1 = margaret *)
        val (S, mode) = get_structure ("tommy", 0, a2)      (* a2 = tommy *)
    in
        ()
    end

and brother2(a1, a2) =
```

```
    let
        val (S, mode) = get_structure ("bob", 0, a1)        (* a1 = bob *)
        val (S, mode) = get_structure ("margaret", 0, a2)   (* a2 = margaret *)
    in
        ()
    end
```

3.5. Flat Resolution (L_2)

The next changes we will make will define a new language, L_2, which will allow predicates to have *bodies*. Specifically stated, each L_2 predicate consists of *clauses* of the form a_0 :- $a_1, \ldots, a_n$, where the a_i's are literals and n is the number of literals in the clause. If $n = 0$, then the clause is simply a *fact*, like the predicates we defined in L_1. But, if $n > 0$, then the clause is called a *rule*, with a_0 referred to as the *head*, and the remaining sequence of a_i's called the *body*. These literals in the body are called *goals*, and each goal in a rule must unify in order for the predicate to succeed.

This change does not effect the query terms. Therefore, the SML code for the query term $foo(h(X), W, g(W, X))$ is still the same.

```
(* foo(h(X),W,g(W,X)) *)
fun query () =
    let
        val x2 = set_variable ()
        val a1 = put_structure ("h", 1) [x2]
        val a3 = set_variable ()
        val x5 = set_variable ()
        val a4 = put_structure ("g", 2) [x5, x2]
    in
        (foo3(a1, a3, a4);
         Top.Term.STR ("foo", 3, [a1, a3, a4]))
    end
```

Programs in L_2 not only represent facts, but now must also handle rules. Given a rule, a_0 :- $a_1, \ldots, a_n$, we have to check if all subgoals $a_1, \ldots, a_n$ succeed. To do this, we still treat the head as a fact, but the body is treated like a conjunction of queries. Therefore, when compiling a rule, we first generate the code for the head and follow it with query code for the body. The following code is what the compiler produces for the rule $foo(Z, h(Y), g(Z, h(a)))$:- $h(a), g(X, Z)$.

```
(* Program code for foo(Z,h(Y),g(Z,h(a))) :- h(a), g(X,Z) *)
fun foo(a1, a2, a3) =
    let
        val (S, mode) = get_structure ("h", 1, a2)      (* a2 = h *)
        val (x4::S) = unify_variable S                  (* Y *)
        val (S, mode) = get_structure ("g", 2, a3)      (* a3 = g *)
        val S = unify_value (a1, S, mode)               (* Z *)
        val (x5::S) = unify_variable S                  (* x5 *)
```

```
        val (S, mode) = get_structure ("h", 1, x5)      (* x5 = h *)
        val (x6::S) = unify_variable S                  (* x6 *)
        val (S, mode) = get_structure ("a", 0, x6)      (* x6 = a *)
        val x7 = put_structure ("a", 0) □
        val _ = h (x7)
        val x8 = set_variable ()
        val _ = g (x8, a1)
    in
        ()
    end
```

Looking through this code, we can see occurrences of the usual program instructions used to represent the head of the rule, **get_structure**, **unify_variable**, and **unify_value**. In addition, however, we now also have **put_structure** and **set_variable** instructions previously only seen in queries. This "query code" appended onto the normal program code now checks if the goals unify by making the appropriate predicate calls, and will determine if the entire rule succeeds.

Now let's add a rule to our "relative" program. In addition to the two relationships we have already established, let us add the rule $uncle(X,Y)$:- $brother(X,Z), parent(Z,Y)$. It is read, "$X$ is the *uncle* of Y, if there exists some Z for which X is brother to Z and Z is the parent of Y. A query which makes an inquiry about this *uncle* relationship will now get treated as follows. First, the query's predicate must unify with *uncle*, and in addition two more queries must also be satisfied, namely $brother(X,Z)$ and $parent(Z,Y)$. The SML code which represents this looks like this:

```
(*
 *  parent(margaret,tommy).
 *  brother(bob,margaret).
 *
 *  uncle(X,Y) :- brother(X,Z), parent(Z,Y).
 *)
fun parent(a1, a2) =
    let
        val (S, mode) = get_structure ("margaret", 0, a1)  (* a1 = margaret *)
        val (S, mode) = get_structure ("tommy", 0, a2)     (* a2 = tommy *)
    in
        ()
    end

and brother(a1, a2) =
    let
        val (S, mode) = get_structure ("bob", 0, a1)       (* a1 = bob *)
        val (S, mode) = get_structure ("margaret", 0, a2)  (* a2 = margaret *)
    in
        ()
    end

and uncle(a1, a2) = let
```

```
        val x3 = set_variable ()
        val _ = brother (a1, x3)
        val _ = parent (x3, a2)
    in
        ()
    end
```

3.6. Prolog (L_3)

The last addition we make completes what is known as *Core Prolog*, or *Pure Prolog*. It possesses the minimum functionality required with which to begin writing interesting programs, although most useful systems contain many additional features. The last change needed to complete this final language, L_3, is the allowance of *disjunctive definitions* in programs.

A *definition* is an ordered set of clauses whose head literals have the same predicate name. A definition with n clauses allows a predicate n different chances to unify. When presented with a query, each of the predicate's clauses will attempt unification in the order in which they are written. Initially, the first clause in a definition is called. If it fails, then the next clause is called, and so on. Failing unification no longer stops program execution, it instead tries the alternatives in order. This process of considering alternative clauses upon failure is know as *backtracking*. A predicate now fails only if each clause in its definition fails.

The way we implement this type of control flow in SML involves keeping track of what the next step in the program will be if a function succeeds. We pass the "next step" from function to function in the form of a *success continuation*[3, 6]. Each runtime function now takes a continuation, which is called upon successful unification. The query initially calls the program with a predefined continuation which prints out variable substitutions. This top level continuation is the last function a program will call, only if it has successfully unified all of its parts.

If unification fails at any point during a program, then we do not want to call the success continuation. Rather, we return the unit value which will return control back to the last function called. Therefore, any time a function makes a call and gets control back, failure must have occurred since a continuation was not called.

Adding this functionality to our system requires slight modifications to all of our runtime functions. Success continuations (denoted `sc`) are passed in as a parameter and called upon success. Instead of raising an exception, failure is now represented by returning the unit value (written `()`). These changes can been seen in the new definition of `get_structure`.

```
fun get_structure (f1, arity1, Xi) sc =
    let val addr = deref(Xi)
    in
        case (addr) of
            (r as (REF (ref UNBOUND))) =>
                let fun for(0) = nil
                      | for(n) = (REF (ref UNBOUND))::(for (n-1))
                    val lst1 = for (arity1)
                in
```

```
            (bind (r, STR (f1, arity1, lst1));
             sc (lst1, WRITE))
        end
    | (STR (f2, arity2, lst2)) =>
        if (f1 = f2) andalso (arity1 = arity2) then
            sc (lst2, READ)
        else () (* Does not unify *)
    | (_) => () (*  Does not unify *)
        end
```

Queries also change slightly to incorporate success continuations. As can be seen in the following query code, the success continuation to print out bound query variables is passed in as an argument. The query calls the program, in this case the function named `foo3`, with this continuation.

```
(* Query for $foo(h(X),W,g(W,X)) *)
fun query (sc) =
    let
        val x2 = set_variable ()
        val x1 = put_structure ("h", 1) [x2]
        val x3 = set_variable ()
        val x4 = put_structure ("g", 2) [x3, x2]
    in
        foo3 (x1, x3, x4) (fn () => sc (Top.Term.STR ("foo", 3, [x1, x3, x4])))
    end
```

The way we implemented the disjunction among predicate definitions involved special control functions, `try_me_else`, `retry_me_else`, and `trust_me`. To ensure that backtracking occurs correctly, these functions have to not only deal with the proper control flow, but they also have to maintain each variable's proper value. Specifically, when a clause fails and the next one is to be tried, all of the changes made in this first attempt must be undone. This means that variables bound in this attempt must be reset to unbound. In order to do this, we keep track of the variables which become bound by keeping them in a list, called the *trail*. Each time a variable is bound it is added to the trail. In order to undo bindings which occur during a clause we keep track of the trail length in a variable named *treg*. Prior to a clause's execution, we set the local variable *old_treg* to *treg*, the length of the trail at that point. During a clause's execution, bound variables may be added to the trail, which would in turn increment *treg*. If at any point the clause fails, then we *unwind* the trail by removing variables one at a time, resetting them to unbound, and decrementing *treg*. This is done until *treg* equals *old_treg*, at which point the trail is completely restored.

The trail and and the trail's length are defined below. They are both globally defined to be ML references, which allows all of the runtime functions to access and modify them. Also listed here is the function which unwinds the trail. It takes an integer, which is the number of times to remove and initialize variables from the trail.

```
val trail = ref ([]:(termref ref list))          (* trail *)
val treg = ref 0                                 (* trail length *)
```

```
fun unwind_trail 0 = ()
  | unwind_trail diff_treg =
    let fun unwind ((r1 as (ref (r2 as (BOUND hd))))::tl) =
                (trail := tl;
                 treg := !treg -1;
                 r1 := UNBOUND;
                 unwind_trail (diff_treg - 1))
          | unwind [] = raise InternalError "unwind empty trail"
          | unwind ((ref UNBOUND)::_) = raise InternalError "unwind unbound"
    in
        unwind (!trail)
    end
```

These next three functions are the ones use to control flow and between clauses in definitions.

```
fun try_me_else me els =
    let val old_treg = !treg
    in
        (me ();
         unwind_trail (!treg - old_treg);
         els ())
    end

val retry_me_else = try_me_else

fun trust_me me = me ()
```

The first function **try_me_else** takes two functions as arguments. It first sets **old_treg** to the current length of the trail. Then it proceeds by calling the first function. If this first function succeeds, a continuation will be called and control will never return to **try_me_else**. If control does return, it means that the first function failed. The trail is unwound to its size prior to the execution of the first function, and then the second function is called. If the second returns the unit value indicating failure, **try_me_else** will also return this value, passing on failure to the function which called it.

The second function, **retry_me_else** has functionality identical to **try_me_else**, and is simply defined as that. The only advantage to having two separate functions is it makes our code look more like the instructions in the WAM which we followed.

The last, **trust_me**, is like the first two functions, except it omits the "try_me" part and goes directly to the "else". It will be called last, and therefore takes in only one function as an argument and then calls it. It returns the answer the functions returns.

A definition with multiple clauses uses these three functions in the following way. The first clause is called with **try_me_else**, which takes two arguments. The first argument is the code for the first clause. The second argument is the code for the rest of the definition, which will be called if the first fails. All of middle clauses are similarly called with **retry_me_else**. The last clause

will be called with **trust_me**, which only takes in the code for the last clause as its argument. The resulting SML code will look something like this:

fun name($x_1, x_2, \ldots, x_n$) sc =
  try_me_else
  code for first clause
    retry_me_else
    code for second clause
      ⋮
    retry_me_else
    code for second to last clause
      trust_me
      code for last clause

Using this outline makes is easier to follow Prolog programs compiled into this more complex SML code. Here's an example of a three line program which is compiled into this new language, L_3.

```
(*
 *   foo(Z,h(y),g(Z,h(a))).
 *   foo(X,Y,Z) :- h(X),g(Y,Z).
 *   foo(a,X,Y).
 *)
fun foo3 (x1, x2, x3) sc =
    try_me_else
    (fn (_) => get_structure ("h", 1, x2)
     (fn (x4::nil, mode) =>get_structure ("y", 0, x4)
      (fn (nil, mode) =>get_structure ("g", 2, x3)
       (fn (x5::x6::nil, mode) =>unify_value (x1, x5, mode)
        (fn () => get_structure ("h", 1, x6)
         (fn (x7::nil, mode) =>get_structure ("a", 0, x7)
          (fn (nil, mode) =>sc ()))))))))

    (fn (_) => retry_me_else
     (fn (_) => h1 (x1)
      (fn (_) => g2 (x2, x3)
       (fn (_) => sc ()))))

     (fn (_) => trust_me
      (fn (_) => get_structure ("a", 0, x1)
       (fn (nil, mode) =>sc ()))))
```

The predicate *foo* is now represented by an SML function which is broken into three parts. The first part contains the **try_me_else** and the code for the first clause. The second part contains the **retry_me_else** and the code for the second clause. The last part has the **trust_me** and the code corresponding to this last clause. Execution of this function will try each of clause's codes in turn by backtracking upon failure.

We will also add a disjunctive definition to our "relatives" example. By adding a second clause to the *uncle* predicate, uncle now has two possibilities for success. Notice how the other two predicates, *parent* and *brother*, remain the same since they each only contain one clause.

```
(*
 *   parent(margaret,tommy).
 *   brother(bob,margaret).
 *
 *   uncle(X,Y) :- brother(X,Z), parent(Z,Y).
 *   uncle(bob,cindy).
 *)
fun parent2 (x1, x2) sc =
    get_structure ("margaret", 0, x1)
    (fn (nil, mode) =>get_structure ("tommy", 0, x2)
     (fn (nil, mode) =>sc ()))

and brother2 (x1, x2) sc =
    get_structure ("bob", 0, x1)
    (fn (nil, mode) =>get_structure ("margaret", 0, x2)
     (fn (nil, mode) =>sc ()))

and uncle2 (x1, x2) sc =
    try_me_else
    (fn (_) =>
     let
         val x3 = set_variable ()
     in
         brother2 (x1, x3)
         (fn (_) => parent2 (x3, x2)
          (fn (_) => sc ()))
     end)
    (fn (_) => trust_me
     (fn (_) => get_structure ("bob", 0, x1)
      (fn (nil, mode) =>get_structure ("cindy", 0, x2)
       (fn (nil, mode) =>sc ()))))
```

4. Optimizations

Satisfied with the way Core Prolog turned out, we were encouraged to further develop the system. We decided to divide the project into two parts, which would allow two of us to work parallel. Until this point, we had closely followed the WAM as much as possible. We felt that there were enough differences between the systems that we should rename ours with a different name, the BLAM.[1]

I chose to work on optimizations made to the system, which will be explained in this section. At the same time the optimizations were being worked on, extensions were independently being

[1] BLAM is an acronym for the Bárbara and Luke Abstract Machine.

added. The progress made in both parts was regularly combined into one system. The final system includes all of the extensions as well as optimizations, although versions along the way only included some of each. Therefore, this section will contain some references to extensions of the system not mentioned above, but which were subsequently added. There are other side-effects encountered due to this "co-development". For example, one benchmark program with which the system is tested uses arithmetic. Since this was only added after the first optimization was done, there is no data for this benchmark for our system with no optimizations.

With this in mind, let us proceed and look at each of the optimizations in turn, and how each of them are obtained.

4.1. Indexing

Normal execution of a predicate entails a strictly sequential search over the clauses which form its definition. This is the optimal method of execution if the arguments of the calling predicate are unbound. But, if some of the arguments of this calling predicate are bound, then that information can be used to limit the search. If the type of a particular argument is either a constant or a structure, the search path can be reduced to include only those clauses with matching argument types. For example, if the calling predicate has three arguments, all of which are constants, then the called predicate only needs to consider searching those clauses whose three arguments are constants or variables.

Accommodating all of the different combinations of argument types would be quite complex and expensive, but a suboptimal compromise turns out to be quite reasonable in practice [1]. This compromise is done by only attempting to match the clause's first arguments, which are referred to as indexing *keys*. The following explains how to add indexing to a definition.

The sequence of clauses $C_1, C_2, \ldots, C_n$ is divided up into subsequences $S_1, S_2, \ldots, S_m$, where each S_i is a maximal subsequence of contiguous clauses with non-variable keys. We will see how the number of tests needed to check each clause in a subsequence will be reduced. If the key of a clause is a variable it will unify with anything and therefore must be in a subsequence of its own. These subsequences are then structured in such a way that each will be be tried in turn until one of them succeeds. Let us look at an example with five clauses C_1 through C_5.

$$S_1 \begin{cases} parent(mary, judy). \\ parent(mary, joe). \\ parent(mother(joe), joe). \end{cases}$$
$$S_2 \{ parent(X, Y) :- child(Y, X).$$
$$S_3 \{ parent(joe, son(jim)).$$

From this it can be seen how the first three clauses, C_1 through C_3 are grouped together in S_1. This is the case since their keys are all non-variable. C_4 must be in a subsequence by itself, due to its key being a variable (X). Although C_5 does not have a variable key, it is also by itself since it has no neighboring non-variable clauses.

Previously, this predicate would be compiled into five SML functions, one for each clause. These five functions would then be arranged so that all of them could be tried in turn, taking a maximum of five tries to succeed or fail. With indexing, only three functions are created, one for each subsequence. The first function accommodates all three of the clauses contained in the

first subsequence. The code for the last two functions, consisting of only one clause each, remains identical to those of L_2. If the query's key is not variable, the maximum number of tries needed to resolve the definition is reduced to the number of subsequences the definition can be broken into.

Each subsequence with multiple clauses is compiled into an SML function which checks all of the clauses in one test. Since the keys are either structures or constant, Two SML case statements are used, one for each. These case statements then look up the SML code corresponding to the clauses in the subsequence. SML's highly optimized case statements resolve this lookup in near constant time.

For example, when a *parent* query calls this program, the key of the query must first be determined. If the key is variable, then all three of the clauses must be checked. If the key is a constant, then the constant's case statement can quickly find the corresponding function and then call it. Likewise, a query with a structure key uses the structure's case statement to find and call the correct function. Determining the query's key and is done by the following function.

```
fun switch_on_term (REF (ref (BOUND t)), s1, c1, f1) =
    switch_on_term (t, s1, c1, f1)
  | switch_on_term (STR(name,0,_):Term.term, s1,c1,f1) =
    c1(name)
  | switch_on_term (STR(name,arity,_):Term.term,s1,c1,f1) =
    f1(name,arity)
  | switch_on_term (REF _ :Term.term,s1,c1,f1) =
    s1()
```

This function, `switch_on_term` takes four arguments: the query's key and three functions, `s1_1`, `c1`, and `f1`. These functions represent the corresponding functions to call for variables, constants, and structures, respectively. It determines the query's key calls one of the three functions. If the query is a variable, `switch_on_term` dereferences it by calling itself recursively until either a constant, structure, or unbound variable is found.

Indexed SML WAM code is quite complicated, since it entails all of these changes. By looking through an example, however, seeing how each of the components fits together becomes clearer. The following code is produced from the five *parent* clauses listed above.

```
(*
 *  parent(mary, judy).
 *  parent(mary,joe).
 *  parent(mother(joe), joe).
 *  parent(X,Y) :- child(Y,X).
 *  parent(joe,son(jim)).
 *)
fun parent2 (x1, x2) ctag sc =
    let fun  s1_1()= try_me_else (fn (_) => s1_1a())
                                 (fn (_) => s1_2())
        (* code for subsequence S1 *)
        and s1_1a() = get_structure ("mary", 0, x1)
            (fn (nil, mode) =>get_structure ("judy", 0, x2)
```

```
        (fn (nil, mode) =>sc ()))
    and s1_2() = retry_me_else (fn (_) => s1_2a())
                               (fn (_) => s1_3())
    and s1_2a() = get_structure ("mary", 0, x1)
        (fn (nil, mode) =>get_structure ("joe", 0, x2)
         (fn (nil, mode) =>sc ()))
    and s1_3() = trust_me (fn(_) => s1_3a())
    and s1_3a() = get_structure ("mother", 1, x1)
        (fn (x3::nil, mode) =>get_structure ("joe", 0, x3)
         (fn (nil, mode) =>get_structure ("joe", 0, x2)
          (fn (nil, mode) =>sc ())))
    (* constant case statement *)
    and c1(name) = case (name) of
        "mary" => try_me_else (fn(_)=> s1_1a())
                              (fn(_)=> trust_me (fn(_)=> s1_2a()))
      | (_) => ()
    (* structure case statement *)
    and f1(name,arity) = case (name,arity) of
        ("mother", 1) => s1_3a ()
      | (_) => ()
    (* determine which function to call based on type of x1 *)
    and  s1() = switch_on_term(x1,s1_1,c1,f1)
    (* code for subsequence S2 *)
    and s2() = (call (child2 (x2, x1))
         (fn (_) => sc ()))
    (* code for subsequence S3 *)
    and s3() = get_structure ("joe", 0, x1)
        (fn (nil, mode) =>get_structure ("son", 1, x2)
         (fn (x3::nil, mode) =>get_structure ("jim", 0, x3)
          (fn (nil, mode) =>sc ())))
in
    try_me_else
     (fn (_) => s1())                          (* code for S1 *)
     (fn (_) => retry_me_else
      (fn (_) => s2())                         (* code for S2 *)
      (fn (_) => trust_me
       (fn(_) => s3()))                         (* code for S3 *)
end
```

The first subsequence, S_1, has three functions which perform the control flow, `s1_1`, `s1_2`, and `s1_3`. The code for each of the three clauses are contained in three more functions, `s1_1a`, `s1_2a`, and `s1_3a`. The case statement for the constant *mary* is in `c1`. Notice that since there there are two keys in this subsequence with the same constant, `c1` itself contains a `try_me_else` and a `trust_me`. The case statement for the structure *mother* is in `f1`.

The main function for this subsequence is stored in `s1`. This is the function which takes the query's key, determines its type, and then calls the appropriate function. If the key is a variable, then `s1_1` is called. Since the variable key will unify with each clause, `s1_1` must be sure to try

subsequent clauses upon failure. A constant will call the case statement for constants, `c1`. This statement checks if the constant's name is correct. If it is, then the appropriate clause code is called. Otherwise, the unit value is returned to indicate there is no constant in this subsequence with that name. Finally, the query's key may be a structure, in which case `f1` is called. This last case statement tests the query structure's functor with the program's. Again, a match calls the clause code, and no matches returns failure.

The last two subsequences, `s2` and `s3`, cannot benefit from indexing since they only contain one clause. Again, this is the case since S_2 is has a variable key, and S_3 has no non-variable key neighbors. These subsequences simply contains the normal unindexed code for their clauses.

The body of the entire function `parent2` consists of a `try_me_else` of `s1()`, a `retry_me_else` of `s2()`, and finally a `trust_me` of `s3()`. These three functions call each of the three subsequence codes in order.

4.2. Last Call Optimization

One problem with recursion is that it requires space linear in the number of recursive calls, compared to constant space which is required by an iterative method. If the last call of a procedure is recursive, however, it is possible to transform it into a iterative form which is logically equivalent. The WAM refers to this as *last call optimization* (LCO), and explains how to implement an optimization which can be applied systematically with or without recursion [1].

LCO is possible since *permanent variables* allocated to a rule are no longer needed after the last instruction preceding the last function call made in a procedure is passed. Therefore, the current environment can be discarded before this last call is made. It introduces special instructions which are specifically designed to be used to do LCO.

Adding LCO to our BLAM was a much easier task. The SML compiler already has tail-call optimization built in, which means that we did not have to explicitly implement it ourselves. In order for SML to do this optimization, the SML function simply has to be in the tail-recursive form, which is having the recursive call be the last call of the function. It was our job then simply to make sure that this was the case wherever possible in our runtime system, and in the SML code which our compiler generated.

We found that an important feature in our design was preventing our generated code to receive this optimization. *Cuts*, one of the first extensions made to our core system changed the way Prolog predicates were called in SML. Implementing *cuts* involved requiring functions to leave a mark which would allow control to return to the function. We had defined the WAM runtime function `call` which added an ML exception handler to the heap as it made function calls. A second runtime function, `cut`, was also defined which could raise an exception and pass control back to the function. For a Prolog program to take advantage of LCO, call must be written in a way so that the SML compiler can recognize and optimize tail recursion. Since `call` is used to call every Prolog predicate, the problem with this method is that the exception added to the runtime stack prevents this optimization from occurring.

The way we initially defined `call` and `cut` are as follows:

```
fun cut ctag sc = raise ctag(sc)
```

```
fun call prog sc =
  let exception ctag of (unit -> unit)
   in
       prog ctag sc
       handle ctag (sc') => sc' ()
  end
```

Our solution to this problem involved using two special SML instructions which are not standard, callcc and throw [5]. It can be seen in the following code that the functionality of both call and cut remain the same, but the cut is now handled with continuations rather than exceptions. The advantage is that there is nothing added to the runtime stack during each call, which now allows SML to make tail-recursive optimizations when they are possible.

```
fun cut ctag sc = ( sc () ; throw ctag () )

fun call prog sc = callcc (fn ctag => prog ctag sc)
```

An example of a Prolog program which benefits from this optimization is *append*. One of the clauses in this predicate calls itself recursively. In the SML code generated for this clause, shown below, we can see that the SML function append3 is the last call made. Since it is an argument to call, it is important that call does not add anything to the heap, and allows the SML compiler to optimize this tail recursive function.

```
(*
 *  append(cons(X,L),K,cons(X,M)) :- append(L,K,M).
 *)
fun append3 (x1, x2, x3) ctag sc =
    let fun s1() =
        get_structure ("cons", 2, x1)
        (fn (x4::x5::nil, mode) =>get_structure ("cons", 2, x3)
         (fn (x6::x7::nil, mode) =>unify_value (x4, x6, mode)
          (fn () => (call (append3 (x5, x2, x7))
                      (fn (_) => sc ())))))
    in
        s1()
    end
```

4.3. Using SML Tools to Reduce Garbage Collection

Both of the previous are well known methods of improving the performance of WAM based systems. In addition to these types of optimizations, we were also concerned with finding out if there were bottlenecks specific to our particular SML implementation which could be improved upon. ML offers useful tools which make this type of analysis quite easy.

One such tool is the ML timer. `System.Timer.start_timer()` starts an internal clock running, and `System.Timer.check_timer` returns the total running time of an ML program since the timer was started, including how much of that time was spent garbage-collecting.

It is useful to know this information. For example, we found that one of our benchmarks, *hanoi.pl*, was spending 75% of its time doing garbage collection. Once this was brought to our attention, we concentrated on finding out what was causing this to happen.

In general, it is known that using SML references may cause a significant garbage collection overhead. We considered that the way we implemented the trail with two references was might be partially responsible for this. We experimented by restructuring the our trail structure in different ways. The most efficient structure we found was to have one reference to a 2-tuple. Reducing the number of references in this way improved our garbage collection down to 65%—an improvement of 10%.

The following data is the information which the timer returns.

```
(* running time of hanoi(10) before trail optimization: gc time = 75% *)
avg non-gc time = 0.765625
avg gc time = 2.25
avg total time = 3.015625

(* running time of hanoi(10) after trail optimization: gc time = 65% *)
avg non-gc time = 0.953125
avg gc time = 1.75
avg total time = 2.703125
```

Another useful tool which ML has available is *profiling*. By compiling SML programs with in a special profiling mode, data regarding the numbers of function calls and their execution time can be determined. From this information, we found out that the function `get_structure` was a time time consuming function. This information encouraged us to make the optimization described in the next section. Some raw profiling data from our final system will be presented in Section 6.

4.4. String Compare Optimization

During unification, structure terms need to be compared for equality. In addition to its arguments, a structure's functor (consisting of name and arity) also needs to be compared. Our initial design simply used strings to implement functor names. The problem with this is that string comparisons can be expensive; for instance, two names which are long and similar may require many integer (character) comparisons before it can be determined that the two strings are, in fact, different.

On the other hand, integer comparisons are resolved in a single comparison. Therefore, we replaced each string with a corresponding unique integer. Structure names, now represented by these integers, could now be compared in a minimum amount of time. Saving the mapping of strings to integers allowed translation from one to the other. Translating strings into integers is used to generate the SML code for programs and queries. Translating from integers back into strings is necessary to print out results, in which case the user would need to see the structure names in the string form in which they were entered.

The two functions in our runtime system which require name comparisons are unify and get_structure. From the following example, it can be seen how the strings get replaced by integers in the arguments of get_structure and in the indexing case statements. This first piece of code is the SML code generated without the optimization for a small Prolog program containing four clauses. Note that indexing is performed on both.

```
(*
 * tall(jim).
 * tall(joe).
 * tall(brother(jim)).
 * tall(sister(joe)).
 *)
fun tall1 (x1) ctag sc =
    let fun  s1_1()= try_me_else
        (fn (_) => s1_1a())
        (fn (_) => s1_2())
        and s1_1a() = get_structure ("jim", 0, x1)
            (fn (nil, mode) =>sc ())
        and s1_2() = retry_me_else
            (fn (_) => s1_2a())
            (fn (_) => s1_3())
        and s1_2a() =
            get_structure ("joe", 0, x1)
            (fn (nil, mode) =>sc ())
        and s1_3() = retry_me_else
            (fn (_) => s1_3a())
            (fn (_) => s1_4())
        and s1_3a() =
            get_structure ("brother", 1, x1)
            (fn (x2::nil, mode) =>get_structure ("jim", 0, x2)
             (fn (nil, mode) =>sc ()))
        and s1_4() = trust_me
            (fn(_) => s1_4a())
        and s1_4a() =get_structure ("sister", 1, x1)
            (fn (x2::nil, mode) =>get_structure ("joe", 0, x2)
             (fn (nil, mode) =>sc ()))
        and c1(name) = case (name) of
            "jim" => s1_1a ()
          | "joe" => s1_2a ()
          | (_) => ()
        and f1(name,arity) = case (name,arity) of
            ("brother", 1) => s1_3a ()
          | ("sister", 1) => s1_4a ()
          | (_) => ()
    in
        s1() = switch_on_term(x1,s1_1,c1,f1)
    end
```

This next piece of code shows the how the same program is compiled into slightly different SML code. The only changes is that the strings have all been replaced by integers. Comments have been added in order to more easily follow these changes.

```
fun tall1 (x1) ctag sc =
    let fun  s1_1()= try_me_else
        (fn (_) => s1_1a())
        (fn (_) => s1_2())
        and s1_1a() = get_structure (18, 0, x1)              (* jim => 18 *)
            (fn (nil, mode) =>sc ())
        and s1_2() = retry_me_else
            (fn (_) => s1_2a())
            (fn (_) => s1_3())
        and s1_2a() =
            get_structure (19, 0, x1)                        (* joe => 19 *)
            (fn (nil, mode) =>sc ())
        and s1_3() = retry_me_else
            (fn (_) => s1_3a())
            (fn (_) => s1_4())
        and s1_3a() =
            get_structure (20, 1, x1)                        (* brother => 20 *)
            (fn (x2::nil, mode) =>get_structure (18, 0, x2) (* jim again *)
             (fn (nil, mode) =>sc ()))
        and s1_4() = trust_me
            (fn(_) => s1_4a())
        and s1_4a() =get_structure (21, 1, x1)               (* sister => 21 *)
            (fn (x2::nil, mode) =>get_structure (19, 0, x2) (* joe again *)
             (fn (nil, mode) =>sc ()))
        and c1(name) = case (name) of
            18 => s1_1a ()                                   (* jim *)
          | 19 => s1_2a ()                                   (* joe *)
          | (_) => ()
        and f1(name,arity) = case (name,arity) of
            (20, 1) => s1_3a ()                              (* brother *)
          | (21, 1) => s1_4a ()                              (* sister *)
          | (_) => ()
    in
        s1() = switch_on_term(x1,s1_1,c1,f1)
    end
```

The strings *jim*, *joe*, *brother*, and *sister* get replaced by the integers *18*, *19*, *20*, and *21*, respectively. Notice how all of the strings are now gone, which allows the comparisons to take less time. Also note how instances of the same string get replaced by the same integer, which retains the same logic values.

5. Results and Analysis

We tested our system with a widely used SICStus Prolog system [4]. It is a good reference for comparison since it is written in a low-level language and is highly optimized. The SICstus code was interpreted byte code, also based on the WAM. We used Version 75 of the Standard ML of New Jersey compiler. All tests were performed on a DECstation 5000/200 with 96 megabytes of memory.

We chose benchmark programs which truly reflect the type of code Prolog programmers write, so we chose popular, well known programs. Our test suite includes five Prolog programs.

- *analogy.pl*: an "A is to B as C is to what?" type of puzzle
- *hanoi.pl*: the popular puzzle involving moving rings between poles according to a set of rules
- *nrev.pl*: a naive algorithm reverses the elements in a list
- *slowsort.pl*: brute force sort permutes all combinations of a list and checks if it is sorted
- *zebra.pl*: given a set of people, attributes, and rules, determine which person has which attributes

Prolog Program	Execution Time (ms)					
	SICStus	BLAM	Indexed	LCO	Garbage	String
analogy.pl	52	64	36	54	39	31
hanoi.pl	172	7619	3465	2925	2787	2543
nrev.pl	20	401	320	286	273	269
slowsort.pl	7781	—	29207	26204	27911	27922
zebra.pl	697	2781	2425	2496	2449	2181

Figure 2: Benchmark suite results

These five programs, seen in the first column of Figure 2, test all of the common features of Prolog. In particular, they involve all of the runtime functions, such as unification and backtracking. In order to evaluate the performance of our system, we first ran each of these each of these programs using SICStus. The execution time from these tests are seen in the second column of Figure 2. We then ran each benchmark using each version of our system, starting with the original BLAM and working our way up to the *string compare* optimization. This allows us to see the progress made each step of the way, as seen in the third through seventh columns.

There are a couple of interesting things worth pointing out. First of all, recall that arithmetic was not part of the original BLAM, so *slowsort.pl* could only be tested once it was added. Also, notice how the Garbage Collection optimization actually produces a slightly inferior performance for *slowsort.pl*. Our method of making a reference to a 2-tuple improves the other benchmarks, but not this one. Apparently, the extra overhead in creating and destructuring tuples outweighs the advantage of eliminating one SML reference.

In order to interpret these results better, Figure 3 shows some minor calculations performed on the raw data. One interesting calculation is how many times slower our system is compared with SICStus. The answer is in the second and third columns of Figure 3, containing these values for

our original system (BLAM) and on our final system (String). They are computed by taking the execution time of our system and dividing it by the execution time of SICStus. These figures tell us how much slower our system was originally, and where we currently stand. The fourth column contains a last calculation, which reveals how much each benchmark improved due to optimizations done to the WAM. These percentages were calculated by taking our original times (BLAM) and dividing them by our final times (String).

Prolog Program	Times Slower Than SICStus		Overall Speedup of the BLAM
	Original(BLAM)	Final(String)	
analogy.pl	1.2	.60	200%
hanoi.pl	44.3	14.8	300%
nrev.pl	20.1	13.45	150%
slowsort.pl	3.8	3.6	100%
zebra.pl	4.0	3.1	130%

Figure 3: Analysis of Results

This last column in Figure 3 reveals that *hanoi.pl* improved the most. This is due to the significant reduction in garbage creation, which significantly reduced collection time. On average, the the benchmarks nearly doubled in speed. This is encouraging, since there are still many of optimizations which can be done.

The third column of numbers tells us that a couple of benchmarks ran on our final system are still fifteen times slower SICStus. To find out where *hanoi.pl* and *nrev.pl* are spending time, we can again look at profiling information. Figure 4 contains the first few profiling entries for *hanoi.pl*. The first column of this chart tells us the most important information, the percentage of the program tied up in one function. Other information is also included, such as the number of times a function is called. From this figure, we can see that garbage collection still tops the list. Although this has already been partially optimized, it still appears to be a good target for further improvement.

The first program in this chart, *analogy.pl*, produces the best results, but is not as accurate of a test as the other benchmarks. It is short and there is much variation in its execution time. However, the good results it produces means that it does do well at pattern matching, the feature it tests the most.

The two programs which are only about three times slower than SICStus are *slowsort.pl* and *zebra.pl*. Although garbage collection is still a main problem in these cases, other functions are

%time	cumsecs	#call	ms/call	name
43.18	14.58	0		(gc)
24.46	22.84	0		(unprofiled)
9.47	26.04	7	114.2857	anon.Runtime.reset_trail
8.94	29.06	0		(toplevel)
1.89	29.70	162594	.0009	anon.Runtime.get_structure.anon
1.12	30.08	278904	.0003	anon.Runtime.deref
1.12	30.46	148524	.0006	anon.Runtime.bind
⋮	⋮	⋮	⋮	⋮

Figure 4: Profiling information for *hanoi.pl*

%time	cumsecs	#call	ms/call	name
17.24	15.38	0		(gc)
10.72	24.94	3253842	.0007	anon.Runtime.get_structure.anon
6.88	31.08	8041140	.0001	anon.Runtime.deref
4.86	35.42	1764672	.0006	anon.Runtime.unwind_trail.unwind
4.66	39.58	1764930	.0005	anon.Runtime.bind
4.17	43.30	1194225	.0007	anon.Runtime.try_me_else.anon
3.45	46.38	2093697	.0003	anon.Runtime.get_structure.anon.for
2.91	48.98	1402086	.0004	anon.Runtime.call.anon.anon
⋮	⋮	⋮	⋮	⋮

Figure 5: Profiling information for *slowsort.pl*

also responsible for a significant portion of the running time. In both of these programs, common runtime functions use a large portion of the time. The profiling information for *slowsort.pl* can be in Figure 5.

6. Conclusions

In Section 5 we witnessed speedups in BLAM ranging from 100% to 300%. This is very encouraging. Since each new optimization added continued to increase the system's performance, it is likely that many more refinements and changes could continue doing so. Even so, the current version of the BLAM is still one to fifteen times slower than SICStus. Although this is significant, it is not discouraging. SICStus is a highly optimized compiler which has been worked on for many years by various researchers. In comparison, our system was developed in two years of part time work by two relatively inexperienced undergraduates. Again, this implies that continued work will produce better performance.

There are three specific areas which can be further improved. First of all, there are the standard Prolog optimizations such as indexing. Many more are also known, such as Register Allocation and Environment Trimming. Second, there are more SML optimizations, like the one we implemented to eliminate string comparisons. Another one to try might be eliminating tag checking. Lastly, there may be more inefficiencies unique to our implementation other than the garbage collection we reduced. Profiling will help find such bottlenecks if any more exist.

Overall I think the results we found are quite optimistic. Although more work would be needed to be more competitive, further refinements to our system would achieve a performance even closer to SICStus or other low-level implementations.

Acknowledgments

I would like to thank Peter Lee and Frank Pfenning for all the time and effort they contributed to this project.

References

[1] Aït-Kaci, H. *The WAM: A (Real) Tutorial.* Technical Report 5, DEC Paris Research Laboratory, January 1990.

[2] Appel, A. and Jim, T. Continuation-passing, closure-passing style. *Proceedings of the Sixteenth Annual ACM 19 Symposium on Principles of Programming Languages*, Austin, Texas, 293-302, January 1989.

[3] Carlsson, M. On Implementing Prolog in Functional Programming. *New Generation Computing* **2**(4), pages 347–359, 1984.

[4] Carlsson, M. and Widen, J. *SICStus Prolog User's Manuel.* Swedish Institute of Computer Science, Kista, Sweden, 1988.

[5] Duba, B., Harper, R., and MacQueen, D. Typing first-class continuations in ML. *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Jan 1991.

[6] Elliott, C., and Pfenning, F. A Semi-Functional Implementation of a Higher-Order Logic Programming Language. *Topics in Advanced Language Implementation*, MIT Press, 1991.

[7] Kowalski, R. A. Algorithm = Logic + Control. *Communications of the ACM* **22**(7), pages 424–436, 1979.

[8] Milner, R., Tofte, M. and Harper, R. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[9] Moura, B. *Compiling Prolog to Standard ML: Extensions.* Technical Report, Carnegie Mellon University, Pittsburgh, PA, 1992.

[10] Sterling, L. and Shapiro, E. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[11] Warr· D. H. D. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Menlo Park, California, October 1983.