The implementation of Mercury, an efficient purely declarative logic programming language

Zoltan Somogyi, Fergus James Henderson and Thomas Charles Conway
{zs,fjh,conway}@cs.mu.OZ.AU
Fax: +61 3 348 1184, Phone: +61 3 282 2401
Department of Computer Science, University of Melbourne
Parkville, 3052 Victoria, Australia

Abstract

We introduce *Mercury*, a new purely declarative logic programming language designed to provide the support that groups of application programmers need when building large programs. Mercury's strong type, mode and determinism systems improve program reliability by catching many errors at compile time. We present a new and relatively simple execution model that takes advantage of the information these systems provide to generate very efficient code. We are developing a prototype compiler which uses this execution model to generate portable C code. Our benchmarks show that the code generated by our experimental implementation is significantly faster than the code generated by mature optimizing implementations of other logic programming languages.

1 Introduction

Logic programming languages are theoretically superior to imperative programming languages such as Pascal, C, C++ and Ada because they operate on a higher level. They are *declarative*, which means they allow the programmer to state *what* is to be done while leaving the details of *how* it is to be done to the language implementation. However, application programmers demand much more from a language than merely being declarative.

Most applications are written by groups of programmers. Languages should therefore support cooperation between programmers; the most effective method to date is to provide a module system that separates interfaces from implementations. This way, changes in one part of the program do not propagate unnecessarily to other parts of the program.

Application programmers want as much help as possible from the compiler in locating errors in their programs. This requires a programming language with redundant information; type declarations have proven useful in this role. This redundant information often turns out to be useful documentation; since the compiler's checks make sure that declarations are accurate, they are much more useful than comments.

Another requirement on the programming language is that its implementation produce reasonably fast and space efficient programs. Customers do not like slow programs.

Our objective is the creation of a logic programming language that meets the requirements of application programmers, i.e. one that has a good module system, detects a large fraction of program errors at compile time, and has an efficient implementation on many platforms.

Current logic programming languages do not meet the requirements of application programmers. Only a few Prolog dialects support modules, and their module systems interact badly with the rest of the language. Many Prolog implementations perform no semantic checks at compile-time, and most of the ones that do confine themselves to simple tests (e.g. detecting variables that occur only once in a clause). The absence of declarations makes it difficult for compilers to gather the information required by the optimizations needed to achieve competitive performance. Prolog programmers therefore often resort to non-logical constructs, which then destroy the declarative semantics of the program.

The lack of modules and compiler-checked declarations in most Prolog dialects tends not to be a problem for small programs, since a single programmer can understand all of the code at once and since such small programs can be debugged using Prolog's limited debugging facilities. However, as the size of the program increases and more people are added to the project team, the benefits of the declarative nature of logic programming are quickly outweighed by the effects of these limitations.

Our new logic programming language, *Mercury*, represents a clean break with previous logic programming tradition. It is necessarily incompatible with existing logic programming languages such as Prolog, since unlike these languages, Mercury has no non-logical constructs that could destroy the declarative semantics that gives logic programs their power. At the same time, it is explicitly designed to support teams of programmers working on large programs. For example, it has a modern module system that separates interfaces from implementations.

Mercury's polymorphic type system, modelled after ML's [9], is very expressive. The type system detects a large fraction of program errors. It is also the basis for a novel strong mode system that lets the compiler prevent errors such as floundering. The mode system is in turn the basis for a strong determinism analysis system, which catches even more errors. One of the most frustrating experiences of a Prolog programmer occurs when a large computation that was intended to succeed fails instead, since the bug could be anywhere in the computation; Mercury's determinism system can point out the bug at compile time. The support provided by these systems goes a long way towards ensuring that Mercury programmers spend less

time tracking down errors via tedious manual debugging than programmers working in other languages, logical or imperative.

We are currently working on a Mercury compiler. We are writing the compiler in Mercury itself, using NU-Prolog for boot-strapping until the compiler is complete. After eleven months and about one and a half person-year of effort, much of the compiler is already operational. The module system has proven that it can do its job of letting programmers cooperate without stepping on each others' toes. The compiler's type, mode and determinism checkers have together caught and hence prevented hundreds of errors in the compiler itself.

The compiler uses the information provided by the type, mode, and determinism systems to generate better code. We have tested the performance of the code generated by the compiler on several standard logic programming benchmarks. We have found it to have much higher performance than all the logic programming implementations we have access to, including Sicstus Prolog and Aquarius Prolog. The fastest of these systems achieve their speed by directly generating native code. Since significant effort is required to retarget them for different architectures, they are available on only a few platforms. Our Mercury compiler generates C code that can be compiled on almost all software and hardware platforms, including Unix systems, Macintoshes and PCs.

The structure of the paper is as follows. Section 2 presents the key components of the Mercury language, the type, mode, determinism and module systems. Section 3 introduces and explains our algorithms for executing deterministic, nondeterministic and semideterministic predicates and for handling negation and if-then-else. Section 4 compares our execution model with the standard logic programming implementation model, the Warren Abstract Machine or WAM. Section 5 discusses some optimizations, while section 6 presents performance results.

2 The Mercury language

Syntactically, Mercury is similar to Prolog with additional declarations. Semantically, however, it is very different. Mercury is a pure logic programming language with a well-defined declarative semantics. Like Gödel [7], Mercury provides declarative replacements for Prolog's non-logical features. Unlike Gödel, Mercury provides replacements for *all* such features, including I/O.

Mercury will appeal to at least two groups of programmers. One class is those with backgrounds in imperative languages such as C who are looking for a higher level and more expressive language. The other class is those with backgrounds in logic programming languages such as Prolog who are looking for a genuinely declarative language that supports the creation of efficient and reliable software solutions to large and complex problems.

Space limitations prevent us from describing the whole language; we give a brief overview which provides the background required for the rest of the paper.

2.1 Types

Mercury's type system is based on a polymorphic many-sorted logic. It is essentially equivalent to the Mycroft-O'Keefe type system [11], and to the type system of Gödel [7]. We borrow our syntax from the NU-Prolog [20] type checkers.

The basic method of defining types is with declarations such as

```
:- type bool ---> true ; false.
:- type list(T) ---> [] ; [T | list(T)].
```

Each of these declarations introduces a new type and lists the one or more (in these cases two) function symbols that can be used to construct terms of that type. (We use the standard logic programming notation for lists. `[]` represents nil, the empty list, and `[H | T]` represents a list with head H and tail T.) The arguments of the function symbols are types. These types can be defined anywhere in the program; we allow forward references as well as (mutually) recursive types. We support parametric polymorphism: type declarations may contain references to type variables (e.g. T is a type variable in list(T) above). To ensure that no run-time type checking is needed, we require that all type parameters occurring on the right hand side of a type definition also occur on the left hand side.

In principle all types can be defined just like this, but in practice types such as `int` must be built into the system.

Since we need a strong type system as a foundation for our strong mode system, the compiler must be able to determine the type of every variable. We require that the types of the arguments of every predicate in the program be declared, like this:

```
:- pred permutation(list(T), list(T)).
```

The compiler then automatically infers the types of all variables. It would be possible for the compiler to also infer the type signatures of predicates, as ML does for functions. However, the absence of explicit predicate type declarations would make programs significantly harder to read and maintain, and this conflicts with our objective of making Mercury a language suitable for developing large programs.

A program and query are *type correct* if there is a unique most general assignment of parametric polymorphic types to the variables and function symbols occurring in the program and the query such that the type of every argument of every atom in the program and the query (including those in clause heads) is identical to the declared type of the corresponding formal argument.

In the rest of the paper, we assume that the program and the query are type correct, and that the type of all symbols is known. Type systems of this nature have been the subject of much research, so we omit a description of the type checking algorithm and refer the reader to

papers describing type checking algorithms for equivalent and related type systems (see e.g. [11, 12]).

2.2 Modes

We consider the *mode* of a predicate as a mapping from the initial state of instantiation of the arguments of the predicate to their final state of instantiation. To describe states of instantiation, we use information provided by the type system. Types can be viewed as regular trees with two kinds of nodes: or-nodes representing types and and-nodes representing function symbols. The children of an or-node are the function symbols that can be used to construct terms of that type; the children of an and-node are the types of the arguments of the function symbol. Following [17], we attach mode information to the or-nodes of type trees.

An *instantiatedness tree* is an assignment of an *instantiatedness* — either *free* or *bound* — to each or-node of a type tree, with the constraint that all descendants of a free node must be free.

A term is *approximated by* an instantiatedness tree if for every node in the instantiatedness tree,

- if the node is "free", then the corresponding node in the term (if any) is a free variable that does not share with any other variable (we call such variables *distinct*);
- if the node is "bound", then the corresponding node in the term (if any) is a function symbol.

When an instantiatedness tree tells us that a variable is bound, there may be several alternative function symbols to which it could be bound. The instantiatedness tree does not tell us which of these it is bound to; instead for each possible function symbol it tells us exactly which arguments of the function symbol will be free and which will be bound. The same principle applies recursively to these bound arguments.

Our mode system allows users to declare names for instantiatedness trees using declarations such as

```
:- inst listskel ---> bound(
      [] ; [free | listskel]).
```

This instantiatedness tree describes lists whose skeleton is known but whose elements are distinct variables. As such, it approximates the term `[A,B]` but not the term `[H|T]` (only part of the skeleton is known), the term `[A,2]` (not all elements are variables), or the term `[A,A]` (the elements are not distinct variables).

As a shorthand, our mode system provides `free` and `ground` as names for instantiatedness trees all of whose nodes are free and bound respectively. The shape of these trees is determined by the type of the variable to which they apply.

As execution proceeds, variables may become more instantiated. A *mode mapping* is a mapping from an initial instantiatedness tree to a final instantiatedness tree, with the constraint that no node of the type tree is transformed from bound to free. Our language allows the user to specify mode mappings directly by expressions such as `inst1 -> inst2`, or to give them a name using declarations such as

```
:- mode m :: inst1 -> inst2.
```

We provide two standard shorthand modes corresponding to the standard notions of inputs and outputs:

```
:- mode in :: ground -> ground.
:- mode out :: free -> ground.
```

These two modes are enough for the vast majority of predicates [4]. Nevertheless, Mercury's mode system is sufficiently expressive to handle more complex data-flow patterns, including those involving partially instantiated data structures. For example, consider an interface to a database that associates data with keys, and provides read and write access to the items it stores. To represent accesses to the database over a network, we will need declarations such as

```
:- type operation --->
      lookup(key, data) ; set(key, data).
:- inst request ---> bound(
      lookup(ground, free) ; set(ground, ground)).
:- mode create_request :: free -> request.
:- mode satisfy_request :: request -> ground.
```

A *predicate mode declaration* assigns a mode mapping to each argument of a predicate. Given the mode names defined by

```
:- mode out_listskel :: free -> listskel.
:- mode in_listskel :: listskel -> listskel.
```

the (type and) mode declarations of the predicates length and append are as follows:

```
:- pred length(list(T), int).
:- mode length(in_listskel, out).
:- mode length(out_listskel, in).

:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

A predicate mode declaration is an assertion by the programmer that for all possible argument terms for the predicate that are approximated by the initial instantiatedness trees of the mode declaration and all of whose free variables are distinct, if the predicate succeeds then the resulting binding of those argument terms will in turn be approximated by the final instantiatedness trees of the mode declaration. We refer to such assertions as *mode declaration constraints*. These assertions are checked by the compiler, which rejects programs if it cannot prove that their mode declaration constraints are satisfied.

Note that for every mode of a predicate in which a node is *produced* (mapped from free to bound) there is another mode for that predicate in which the node is *consumed*

(mapped from bound to bound), and for every mode in which a node is ignored (mapped from free to free) there is another mode in which the node is mapped from bound to bound. Such modes are called *implied modes*.

The *mode set* for a predicate is the set of mode declarations for the predicate plus all their implied modes. A mode set is an assertion by the programmer that the predicate should only be called with argument terms that are approximated by the initial instantiatedness trees of one of the mode declarations in the set (i.e. the specified modes and the modes they imply are the only allowed modes for this predicate). We refer to the assertion associated with a mode set as the *mode set constraint*; these are also checked by the compiler.

Now we come to defining *well-modedness*. We want to reject programs that violate either the mode declaration constraints or the mode set constraints. In general, this is undecidable. We want a definition of well-modedness that is easy for the compiler to check, and easy for programmers to understand. Rather than requiring global analysis of the entire program, determining if a predicate is well-moded should require only local analysis using the clauses of the predicate and the mode declarations for any called predicates. This prompts the following definition.

A predicate p is *well-moded with respect to a given mode declaration* if given that the predicates called by p all satisfy their mode declaration constraints, there exists an ordering of the literals in each clause of p such that

- p satisfies its mode declaration constraint, and
- p satisfies the mode set constraint of all of the predicates it calls

We say that a predicate is well-moded if it is well-moded with respect to all the mode declarations in its mode set, and we say that a program is well-moded if all its predicates are well-moded.

The mode analysis algorithm works with clauses in *superhomogeneous form*, where each atom is of one of the forms

```
p(X1, ..., Xn)
Y = X
Y = f(X1, ..., Xn)
```

Any clause can be converted to superhomogeneous form by replacing with distinct variables the arguments in the head and in calls in the body, generating explicit unifications for these variables in the body, and then breaking complex unifications down into several simpler ones.

The mode analysis algorithm checks one mode of one predicate at a time. It abstractly interprets each clause of the predicate, keeping track of the instantiatedness of each variable, and selecting a mode for each call and unification in the clause body. To ensure that the mode set constraints of called predicates are satisfied, it may reorder the literals in the clause; it reports an error if no satisfactory order exists. Finally it checks that the resulting instantiatedness of the predicate's arguments matches that given by the predicate's mode declaration. For further details we refer the reader to [6, 16, 17].

The mode analysis algorithm annotates each call with the mode used. This is necessary because our implementation generates separate code for each mode of each predicate, and therefore the code generated for a call depends on the mode in which the predicate is being called. We generate inline code for some built-ins such as the arithmetic predicates and for certain instances of unification. These instances are:

- instances of `Y = X` where one of X and Y is input, i.e. has the mode `ground -> ground`, while the other is output, i.e. has the mode `free -> ground`; in the rest of the paper we indicate this as e.g. `Y := X`.
- instances of `Y = X` where both X and Y are input and of atomic type; in the rest of the paper we indicate this as e.g. `Y == X`.
- instances of `Y = f(X1, ..., Xn)` where Y is output and the X_is are either input or void (i.e. this is their only appearance in the clause); we indicate this as `Y := f(X1, ..., Xn)`.
- instances of `Y = f(X1, ..., Xn)` where Y is input and the X_i are output or void; we indicate this as `Y == f(X1, ..., Xn)`.

For other instances of unification we call an out-of-line procedure whose code is derived from the type of the arguments. For example, to test whether two lists are equal we call a predicate such as

```
unify_list_t(A, B) :- A = [], B = [].
unify_list_t(A, B) :- A = [AH|AT], B = [BH|BT],
   unify_t(AH, BH), unify_list_t(AT, BT).
```

The predicate we call to unify AH and BH depends on their type, which is the type of the elements of A and B. We can implement polymorphic predicates such as lookups on association lists either by generating separate code for each type they are used with or by generating one piece of code that takes the address of a unification routine as a hidden parameter. The code for the second approach would call the procedure identified by this hidden parameter instead of `unify_t`.

We can generalize this idea to the notion of polymorphic modes. Other generalizations include the notions of circular modes and of mode segments. As these are beyond the scope of this paper, we refer the interested reader to [6, 16, 17].

2.3 Determinism

For each mode of a predicate, we categorize that mode according to the number of solutions that can be returned by calls to the predicate in that mode:

- If all calls which match a particular mode of a predicate have exactly one solution, then that mode of the predicate is *deterministic*.

- If all calls which match a particular mode of a predicate either have no solutions or have one solution, then that mode of the predicate is *semideterministic*.
- If a mode of a predicate is neither deterministic nor semideterministic, then it is *nondeterministic*.

Whenever the programmer declares a mode for a predicate, Mercury requires that the declaration also state the determinism of that mode as either `det`, `semidet`, or `nondet`. For example, here are the declarations of the predicate sort/2.

```
:- pred sort(list(T), list(T)).
:- mode sort(in, out) is det.
:- mode sort(out, in) is nondet.
:- mode sort(in,  in) is semidet.
```

The compiler analyses the bodies of predicates to check that their declarations are correct. Although this problem is undecidable in general, several fast approximate solutions are known [3, 14, 15], and they work well in practice.

In fact the compiler needs determinism declarations only for exported predicates – it can (and does) infer the determinism of predicates local to a module. However, there is no advantage in making them optional, since determinism declarations are not only very easy to write, they provide important documentation and help the compiler to pinpoint quite a significant number of program errors. These errors, e.g. the failure of a query that was intended to succeed, are very difficult to debug using conventional methods, since the failure may be *anywhere* in the call-tree of the query predicate, and this can be a very large amount of code.

2.4 Modules

The mercury module system is simple and straightforward. Each module must start with a `begin_module` declaration, specifying the name of the module. An `interface` declaration specifies the start of the module's interface section: this section contains declarations for the types, function symbols, instantiation states, modes, and predicates exported by this module. Mercury provides support for abstract data types, since the definition of a type may be kept hidden, with only the type name being exported. An `implementation` declaration specifies the start of the module's implementation section. Any entities declared in this section are local to the module and cannot be used by other modules. The implementation section must of course contain definitions for all abstract data types and predicates exported by the module, as well for all local types and predicates. The module may optionally end with an `end_module` declaration.

If a module wishes to make use of entities exported by other modules, then it must explicitly import those modules using one or more `import_module` declarations. These declarations may occur either in the interface or the implementation section.

For example, here is the definition of a "queue" module.

```
:- module queue.
:- interface.

% Declare an abstract data type.

:- type queue(T).

% Declare some predicates which
% operate on the abstract data type.

:- pred empty_queue(queue(T)).
:- mode empty_queue(out) is det.
:- mode empty_queue(in) is semidet.

:- pred put(queue(T), T, queue(T)).
:- mode put(in, in, out) is det.

:- pred get(queue(T), T, queue(T)).
:- mode get(in, out, out) is semidet.

:- implementation.

% Queues are implemented as lists.
% We need to import the 'list' module for
% the declarations of the type list(T),
% with its constructors '[]'/0 and '.'/2,
% and the predicate append/3.

:- import_module list.

% Provide a definition for the queue ADT.

:- type queue(T) == list(T).

% Definitions for the exported predicates.

empty_queue([]).

put(Queue0, Elem, Queue) :-
        append(Queue0, [Elem], Queue).

get([Elem | Queue], Elem, Queue).

:- end_module queue.
```

Mercury has a standard library which includes modules for lists, stacks, queues, priority queues, sets, bags (multi-sets), maps (dictionaries), random number generation, input/output, filename and directory handling and meta-programming.

3 The Mercury execution algorithm

We want our Mercury implementation to be fast yet portable. Our speed objective is incompatible with interpretation, yet we do not want to generate native code directly as this would tie us to a particular machine architecture. What we would like to do is generate code in a portable assembly language. The best such language

we know of is GNU C. It exists on many platforms, and it extends standard ANSI C with several features that let us access the underlying machine. The extensions we can exploit are

- the ability to take the address of a label using a statement such as `succip = &&label;` and later jump to that address using a statement such as `goto *succip;` [1]
- the ability to make direct use of the machine registers using global register variable declarations such as `register int r1 __asm__("s1");`

Although we can exploit these extensions, they are not necessary for our execution model, Depending on the options it is given, the Mercury compiler will exploit both, either or none of these features; when not using either GNU C feature, the code we generate is pure ANSI C.

When not exploiting GNU C's nonlocal gotos, we compile each labelled piece of code into a function, and we implement abstract machine gotos as return statements that tell a driver program which such function to call next. Since this hides the flow of control, all our examples will show GNU C code.

Most implementations of logic programming languages use a single algorithm to execute all predicates. They therefore pay overheads even when they are not required; one example is preparing for the failure of predicates that can never fail. On the other hand, the determinism analysis phase of the Mercury compiler classifies each mode of each predicate in the program as being either deterministic, semideterministic or nondeterministic. The code generator takes advantage of this fact and uses a specialized execution algorithm for each class. Sections 3.2 to 3.4 describe these algorithms. We start with a description of Mercury's approach to data representation.

3.1 Data representation

Since we know all types at compile time, we can and do specialize the representation of terms for each type. This reduces storage requirements somewhat and improves time efficiency considerably.

Our primary target machines are byte-addressed machines with 32-bit words. Many such machines require words to be aligned on natural boundaries; we store all our data in aligned words even on machines that do not have this requirement. Since the low-order two bits of pointers to aligned words are always zero, we can use these bits as a tag, giving us four different tag values. 64-bit machines would give us eight.

For types with up to four different alternatives, the two tag bits are sufficient to distinguish them; the remainder of the word is a pointer to a sequence of words on the heap containing the arguments of the function symbol, if any. For an example, consider a variable of type `list(int)` after the compiler allocates the tag value 0 to nil and the tag value 1 to cons (for readability, we will write these values as `TAG_NIL` and `TAG_CONS` in the rest of the paper). If the tag on the variable is zero, then the rest of the word is zero as well (no pointer needed). If the tag is one, then the remainder of the word points to a two-word block of memory on the heap, the first word being the head element of the list and the second word being the tail of the list.

Note that this representation is similar to the optimized list representation used by many Prolog systems. The difference is that in our scheme, it applies to *any* data type with less than five alternatives, not just to lists.

If a type has five or more alternatives, then some have to share the same two bit primary tag value. We share a tag only between several constants or several non-constants. If several constants share a two-bit primary tag value, we use the rest of the word as a 30-bit local secondary tag. If several non-constants share a two-bit primary tag value, we store an extra word at the start of the argument block as a 32-bit remote secondary tag.

This scheme can represent a billion constants and four billion non-constants using only two primary tag values, leaving two other values free for use by the most frequently occurring alternatives. One can think of 32-bit integers and floats as types in which all four primary tags values have fully used local secondary tags, although of course in practice the built-in operations operate on the entire word as a unit.

Relatively few types in real programs need secondary tags. For simplicity of presentation, we do not show any such types. We use the following C macros for manipulating primary tags and the words in which they occur:

```
#define mkword(t,p)  ((unsigned)(p) + (t))
#define tag(w)       ((w) & 0x3)
#define body(w,t)    ((w) - (t))
#define field(t,w,i) ((Word *) body(w,t))[i]
```

Mkword puts the tag t on the pointer p (which must be aligned). Tag extracts the tag of a word, while body extracts the non-tag part. Field yields a reference to the i'th word in the argument block pointed to by the word w with tag t. Note that if t and i are both constants, as they always are in the code we generate, neither the subtraction of the tag t nor the indexing with i in the field reference require separate instructions on most machines, as they are folded into the offset part of the load or store instruction.

Note that if a function symbol occurs in more than one type, it will be represented differently for each type. The type checker will determine the type of each occurrence

[1] According to the GNU C manual, "totally unpredictable things will happen" if computed gotos jump to code in a different function, as they do in the code we generate (when exploiting gcc's nonlocal gotos, we compile each Mercury module into one C function). However, considerable inspection of the gcc source code shows that we can use such jumps quite safely provided we take suitable precautions, the most important being (a) making sure that none of the functions involved has any local variables and (b) making sure that the compiler has access to a big enough stack frame to hold any spilled temporary values. Successful testing on several different architectures (including the i486, which is very short on registers) has confirmed our assessment.

of the function symbol. The different representations do not cause any problems, since terms of different types cannot be compared directly.

3.2 The deterministic execution algorithm

Deterministic predicates have exactly one solution and are hence essentially equivalent to imperative programs. Our aim is to implement such predicates as efficiently as imperative languages like C do.

The memory areas of our abstract machine for executing deterministic predicates are a code segment, a stack and a heap. The code segment is static; the stack and the heap grow upwards. The order of these areas in the address space does not matter. The abstract machine also has the following registers:

sp	The stack pointer: it holds the address of the next free word on the stack.
hp	The heap pointer: it holds the address of the next free word on the heap.
succip	The success instruction pointer: it holds the address of the label to return to when the current predicate succeeds.
r1, r2, ...	General purpose registers for parameter passing and for temporary values.
f1, f2, ...	Floating point registers for parameter passing and for temporary values.

The floating point registers are separate because many modern machines have a separate set of FP registers, and we want to map our abstract machine registers directly to the underlying hardware registers. However to simplify our presentation we do not discuss floating point in the rest of the paper.

To prepare to execute a query, the system sets the stack pointer and the heap pointer to point to the start of their respective data areas, sets the success IP register to point to code that will terminate execution of the abstract machine, sets up the registers to reflect the input arguments of the query, and jumps to the code of the query. (We will assume the query is a single atom; if it isn't, one can introduce a new "answer" predicate with one clause whose body is the query.)

As an example, consider the query

```
:- append([1,2], [3], L).
```

This calls append in its first (forward) mode, whose form after mode analysis is:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
append(A, B, C) :- A == [], C := B.
append(A, B, C) :- A == [H|T],
   append(T, B, NT), C := [H|NT].
```

In the absence of any floating-point arguments or optimization, our parameter passing convention is that

- if any part of argument i is input, it should be in register ri at call but may not be there at return
- if all parts of argument i are output, ri may be undefined at call but will be in register ri at return
- the called procedure is free to destroy rj for values of j greater than its arity

For our example, the system will put pointers to the lists [1,2] and [3] into registers r1 and r2 respectively before calling append, which is implemented by GNU C code like the following. (The actual code generated by the compiler differs from the following code in minor ways. Two examples are that it contains explicit manipulations of the stack and heap pointers and that it refers to labels via macros whose definition depends on whether we are exploiting GNU C's nonlocal gotos or not. None of these differences affect the substance of our presentation; we have chosen the notation below for clarity of exposition.)

```
/* version 1 of append_1 */
append_1:
   /* clause 1 */
append_1_a0:
   if (r1 != TAG_NIL)
      goto append_1_b0;
append_1_a1:
   r3 = r2;
append_1_a2:
   proceed();
   /* clause 2 */
append_1_b0:
   push(succip);
   push(field(TAG_CONS, r1, 0));
   r1 = field(TAG_CONS, r1, 1);
append_1_b1:
   call(append_1, append_1_b2);
append_1_b2:
   r4 = pop();
   r3 = mkword(TAG_CONS,
      create2(r4, r3));
append_1_b3:
   succip = pop();
   proceed();
```

Each label begins with the name of the predicate, followed by the number of the mode this abstract machine procedure implements (in this case mode 1 is the forward mode). For the label that starts an abstract machine procedure, this is all. Other names encode the position of the label within the internal form of the predicate: a, b, c etc indicate which clause we are in and 0, 1, 2 etc indicate how many literals precede this point in the clause. Some of these labels are not necessary since they are not referred to in the program; we added them to make discussing the code easier.

The code between points a0 and a1 implements the test `A == []`. In general, we have to strip off the non-tag bits of r1 before the comparison, but since we are comparing against a constant and this constant's primary tag value is not shared, we would be stripping only zeroes anyway.

(If the function symbol we are comparing against had a shared primary tag, the condition of the if statement would test the secondary tag after testing the primary tag.) The code between points a1 and a2 implements the assignment `C := B`. The proceed macro then branches to the success continuation, the label whose address was put into the succip register by the caller (i.e. when compiling for GNU C the macro proceed() is defined as goto *succip).

The code at b0 starts by saving the value of succip on the stack, since the succip register will be overwritten by the call to append in the body of the clause. The rest of the code between b0 and b1 implements `A == [H|T]`. It does not need to check that A has the cons tag, since execution gets to this point only if A has a tag other than nil, and cons is the only such tag in the type of A. Therefore this code has only to extract the values of H and T from the cons cell pointed to by A, a cons cell whose address is computed by the field macro by subtracting the value of the cons tag from the value in r1. The value of H is not needed immediately but it must be preserved across the recursive call. Since the call may destroy any register, H must be preserved by pushing it on the stack. The value of T does not need to be preserved across the call but must be put in r1 for the call. The other input argument of the call, B, is already in r2.

The call macro at b1 puts the address of the append_1_b2 label into the succip register and then branches to the append_1 label to begin execution of the recursive invocation of append. Execution will continue at append_1_b2 when the recursive invocation executes a proceed, at which time r3 will contain the value of NT. The create2 macro at b2 creates a two-word cell on the heap and fills the first word in this cell with the value of H popped off the stack and the second word with the value of NT in r3. It then returns the address of the cell. The mkword macro then puts the cons tag on the low-order two bits of the address and puts the result in r3, thus completing execution of `C := [H|NT]`. The rest of the code in the clause returns to the caller by popping the saved value of succip off the stack and branching to the label whose address it contains.

The code we have just discussed makes several assumptions which are safe because they are invariants of our execution model:

- Each called procedure leaves the stack pointer exactly as it found it. Therefore if a procedure leaves a value at a particular offset from sp, it knows it will still be there when a procedure it calls returns.
- The values in the registers holding the input arguments, r1 and r2, cannot be unbound variables. The mode analysis algorithm guarantees that consumers of a variable (or part of a variable) will be scheduled to execute after its producer.
- The values in r1 and r2 do not need to be dereferenced. The ordering of consumers after producers means that consumers can be passed the values of variables rather than pointers to them. Since the values are available when building structures, fields inside structures do not need dereferencing either.

3.3 The nondeterministic execution algorithm

The execution algorithm of the previous section does not work for nondeterministic predicates because the clauses of such predicates may be backtracked into. When that happens, continued execution requires the values of the variables occurring in the clause, but these values were popped off the stack when the procedure succeeded for the first time. We must therefore add to our execution model a new stack for nondeterministic predicates whose frames are popped not on success but on failure. Like our other dynamic areas, this *nondet stack* grows upwards and its placement in the address space does not matter. It is addressed by two new abstract machine registers:

curfr — Points to the nondet stack frame of the currently executing procedure.

maxfr — Points to the highest frame on the nondet stack.

Each frame of the nondet stack consists of four or more words containing

succip — The address of the label to return to when the current predicate succeeds.

redoip — The address of the label to return to when the current clause fails; points to the code for the next clause if there is one.

succfr — The curfr value to restore when the current predicate succeeds; points to the nondet stack frame of the caller if the caller is a nondeterministic predicate.

prevfr — The curfr value to restore when the current predicate fails; points to the immediately previous frame on the nondet stack.

framevars — Zero or more variables whose values have to be saved either between clauses (i.e. input arguments) or between calls in a clause (i.e. variables local to a clause).

A framevar slot will contain garbage before the execution of the producer of the variable it corresponds to; afterward it will contain the value of the variable. At no point will these slots contain unbound variables, and pointers will never point to these slots.

The framevar slots are at the lowest addresses within each frame while the curfr and maxfr registers and the pointers between frames all contain the address of the highest word within the frame they point to. This arrangement lets us avoid storing the number of saved values in each frame (it can be recalculated as frameaddr - prevfr - 4 words if necessary). The disadvantage is that it is impossible to add new saved value slots after the frame is set up.

As an example, consider the query `?- append(A, B, [1,2])`. This calls append in its second (backward) mode, whose form after mode analysis is:

9	append prevfr:	&nondetstack[3]
8	append succfr:	&nondetstack[3]
7	append succip:	&&ask_more
6	append redoip:	undefined
5	append var(0):	[1,2,3]
4	append var(1):	undefined
3	interp prevfr:	unused
2	interp succfr:	unused
1	interp succip:	unused
0	interp redoip:	&&no_more

Figure 1: Nondet stack near start

```
:- pred append(list(T), list(T), list(T)).
:- mode append(out, out, in) is nondet.
append(A, B, C) :- A := [], B := C.
append(A, B, C) :- C == [H|NT],
   append(T, B, NT), A := [H|T].
```

The corresponding C code is:

```
append_2:
   mkframe(2);
   framevar(0) = r3;
append_2_a0:
   /* clause 1 */
   modframe(&&append_2_b0)
   r1 = mkword(TAG_NIL, 0);
   r2 = r3;
   succeed();
append_2_b0:
   modframe(dofail);
   r1 = framevar(0);
   /* clause 2 */
   if (tag(r1) != TAG_CONS)
      redo();
   r2 = field(TAG_CONS, r1, 0);
   r3 = field(TAG_CONS, r1, 1);
   framevar(1) = r2;
   call(append_2, append_2_b2);
append_2_b2:
   r1 = mkword(TAG_CONS,
      create2(framevar(1), r1));
   succeed();
```

Following the pattern of all nondeterministic procedures, the code of append_2 starts with an invocation of the mkframe macro, which creates a new frame on the top of the nondet stack and sets both maxfr and curfr to point to the new frame. Mkframe's argument gives the number of framevar slots to reserve in the frame. Mkframe also fills in the succip slot (with the value put in the succip register by the call), the prevfr slot (with the previous value of maxfr), and the succfr slot (with the previous value of curfr). The call to mkframe is then followed by code to save each input register in the framevar slots of the current frame for use in later clauses. In this case the only input argument is in r3 (the calling convention is the same for deterministic and nondeterministic predicates).

Figure 1 shows the state of the nondet stack when execution reaches append_2_a0. Each box represents a word. The text on the left side says what the word is used for while the text on the right side says what the current value is (if it is meaningful). Values starting with "&&' point to labels; values starting with the normal address–of operator "&' are links within the nondet stack. The value "[1,2]" means a word whose tag is cons and whose body is a pointer to the rest of the list, which is on the heap. Both maxfr and curfr contain &nondetstack[9].

Note that the query interpreter is special because its lifetime is the lifetime of the entire process. It may be backtracked into, and thus needs its redoip slot, but it never fails and never succeeds, and so its prevfr, succfr and succip slots are unused.

The code for the first clause starts by setting the redoip slot of the current frame to point to the start of the next clause. It then sets r1 to [] and sets r2 to the value in r3. At the end of the clause (as at the end of every clause) we have to reset curfr to point to the frame of the calling procedure and branch to the continuation address within that procedure. The succeed macro carries out both actions, taking both addresses from their slots in the current frame. Since maxfr is not reset, the frame of this invocation of append remains on the stack, so the stack remains as in figure 1 with the exception that the redoip slot at offset 6 now contains &&append_2_b0. Maxfr stays at &nondetstack[9] but curfr becomes &nondetstack[3].

In this case, the call to append from the interpreter specified the label ask_more as the value of succip. The code at this label prints the solution (A = [], B = [1,2]) and asks if the user wants another solution. If no, it resets the stacks and the heap for the next query. If yes, it executes a *redo* action to find the next solution.

The Mercury execution model uses simple chronological backtracking, i.e. we always backtrack to the most recent choice. The redo macro therefore branches to the label whose address it finds in the redoip slot in top nondet frame and sets curfr to point to that frame to provide the proper environment for the code being branched to. In this case, curfr is set back to &nondetstack[9] and execution continues at append_2_b0.

The code for the second clause starts by updating the redoip slot via the modframe macro. Since we are entering the last clause, the new value of redoip will cause this call to fail if it is ever asked for more solutions. Dofail is a global variable that contains the address of a label. The code at this label executes the fail macro, which removes the topmost frame on the nondet stack (by setting maxfr to the value of the prevfr slot of the frame being removed), sets curfr to point to the newly exposed frame, and branches to the label whose address is in the redoip slot of that frame. In this case, the failure of append will expose the frame of the interpreter and cause a branch to the label no_more, which prints "no more solutions" and asks the user for the next query.

Before execution gets to that point, however, we must execute the second clause of append. The code starts by copying the saved value of the argument C from a

21	append3 prevfr:	&nondetstack[15]
20	append3 succfr:	&nondetstack[15]
19	append3 succip:	&&append_2_b2
18	append3 redoip:	&&append_2_b0
17	append3 var(0):	[]
16	append3 var(1):	undefined
15	append2 prevfr:	&nondetstack[9]
14	append2 succfr:	&nondetstack[9]
13	append2 succip:	&&append_2_b2
12	append2 redoip:	dofail
11	append2 var(0):	[2]
10	append2 var(1):	2
9	append1 prevfr:	&nondetstack[3]
8	append1 succfr:	&nondetstack[3]
7	append1 succip:	&&ask_more
6	append1 redoip:	dofail
5	append1 var(0):	[1,2]
4	append1 var(1):	1
3	interp prevfr:	unused
2	interp succfr:	unused
1	interp succip:	unused
0	interp redoip:	&&no_more

Figure 2: Nondet stack near end

framevar slot into a register (r1) for further manipulation. This register doesn't have to be (and in this case isn't) the same register as the one the argument was passed in originally; indeed the value does not need to be put into a register at all if the body of the particular clause does not need it. Here, however, we must test the value of C and possibly get H and NT out of the cons cell it points to, so it is more efficient to load it into a register.

If C's tag is not cons, i.e. C is nil, the redo macro causes a branch to the label whose address is in the redoip slot of the topmost frame. Since this is the topmost frame, and we just put dofail into the redoip slot, this will cause the current call to fail.

If C's tag is cons, we extract H and NT from the cell it points to. Since the value of H will be needed after the call but the value of NT will not be, we save the value of H in a framevar slot and put NT into r3, where it should be for the recursive call. In our example, the value of NT will be [2]. The recursive call may succeed several times. In this case it succeeds twice, the solutions being T = [], B = [2] and T = [2], B = []. When the recursive call returns one of these solutions, the code at append_2_b2 will put the value of H (which is 1) at the front of T and return the result as the value of A, together with the untouched value of B. Therefore the second clause succeeds twice with the solutions A = [1], B = [2] and A = [1,2], B = [].

Figure 2 shows the state of the nondet stack just after the last of these successes; different invocations of append are denoted by different suffixes. The usual redo from the interpreter sets curfr to &nondetstack[21] and causes a branch to append_2_b2. The code there sets redoip to dofail and tests the value of C saved in nondetstack[17] to see if it has a cons tag. It does not, so the if statement executes the redo, which now causes a branch to an invocation of the fail macro. This will remove the topmost frame by setting maxfr and curfr to &nondetstack[15] and branch through the redoip slot of the new topmost frame. All remaining frames corresponding to invocations of append have dofail in their redoip slots, so they are discarded one by one. The chain of failures stops in the interpreter's frame, whose redoip slot points to code to print "no more solutions" and ask the user for the next query.

Each solution after the first requires the creation of new cells on the heap. These cells are never accessed after the interpreter prints the solution for which they were created and executes a redo, because they were allocated between the last backtrack point and the site of the redo. The right time to recover the space of these cells is when restarting forward execution at a backtrack point. We can implement this by saving the value of the heap pointer register hp on entry to the first clause, and restoring this value on entry to the second and later clauses. (The space allocated in the last clause is recovered when we backtrack to the previous frame.) Beside the time taken by the saving and restoring code, the only cost is the extra framevar that stores the saved hp. This extra cost is usually more than repaid by the better cache and virtual memory performance that results from the more efficient use of memory.

3.4 The semideterministic execution algorithm

Semideterministic predicates cannot succeed more than once. Since they can never be backtracked into, their local variables can be stored in the det stack. This is faster than storing them on the nondet stack, since we never have to allocate space for and fill in the fixed slots of nondet stack frames.

There are two ways to implement failure in semidet code. One approach uses the mechanisms we introduced in the previous section. When a test unification fails in a semidet predicate, this approach calls for executing a redo to cause backtracking to the most recent point where an alternative action exists. This means that predicates which call semidet predicates must ensure the redoip slot in the top nondet stack frame points to the appropriate failure continuation. Such predicates must also be prepared to restore the det stack pointer to the value it had before the call. (The semidet predicate cannot restore the det stack pointer itself, since it does not create its own nondet stack frame, and therefore cannot gain control at the appropriate time.) Such restoration requires a nondet stack frame with a slot reserved to store the det stack pointer. This is why predicates that call a semidet predicate must establish their own notdet stack frame instead of borrowing an existing one. This complicates the implementation of semidet predicates called from within det predicates, e.g. as the condition of an if-then-else.

The other approach is to conceptually transform semidet predicates into deterministic predicates that return a suc-

cess/failure indication. We add a hidden argument to the predicate to hold this boolean value. If the predicate completes successfully, then it must store the value of any output arguments in the appropriate registers, set the success indication to true, and then return. On failure, the predicate must set the success indication to false and return immediately from the point of failure. Predicates that call a semidet predicate must examine the value of the status register and act accordingly. This approach requires either that the abstract machine reserve one register purely for holding success indications, or that we use one of the general purpose registers for this. The latter implies that predicates must know the determinism of the predicates they call, in order to determine which registers to use for argument passing. However, this requirement poses no difficulty, since that information is readily available in declarations.

The current implementation uses the second approach, with success indications stored in a general purpose register (r1). We plan to eventually implement all the approaches we have described, and to compare their efficiency for large programs.

3.5 If-then-else and negation

The if-then-else and negation constructs in most variants of Prolog are non-logical and unsound: they can cause the system to compute answers which are inconsistent with the program viewed as a logical theory. Some existing logic programming systems such as NU-Prolog and Gödel provide logical and sound replacements for these Prolog constructs. Unfortunately, these systems enforce safety via runtime groundness checks, which can be prohibitively expensive in the presence of large terms. This effect can increase the runtime of a program by an arbitrarily large factor.

Mercury provides logical if-then-else and negation without any runtime safety checks. The information provided by the mode system lets the compiler check the safety of these constructs at compile time. For example, consider the predicate to add an element to a list if it isn't already there:

```
:- pred add_element(T, list(T), list(T)).
:- mode add_element(in, in, out) is det.
add_element(Elem, List0, List) :-
   if member(Elem, List0)
   then List = List0
   else List = [Elem|List0].
```

The code the compiler generates for this is:

```
add_element:
   push(succip);
   push(r1);
   push(r2);
   r3 = r2;
   r2 = r1;
   call(member, add_element_l1);
add_element_l1:
   if (!r1) goto add_element_l3;
add_element_l2:
   r3 = pop();
   pop();
   succip = pop();
   proceed();
add_element_l3:
   r2 = pop();
   r1 = pop();
   r3 = mkword(TAG_CONS, create2(r1, r2));
   succip = pop();
   proceed();
```

Since the body of add_element contains a call, we start by saving the succip register. We then save the values of Elem and List0 since they may be needed after the call. Next we set up the input arguments to member; since the called mode of member is semideterministic, they are in r2 and r3 respectively. Member returns its success indication in r1. If member succeeds, we continue execution at add_element_l2, where we pop List0 off the stack and put it in r3, and then return it as the value of List. If member fails, we branch to add_element_l3, where we construct a new cons cell with Elem as the head and List0 as the tail, and return this new list as List.

In general, the condition of an if-then-else may be nondeterministic (this implies that the containing predicate is nondeterministic as well). In such cases the compiler sets up the success and failure continuations differently. The code before the call to the condition sets the redoip slot of the current frame to point to the failure continuation; the call itself puts the success continuation in the succip register. Both continuations must reset the redoip slot to point to the next alternative after the if-then-else; the code that does this at the success continuation prevents backtracking to the else part after the last failure of the condition.

Any variables produced in the condition of an if-then-else may be consumed only in the then part of that if-then-else. Different solutions of the condition may thus lead to different computations in the then part. If a nondeterministic condition produces no such variables, or if they are not consumed anywhere, then only the first solution of the condition is useful. For such conditions the compiler generated code saves the value of maxfr before the call to the condition and restores it afterwards in the success continuation to prune away the unnecessary nondet stack frames.

Our implementation of negation bears strong resemblance to the implementation of if-then-else. This is not surprising, since one can transform queries such as `?- not member(Elem, List0)` into `?- if member(Elem, List0) then fail else true`. For reasons of space, we omit a more detailed discussion.

4 Comparison with the WAM

It is interesting to compare our execution model with the Warren Abstract Machine [1, 23]. We did not design the Mercury execution model by starting with the WAM and

modifying it; we designed it from scratch. Similar constraints led us to similar solutions for some problems, but in most cases, we have found different tradeoffs to be appropriate, usually because of the better information provided by Mercury's declarations. The main differences between the Mercury execution model and the standard WAM are the following.

Our execution model does not need and does not have a general unification primitive. As we have seen at the end of section 2.2 and in the examples of section 3, strong types and modes allow the compiler to generate specialized code for all unifications. These specialized codes have no equivalent of the WAM's read/write mode switch. Deep tests (such as comparing two lists for equality) are implemented as calls to automatically generated recursive predicates. In our experience, such deep tests are quite rare.

Indexing is much simpler in our model. We do not need instructions such as switch_on_term because the mode system tells the compiler which parts of which arguments will be bound at call. Strong typing lets us use simply indexed dense arrays instead of hash tables or decision trees when an argument has a type with many alternatives. This is faster indexing than even C has. A C compiler always has to emit code to check that the expression being switched on is within the domain of the jump table; a Mercury compiler doesn't always have to do this because in many cases it knows the full set of values the switched-on variable can take.

We have separate algorithms for deterministic and nondeterministic code. We combine choice points and environments into one data structure, nondet stack frames. These two points are related. One can classify predicates into three classes based on the number of clauses they have after indexing and the nature of the calls in those clauses:

- predicates with one clause containing calls only to deterministic predicates
- predicates with one clause containing some calls to nondeterministic predicates
- predicates with several clauses

The WAM cannot distinguish between the first two classes because Prolog lacks a mode system and a predicate may be deterministic in one mode and not in another. The WAM therefore draws a line between the second and third classes: it creates environments for predicates in the first two classes and both environments and choice points for predicates in the third class. The omission of choice points for the first two classes is an important optimization because choice points are big. We on the other hand can and do detect determinism and optimize the first class separately and thus very effectively. We treat predicates in the second and third classes the same way because this lets us keep overheads low. For predicates in the third class, our scheme is much faster because we must create only one structure whereas the WAM must create two: we increment a stack pointer only once, do not need to set up links between environment and choice point, and fill in many fewer fixed slots (choice points and environments have a total of nine fixed slots whereas our frames have only four). For predicates in the second class, the WAM looks marginally faster because its environments have only two fixed slots, CE and CP; however, our frames need not store unbound variables, so the total number of slots will be about the same.

Our execution model does not need a trail. In the absence of destructive updates, this is a simple consequence of the fact that our mode system knows exactly what is bound and what is free at any point in the program, and prevents any access to free variables. We can therefore represent free variables as uninitialized words, and resetting variables to "uninitialized" is obviously unnecessary.

To avoid the need for a trail in general, we generate code for destructive updates only in deterministic code, and even then only to update data structures that were allocated since the last backtrack point. This works because it is unnecessary to reset the values of words that will not be referenced after execution restarts from the backtrack point. We apply destructive update only in cases where trailing is not required because the overhead of trailing would make it unprofitable in the majority of the cases we are interested in.

Our execution model does not need dereferencing. The reason is that the mode system guarantees that producers will occur before consumers; therefore the values themselves are available whenever they are needed (to be passed to a predicate or to be put into a structured term). This avoids much unnecessary indirection.

Last call optimization is easier with the WAM. Many predicates that can exploit last call optimization with the WAM cannot do so with our execution model as we have presented it so far. When a predicate takes some values produced by the last call and puts them into a newly constructed cell, the Mercury mode analysis algorithm will reorder the superhomogeneous form of the clause to put the construction unification after what used to be the "last" call. Since the execution algorithm must regain control after the "last" call to construct a cell, the called predicate cannot return directly to the calling predicate's caller. A Prolog system using the WAM would do the construct unification first and then perform the call, and would thus be able to exploit last call optimization.

Tail recursion optimization is last call optimization applied to the case when the last call is a recursive call. When counted dynamically, i.e. by frequency of occurrence at runtime, most last call optimizations are in fact tail recursion optimizations. The Mercury execution model needs only a slight change to let it support tail recursion; section 5.2 shows how to do this. That section also introduces a recursion optimization that works just fine with the clause order produced by mode analysis; section 5.3 then shows that this new optimization can be even more effective than tail recursion optimization.

The Mercury execution model can do last call optimization trivially whenever the last call of a predicate leaves the right values in the right registers. Since programming techniques such as accumulator passing encourage it, this happens surprisingly often.

5 Optimizations

One can apply many optimizations to Mercury programs. Some of these optimizations are standard in the logic programming community, for example, early discarding of nondet frames (choice points in the WAM). Others are standard in the imperative language community, for example register allocation and the elimination of jumps to jumps. For presentation in this section we have selected some optimizations that we believe to be significant and that are either novel in themselves or require a novel implementation in Mercury programs.

5.1 Structure reuse

Conventional logic programming languages cannot express the low-level notion of destructive update. Prolog programmers must choose between resorting to non-logical operations such as assert and retract or implementing an update as the creation of an almost-identical copy. Programs written in the second style have many clauses in which the last reference to a memory cell (structure) is followed almost immediately by the allocation of another cell of the same size. One obvious optimization is to reuse the newly freed memory cell for the following allocation.

This is by now a well researched area [6, 8, 10, 18]. The key problem is the analysis required to identify the location of the last reference to a memory cell. These analyses require dataflow information, and their accuracy is limited by the accuracy of the dataflow information they are based on. Since the Mercury compiler has perfect dataflow information, reuse analyses for Mercury can be simpler, faster and yet more effective than reuse analyses for logic languages without strong type and mode systems.

Suppose the reuse analysis finds that an input argument to a call is the last reference to a structure. It is quite unlikely that *all* calls to the predicate in question will have last references to structures in the same argument position. The object code of the program must therefore contain two separate implementations of the predicate: the old, general version and the new, specialized version. Calls in which that argument is a last reference jump to the specialized version; other calls jump to the general version.

Structure reuse optimization can be applied to many predicates in real programs. We illustrate it with a simple example here; we describe a slightly more complicated example (binary search tree insertion) in section 5.3.

Our example here is the append predicate from section 3.2. The reuse opportunity in append arises when program analysis can prove that there are no further references to the list passed in r1 after the call to append_1, which will frequently be the case. The specialized version of append is:

```
/* append_1 with reuse */
append_1:
   if (r1 != TAG_NIL)
      goto append_1_b0;
   r3 = r2;
   proceed();
append_1_b0:
   push(succip);
   push(r1);
   r1 = field(TAG_CONS, r1, 1);
   call(append_1, append_1_b2);
append_1_b2:
   r4 = pop();
   field(TAG_CONS, r4, 1) = r3;
   r3 = r4;
   succip = pop();
   proceed();
```

This version differs from the version presented in section 3.2 in that the second clause saves A on the stack instead of H, and that instead of allocating a new cons cell on the heap, it reuses A's cons cell for C. Since the first field of this cell already contains H, we do not need to either save H or put its value into the cell.

When applied to very large structures or arrays, structure reuse becomes so important as to be indispensable. It is not sufficient to just hope that the compiler will be able to optimize your array updates — you need to *know* that it will, lest your code suffer orders of magnitude increases in execution time. Mercury's mode system therefore provides a concept called *unique modes*, analogous to linear types in functional programming [22]. Unique modes essentially allow the programmer to declare that there should be only one reference to a particular object, thus guaranteeing that the compiler will be able to optimize updates to that object. Like all the other declarations in Mercury, unique mode declarations are checked at compile time to ensure that they are correct. This checking can be done using only local analysis, since the compiler need only look at the declarations for called predicates, not at their bodies.

Mercury uses unique modes to implement logical input/output. An I/O predicate is logically considered to be a relation between the state of the world before and after the I/O operation is carried out. The unique mode system ensures that there can be only one live reference to the current state of the world, so the compiler can safely implement I/O predicates by destructively updating the state of the world. Determinism analysis must ensure that the implementation never has to backtrack past any I/O operation. This approach is essentially equivalent to the linear types approach in functional programming languages [22].

5.2 Recursion optimizations

Most Prolog systems implement recursion optimization (usually as a special case of tail call optimization). For

the first, deterministic mode of append, the one discussed in section 3.2, mode reordering puts the construction unification `C := [H|NT]` after the recursive call. To exploit tail recursion optimization, we must reverse this order. The reordered clause is

```
append(A, B, C) :- A == [H|T], C := [H|NT],
   append(T, B, NT).
```

This order requires that append return its last argument not in a register but in memory, in the second field of C's cons cell. The tail recursive version of append should therefore be passed the address of this field in r3. To allow other predicates to call append without knowing how append was compiled, we need an interface procedure to convert between the parameter passing conventions. Using a variant of the code of the predicate itself as the interface procedure is a simple and efficient technique. The new implementation of append is

```
/* append_1 interface procedure */
append_1:
   if (r1 != TAG_NIL)
      goto append_1_b0;
   r3 = r2;
   proceed();
append_1_b0:
   push(succip);
   r4 = mkword(TAG_CONS,
      create2_bf(field(TAG_CONS, r1, 0)));
   push(r4);
   r3 = (Word) &field(TAG_CONS, r4, 1);
   r1 = field(TAG_CONS, r1, 1);
   call(append_1tr, append_1_b1);
append_1_b1:
   r3 = pop();
   succip = (Code *) pop();
   proceed();

/* tail recursive internal procedure */
append_1tr:
   if (r1 != TAG_NIL)
      goto append_1tr_b0;
   * (Word *) r3 = r2;
   proceed();
append_1tr_b0:
   r4 = mkword(TAG_CONS,
      create2_bf(field(TAG_CONS, r1, 0)));
   * (Word *) r3 = r4;
   r1 = field(TAG_CONS, r1, 1);
   r3 = (Word) &field(TAG_CONS, r4, 1);
   goto append_1tr;
```

The second clause of the interface procedure allocates a cons cell for C but fills in only the first field, with H. It remembers the address of this cell for later return to the caller in r3. It then sets up the arguments of the call to the tail recursive internal procedure: it puts T in r1, as usual, and puts the address where NT should be put in r3. B is already in r2. It then calls the internal procedure append_1tr, the only call to that procedure in the program.

The first clause of append_1tr is different only in where it puts the output argument. The second clause creates a new cons cell (filling in the first field with H) and puts in the address of this cell where the caller asked it to be put. It then sets up the arguments for the recursive call to fill in the second field of the new cons cell and performs tail recursion. Eventually control will reach the first clause of append_1tr and the proceed invoked from there will return to append_1_b1.

Another way to optimize append is a new compilation technique we call middle recursion optimization. This applies whenever the execution pattern of a deterministic predicate follows the pattern of the grammar $D^n A U^n$. The pattern applies to all simply (i.e. not mutually) recursive deterministic predicates.

In the case of append, the D (down) part is the execution of `A == [H|T]`, the A (across) part is the execution of `C := B`, and the U (up) part is the execution of `C := [H|NT]`, n being given by the length of the list in r1. `A == []` is part of the control flow, as is the part of `A == [H|T]` that tests A for a cons tag. The optimization consists of replacing the original control structure with two loops written inline containing the original code for `A == [H|T]` and `C := [H|NT]` respectively, with the code for `C := B` in the middle.

```
/* append_1 with middle recursion */
append_1:
   r5 = 0;
   while (r1 != TAG_NIL) {
      push(field(TAG_CONS, r1, 0));
      r1 = field(TAG_CONS, r1, 1);
      r5 = r5 + 1;
   }
   r3 = r2;
   while (r5 > 0) {
      r4 = pop();
      r3 = mkword(TAG_CONS, create2(r4, r3));
      r5 = r5 - 1;
   }
   proceed();
```

The register r5 counts up from 0 to n and then down again. If there were calls to other predicates (as opposed to system primitives) before and/or after the recursive call, we would have to keep the up and/or the down counter on the stack, since calls can destroy any register. (For code that in Prolog systems would be tail recursive, the down counter can always be kept in a register.) If e.g. the up counter must be on the stack, we must push zero on the stack before the loop; inside the loop we must then pop the counter off the stack, push any values to be saved and then push the new value of the counter. This way, the counter is always on the top of stack when a call is made.

Simply recursive predicates with more than one recursive clause may still be deterministic; insertion into a binary search tree is an example. In such cases, the down part must select the recursive clause that will produce the solution, and push the number of the clause on the stack;

the up part must execute the remaining part of the indicated clause.

Accessing the stack for loop counters or clause indicators reduces the performance gain to be had from middle recursion optimization. Nevertheless the optimization can still be worthwhile, especially since it sometimes creates the conditions required for the destructive assignment optimization.

5.3 Destructive assignment optimization

The tail recursion and structure reuse optimizations are orthogonal; they can be applied independently of each other. The middle recursion and structure reuse optimizations are also orthogonal, but applying both also creates an opportunity for further optimization.

```
append(A, B, C) :- A == [], C := B.
append(A, B, C) :- A == [H|T],
   append(T, B, NT), C := [H|NT].
```

With structure reuse optimization, the second clause replaces T with NT in the cons cell of A, which it reuses as the value of C. When a recursive call invokes the second clause of append, the reuse is total; the cons cell already has the right values in both its fields. Since all recursive calls except the last invoke the second clause, this means that in the pattern D^nAU^n, the last U^{n-1} part performs no useful computation. It can therefore be removed provided we also adjust the code that manipulates the stack. We need only the first and last values to be popped off the stack; the first provides the address of the only cons cell whose contents are to be modified and the last provides the final value of r3. We can replace both uses of the stack with variables, one assigned before the loop (r6) and one assigned every time through the loop (r7). The final touch is making sure that the case $n = 0$ is handled correctly. We could test the counter r5 and execute the code for modifying the last cons cell only if r5 is greater than zero, but this would require us to keep the r5 counter. It is better to eliminate r5 altogether and to handle the case $n = 0$ separately; the code for this is exactly the original code for the first clause. The resulting code looks like this:

```
/* append_1 with destructive assignment */
append_1:
   if (r1 != TAG_NIL)
      goto append_1_b0;
   r3 = r2;
append_1_b0:
   r6 = r1;
   do {
      r7 = r1;
      r1 = field(TAG_CONS, r1, 1);
   } while (r1 != TAG_NIL);
   r3 = r2;
   field(TAG_CONS, r7, 1) = r3;
   r3 = r6;
   proceed();
```

The C compiler ought to be able to optimize this code even further. Value numbering should let the compiler deduce that the assignment `r3 = r2` can be eliminated by assigning r2 directly into the cons cell. If one could tell the C compiler that r6 is intended to be dead at the end of the clause, then the two assignments in which it is involved could be short-circuited as well, yielding the following code:

```
/* fully optimized append_1 */
append_1:
   if (r1 != TAG_NIL)
      goto append_1_b0;
   r3 = r2;
   proceed();
append_1_b0:
   r3 = r1;
   do {
      r7 = r1;
      r1 = field(TAG_CONS, r1, 1);
   } while (r1 != TAG_NIL);
   field(TAG_CONS, r7, 1) = r2;
   proceed();
```

This is very close to the version that a C programmer would write from scratch. The code for append in the GNU C++ library (cleaned up for publication) is

```
ListNode *append(ListNode *a, ListNode *b)
{
   if (a == NULL) {
      return b;
   } else {
      ListNode *tmp = a;
      while (tmp->next != NULL)
         tmp = tmp->next;
      tmp->next = b;
      return a;
   }
}
```

In fact, the "fully optimized" version of append above yields slightly faster code than this handwritten version.

Our optimizations produce similarly good code for other predicates besides append. Consider the predicate that inserts a key/value pair into a binary search tree.

```
:- pred insert(tree(int, T), int, T,
   tree(int, T)).
:- mode insert(in, in, in, out) is det.
insert(Init_tree, Key, Value, Final_tree) :-
   Init_tree == leaf,
   Final_tree := tree(leaf, Key, Value, leaf).
insert(Init_tree, Key, Value, Final_tree) :-
   Init_tree == tree(Left, Curkey, _, Right),
   compare(Key, Curkey, Result),
   (
      Result = lt,
      insert(Left, Key, Value, Newleft),
      Final_tree :=
         tree(Newleft, Key, Value, Right)
   ;
      Result = eq,
      Final_tree :=
```

```
        tree(Left, Key, Value, Right)
    ;
        Result = gt,
        insert(Right, Key, Value, Newright),
        Final_tree :=
            tree(Left, Key, Value, Newright)
    ).
```

The structure reuse, tail recursion and middle recursion optimizations can all be applied to this code, although as usual the two recursion optimizations are mutually exclusive. The structure reuse optimization reuses the input tree cell in the first and third alternatives in the disjunction. The tail recursion optimization alters the parameter passing so that calls to the internal tail-recursive procedure pass the address where the result should be put in r4. If structure reuse and tail recursion are applied together, the returned result will almost always be the address passed in r1.

If structure reuse and middle recursion are applied together, we can apply the destructive assignment optimization as well. The code generated by the destructive assignment optimization has a structure similar to the code for append above, in that it has separate code for the first clause, and that the code for the second clause is mostly a loop. In this case the loop has two termination conditions: we exit the loop if either Init_tree is a leaf or if it is a tree node whose key is equal to Key. The loop body puts the address of Init_tree into a register (say r5), records the result of the comparison between Key and Curkey in another register (say r6), and sets the register containing Init_tree to either Left or Right depending on the result of the comparison. If we exited the loop because Init_tree is a leaf, the code following the loop allocates a new tree node, fills it with Key, Value and two leaf subtrees and puts it in either the left or right subtree field of the tree cell pointed to by r5 depending on the value of r6. If we exited the loop because Key was equal to the key field of Init_tree, the code simply overwrites the value field of the last Init_tree with Value.

6 Performance results

In this section we compare the performance of Mercury programs with the performance of programs written in other logic programming languages.

6.1 Background

The Mercury compiler can generate four types of C programs; the programmer can choose between them via command-line switches. The four options are:

ansi not using either non-local gotos or global register variables

reg using global register variables, but not using non-local gotos

jump using non-local gotos, but not using global register variables

fast using both non-local gotos and global register variables

With the first two options, ansi and reg, the Mercury compiler converts each labelled piece of code into a function, and implements abstract machine gotos as return statements, with the return value being a function pointer that tells the driver routine which function to call next. With the last two options, jump and fast, the Mercury compiler emits code that looks like the examples in the paper. The reg and fast options tell the compiler to declare the most frequently used abstract machine registers as global register variables. The number of abstract machine registers that can be so declared depends on the hardware architecture and the configuration of gcc. To simplify calls to C library functions, at the moment we prefer to exploit only registers that gcc designates to be callee-save; we use 10 global register variables on SPARCs and 8 on MIPS processors. When not exploiting either GNU C extension, i.e. when using the ansi option, the Mercury compiler generates portable C code.

The Mercury runtime system is implemented as a shared library on machines that support shared libraries (really shared objects), and as a traditional library on machines that do not. The run-time system contains the driver routine for programs compiled with the ansi and reg options. It also defines several system-provided labels, including the one that handles failures. The runtime system is also responsible for the initial allocation of memory for the various data areas. This allocation is careful to place the data areas in such a way that their initial areas do not collide in direct-mapped caches. (While this step doesn't hurt large programs, omitting it can slow down small programs by more than 30%.)

On Unix machines that support the mprotect system call, which includes machines running Solaris 2.x, IRIX 5.x and many other systems based on System V Revision 4, the run-time system detects overflow of any of the data areas. (This approach had problems with earlier versions of Unix, but works just fine with Solaris 5.3 and IRIX 5.2. For machines on which mprotect doesn't work, or which do not have mprotect or its equivalent, the compiler will have to put code to check for overflows at the start of each procedure, but we expect such machines to be increasingly rare.) We intend the signal handler that is notified on overflow to arrange for the invocation of the garbage collector. Our garbage collector is not finished yet, but this is not a problem since the benchmark programs do not require any garbage collection. We do have a design for the garbage collector that solves the difficulties caused by our tagging scheme and the presence of undefined values on the stack and in the heap; the design uses techniques developed for the garbage collection of strongly typed imperative languages such as Modula-3. In any case, Mercury can be expected to use somewhat less heap space than traditional systems, since in Mercury most structures do not need an extra word to store the identity of their function symbol.

6.2 Speed tests

System	Variant	arith	geom	harm
SWI	no decls	1.00	1.00	1.00
NU	no decls	1.84	1.77	1.72
	decls	4.92	2.17	1.67
Sicstus	compact, no decls	2.09	2.05	2.02
	compact, decls	4.10	2.26	1.88
	fastcode, no decls	7.45	7.12	6.83
	fastcode, decls	5.77	3.70	3.15
Aquarius	no analysis, no decl	12.76	11.23	9.87
	no analysis, decl	16.09	13.80	12.26
	analysis, no decl	24.79	21.49	18.42
	analysis, decl	28.92	22.96	19.00
Mercury	ansi	13.08	11.62	10.54
	jump	20.25	18.15	16.66
	fast	40.36	35.25	32.22

Table 1: Averages of benchmark speed ratios

We have tested the speed of the Mercury implementation on a set of standard Prolog benchmarks which we have translated to Mercury. Besides Mercury, we ran the benchmarks on four other logic programming languages: SWI-Prolog 1.9.0 [24], NU-Prolog 1.6.4 [20], Sicstus Prolog 2.1 [2] and Aquarius Prolog 1.0 [21]. These versions are the latest we have access to. We included Aquarius in our benchmarks because of its reputation as the fastest generally available Prolog system. We included Sicstus because of its wide user base. We included NU-Prolog because of its pioneering implementation of coroutining. We included SWI-Prolog because its portability and free availability should enable every reader to calibrate our results. We would have liked to run the benchmarks on some other systems as well, Quintus Prolog and PARMA [19] in particular, but we do not have access to those.

For all the Mercury benchmarks we report on, the code we tested came straight out of the compiler; we did not modify any of them in any way. Since we have not yet incorporated the optimizations described in section 5 into the compiler, our performance results indicate directly the effectiveness of the execution model we described in section 3.

SWI-Prolog and NU-Prolog are bytecode interpreters. The compact option of Sicstus is also a bytecode interpreter, while its fastcode option is a native code compiler (the fastcode we tested is not the one described in [5], but based on the results reported in that paper, our results would not be much different if it were). Aquarius is also a native code compiler. Aquarius has an option asking the compiler to analyze the program; we tested Aquarius both with and without this option. With the exception of SWI-Prolog, all these Prolog systems allow programmers to supply declarations, but none require it. We therefore tested each such system both with and without declarations.

NU-Prolog's when declarations and Sicstus Prolog's block declarations specify that a predicate should be called only when certain variables have been bound. This can be used to achieve coroutining; the NU-Prolog and Sicstus compilers also use the information these declarations provide to generate better code for indexing. However, the declarations cause checks to be made at runtime.

Aquarius's declarations can specify that an argument of a predicate is ground at the time of the call, that it is already dereferenced at the call, and that it is a list or an integer. Although these declarations cannot be used to achieve coroutining, they can speed up the program. The problem with these declarations is that the Aquarius compiler does not catch incorrect declarations even though they can cause the program to crash. This is a significant concern, since the declarations are quite easy to get wrong.

The native code generators of Sicstus Prolog and Aquarius Prolog each target a small number of platforms, We ran the benchmarks on the fastest machine we have access to that can run binaries generated by these systems. This machine is a Sun SPARCserver 1000 with four 50 MHz TI SuperSPARC processors and 256 megabytes of memory running SunOS 5.3 (Solaris 2.3). Each processor is rated at 60.3 SPECint92, and has a 4-way associative 16 Kb I-cache and a 5-way associative 20 Kb D-cache backed up by 1 Mb of secondary cache. None of our tests used more than one processor, but the presence of the other processors significantly reduced the effect of other machine loads on our benchmarks.

The Aquarius compiler does not run under Solaris 2 due to differences in assembler formats, so we compiled Aquarius benchmarks on a SPARCstation 2 running SunOS 4.1.2, although of course we ran the resulting executables on the SPARCserver 1000. The register windows on SPARC processors make it difficult to exploit global register variables without also exploiting non-local gotos, so on these machines we can report results for only three of Mercury's four code generation options. We ran the benchmarks in multiuser mode on a mostly quiescent machine at night. To eliminate the effects of any background loads as far as possible, every result we report is the best one out of four or more trials.

Table 1 contains summaries of our timing results, using the arithmetic, geometric and harmonic means respectively. These numbers are derived from table 2, which shows the speed of each variant of each system on each benchmark. All speeds in this table are normalized to the speed of SWI-Prolog, a publically available system. Table 3 contains the same data normalized to the speed of the fastest system on each given benchmark; the speed of this system is shown as 100%. Table 4 contains the raw timing data, user-mode times required to produce all solutions. The units are seconds for queens and microseconds for deriv, nrev and qsort, and milliseconds for all the other benchmarks. The times were produced by running each benchmark many times, in a failure driven loop for the Prolog benchmarks and in its equivalent in a test harness for Mercury, and dividing the total user-mode runtime by the number of repetitions; this eliminates the uncertainty involved in measuring small times.

System	Variant	cqueens	crypt	deriv	nrev	poly	primes	qsort	queens	query	tak
SWI	no decls	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
NU	no decls	1.5	1.4	1.5	3.4	1.8	1.6	1.6	1.6	1.9	2.2
	decls	1.6	1.2	0.9	3.4	1.6	1.7	1.2	33.8	1.9	1.9
Sicstus	compact, no decls	2.1	2.1	1.6	3.0	2.2	1.7	1.8	2.1	1.8	2.5
	compact, decls	1.9	1.8	1.3	1.9	1.9	1.5	1.4	25.3	1.7	2.2
	fastcode, no decls	6.1	6.0	6.8	12.1	7.9	5.7	6.9	6.8	4.4	11.6
	fastcode, decls	3.5	3.1	2.4	2.2	3.2	2.7	2.2	30.8	3.9	3.8
Aquarius	no analysis, no decl	16.2	7.3	7.2	15.7	9.8	8.1	14.7	15.9	4.5	28.0
	no analysis, decl	17.5	7.4	7.9	19.4	12.6	9.0	16.7	17.4	8.6	44.3
	analysis, no decl	25.9	7.5	15.7	36.9	18.8	11.7	32.6	26.1	18.2	54.5
	analysis, decl	25.9	7.6	15.8	36.8	18.9	12.9	32.8	26.1	18.2	94.2
Mercury	ansi	10.8	12.3	5.6	7.4	12.1	10.2	10.0	9.6	23.2	29.5
	jump	20.9	16.1	10.1	11.0	20.2	15.0	14.4	16.3	29.4	49.0
	fast	39.1	24.5	19.4	29.9	39.7	24.3	41.3	28.8	39.6	116.8

Table 2: Benchmark speed ratios on SPARCserver 1000, SWI-Prolog = 1

System	Variant	cqueens	crypt	deriv	nrev	poly	primes	qsort	queens	query	tak
SWI	no decls	2.6	4.1	5.2	2.7	2.5	4.1	2.4	3.0	2.5	0.9
NU	no decls	3.7	5.6	7.7	9.3	4.4	6.7	3.8	4.8	4.8	1.9
	decls	4.2	4.9	4.5	9.3	4.0	6.8	3.0	100.0	4.7	1.7
Sicstus	compact, no decls	5.3	8.5	8.0	8.2	5.6	7.0	4.4	6.1	4.5	2.2
	compact, decls	4.9	7.4	6.9	5.2	4.8	6.3	3.3	75.0	4.3	1.9
	fastcode, no decls	15.5	24.6	35.2	32.8	20.0	23.6	16.8	20.3	11.1	9.9
	fastcode, decls	8.9	12.7	12.5	5.9	8.0	11.0	5.4	91.1	9.7	3.2
Aquarius	no analysis, no decl	41.5	29.7	37.2	42.7	24.8	33.4	35.7	47.0	11.4	24.0
	no analysis, decl	44.6	30.3	41.1	52.7	31.6	36.9	40.5	51.5	21.8	38.0
	analysis, no decl	66.2	30.7	81.3	100.0	47.4	48.2	78.8	77.3	45.8	46.7
	analysis, decl	66.2	30.9	81.4	100.0	47.6	52.9	79.4	77.4	45.8	80.7
Mercury	ansi	27.6	50.1	29.0	20.1	30.5	42.1	24.1	28.5	58.5	25.3
	jump	53.4	65.7	52.4	29.9	51.0	61.7	34.9	48.3	74.1	41.9
	fast	100.0	100.0	100.0	81.2	100.0	100.0	100.0	85.2	100.0	100.0

Table 3: Benchmark speed percentages on SPARCserver 1000, fastest = 100%

The benchmarks we used are standard Prolog benchmarks which we have translated to Mercury. Crypt solves a cryptoarithmetic puzzle. Deriv symbolically differentiates four functions of a single variable. Nrev reverses a list of 30 elements using the naive algorithm. Poly symbolically raises 1+x+y+z to the tenth power. Primes finds all primes up to 100. Qsort quicksorts a list of 50 integers using difference lists. Queens finds all safe placements of 9 queens on a 9x9 chessboard. Cqueens is the same benchmark after it has been put through the source-to-source transformation of Seki and Furukawa [13]. Query finds countries with approximately equal population density. Tak is an artificial benchmark, originally written in Lisp; it is heavily recursive and does lots of simple integer arithmetic. Cqueens, crypt, queens and query are mostly nondeterministic while deriv is mostly semideterministic; the other five benchmarks are mostly deterministic.

Table 1 shows Mercury to be the fastest system overall with all three averaging methods. Using the harmonic mean, the fast option of Mercury is 70% faster than the fastest variant of Aquarius, and it outperforms Sicstus fastcode by a factor of 4.7; it outperforms all the other systems we measured[2] by factors ranging from 16 to 32. Using the geometric mean, Mercury is 53% faster than Aquarius, while using the arithmetic mean, Mercury is 40% faster. For averaging rates, the harmonic mean is the most appropriate of the three averaging methods, since it is the one least influenced by a single good result. For example, tables 2 to 4 show that declarations slow down Sicstus compact code on every benchmark except one (queens), yet the arithmetic and geometric means show Sicstus compact code to be faster with declarations than without, due to the big speedup made possible by declarations on that one benchmark.

Although we expect the fast version of Mercury to be available on almost all machines, even the ansi version of Mercury is more than 50% faster than Sicstus fastcode, which, like Aquarius, runs on only a few machines. Given that the only optimizations implemented by the Mercury compiler so far are some forms of indexing and some local peephole optimizations, these results reflect the inherent efficiency of the basic Mercury execution algorithm from

[2] Andrew Taylor, the author of PARMA [19], has seen the C code generated by the Mercury compiler and the assembly code generated from that by gcc. He says that for the examples he looked at, Mercury usually generates better code than PARMA.

System unit	Variant	cqueens msec	crypt msec	deriv usec	nrev usec	poly msec	primes msec	qsort usec	queens sec	query msec	tak msec
SWI	no decls	2421	100.1	831.4	4265	384.9	9.69	4395	90.78	78.08	1450.5
NU	no decls	1663	73.0	557.9	1246	219.2	5.99	2776	56.10	40.82	667.3
	decls	1480	83.3	945.3	1246	245.2	5.82	3550	2.69	41.70	749.4
Sicstus	compact, no decls	1176	48.1	533.8	1404	172.3	5.66	2410	43.94	43.73	577.5
	compact, decls	1258	55.4	618.9	2220	202.9	6.32	3183	3.58	45.73	663.6
	fastcode, no decls	399	16.6	122.0	352	48.5	1.69	633	13.27	17.73	125.4
	fastcode, decls	694	32.1	344.2	1955	121.1	3.62	1986	2.95	20.21	384.4
Aquarius	no analysis, no decl	149	13.7	115.4	271	39.1	1.19	298	5.72	17.21	51.8
	no analysis, decl	139	13.5	104.6	219	30.7	1.08	263	5.22	9.05	32.7
	analysis, no decl	93	13.3	52.8	116	20.4	0.83	135	3.48	4.30	26.6
	analysis, decl	93	13.2	52.8	116	20.4	0.75	134	3.47	4.30	15.4
Mercury	ansi	224	8.1	148.0	577	31.7	0.95	441	9.43	3.37	49.1
	jump	116	6.2	82.0	387	19.0	0.65	304	5.57	2.66	29.6
	fast	62	4.1	43.0	142	9.7	0.40	106	3.15	1.97	12.4

Table 4: Benchmark times on SPARCserver 1000

section 3. We therefore confidently expect that Mercury will perform just as well on large programs.

Mercury is the fastest system on eight out of the ten benchmarks, and it is within 20% the speed of the fastest system on the other two. Aquarius with analysis and declarations is the next fastest system. Aquarius beats Mercury on one benchmark, nrev, because Mercury does not yet implement either of the recursion optimizations discussed in section 5.2. The Mercury peephole optimizer does perform last call optimization (and therefore tail recursion optimization) if the last call happens to leave the values of all the outputs of the predicate in the right registers. Such opportunities do in fact occur in primes, qsort and tak. They are surprisingly frequent in real programs as well due to the popularity of the accumulator-passing style of programming.

Mercury and Aquarius are both beaten by NU-Prolog and Sicstus on the nine-queens program. The complexity of this benchmark is O($N!$) when executed left-to-right, but coroutining reduces this to O(N^2). This is why NU-Prolog and Sicstus Prolog get such good results. However, the results for cqueens show that even on these systems, compiling away the coroutining improves performance significantly. With NU-Prolog, the improvement is 82%; with Sicstus Prolog, the improvement is 196% for compact code and 338% for fastcode.

This is why we currently do not have any plans to implement coroutining for Mercury. We intend instead to implement source-level program transformations to change the order of evaluation to a more efficient one, as in [13], and to get the compiler to emit parallel code. (These two approaches also work for programs with circular data dependencies, which currently we do not allow; we intend to relax this restriction.) Coroutining and parallelism both impose overheads on the basic execution algorithm: in some places the code must check whether a variable is instantiated yet, while all code fragments that can possibly instantiate a waited-on variable must check whether they need to wake up any delayed goals. As the results for NU-Prolog and the two variants of Sicstus Prolog show, with a coroutining system these overheads are usually worthwhile only if they can transform the big-O complexity of the program. With a parallel system, the overhead should be worthwhile whenever the program has any significant amount of parallelism. Since Mercury is a pure language, the only dependencies between predicates are those required by the data flow, and therefore many Mercury programs should have useful parallelism. This holds true even for fully deterministic Mercury programs with a stream AND-parallel implementation [17]. A parallel Mercury implementation running on a single processor is of course a coroutining Mercury implementation.

It is striking that with NU-Prolog and Sicstus Prolog, although declarations can speed up the program dramatically by inducing coroutining (as in queens) or slightly by allowing better indexing (e.g. NU-Prolog on cqueens), they can also cause major slowdowns due to the overhead incurred by the runtime checks they cause to be generated. This is one reason why Prolog programmers traditionally dislike declarations. This shouldn't be a problem with Mercury, since our results show that the Mercury compiler puts declarations to good use.

The tables show that Mercury benefits significantly from both gcc's non-local gotos and its global register variables, with the latter being the more important optimization. Without gcc's nonlocal gotos, each transfer of control costs two jump instructions (to the driver and to the destination) plus possibly one more back to the start of the unrolled loop of the driver. This cost is incurred even when falling through a label. With gcc's nonlocal gotos, the costs are one instruction and zero instructions respectively. In our context "global register variables" means keeping the most important abstract machine registers in the registers of the physical machine. The payoff from using global register variables is large (in program size as well as execution time) because the alternative is accessing memory, or at least the cache, on every reference to a virtual machine register, and these occur on almost every line of code.

The speeds reached by the three versions of Mercury

on the SPARCserver 1000 on the nrev benchmark correspond to 3.5 Megalips, 1.3 Megalips and 860 Kilolips respectively. (A single execution of the naive reverse benchmark counts as 496 logical inferences; "lips" is logical inferences per second.) Applying the destructive assignment optimization from section 5.3 to append raises the speed of nrev to 10.0 Megalips, 4.8 Megalips and 4.5 Megalips for the fast, jump and ansi options respectively. Note the much smaller difference between the speeds of the jump and ansi versions. The reason is that the optimized version of append uses iteration instead of recursive calls, so the speed of calls is much less important than with the original code.

6.3 Program size and compilation time

The programs generated by the Mercury compiler for these benchmarks are quite small. The biggest benchmark is poly, whose source is about 260 lines of Mercury. Compiled to ANSI C, its executable is 22 Kb; the stripped executable is 14 Kb. Compiled to exploit both GNU C extensions, its executable is 11 Kb, with the stripped executable being 8 Kb. Aquarius, on the other hand, creates 900+ Kb executables for even the smallest benchmarks, of which more than 770 Kb remains after stripping. The standard Aquarius library accounts for almost all of this bulk. On SPARCs, you need to run Solaris to be able to conveniently create shared libraries, and the Aquarius compiler does not run on Solaris; it also lacks the ability to load in only the parts of the library that are needed by the program. The executables created by NU-Prolog are small shell scripts that refer to save files that are always at least 270 Kb in size; most of this is the NU-Prolog interpreter. SWI-Prolog produces very small executables ranging from 1.5 to 4 Kb in size; these are actually shell scripts invoking the SWI-Prolog runtime system and giving it its binary data.

To give a real example, the typechecking module of the Mercury compiler is about 2600 lines of Mercury. When given the fast option and this 95 Kb file, the Mercury to C compiler generates about 330 Kb of C code, which gcc -O2 -msupersparc then converts to about 90 Kb of object code. Given the high level nature of Mercury code, this is quite competitive with compilers for imperative languages such as C. Although we generate a separate code sequence for each mode of a predicate, this does not cause any problems with code size for two reasons. First, very few predicates are used in more than one mode. Second, the code for each mode can be small because it is specialized; the native code emitted by a conventional Prolog compiler has to be prepared to handle all possible dataflow patterns.

Sicstus does not have a separate compiler as such, instead the interpreter compiles consulted files internally if a certain flag is set. This works because Sicstus's internal compiler is very fast, compiling each benchmark in small fractions of a second. SWI-Prolog's compiler is about as fast. Among the other systems, NU-Prolog's compiler is fastest with Aquarius Prolog's compiler being the slowest. The speed of the Mercury compiler falls in between.

In most cases, compiling Mercury to C and compiling C to optimized object code each account for about half of the time required for a Mercury compilation. We expect the speed of the Mercury to C part of the compilation to increase significantly once we bootstrap the Mercury compiler. Since the Mercury compiler is currently run as a NU-Prolog program, the performance data in the tables above provides ample support for this expectation. The way to speed up the C to object part of the compilation is simply to invoke the C compiler without any optimization flags. This is what C programmers do whenever compilation time is important to them.

The C compiler currently performs optimizations we know to be unnecessary or ineffective for the code we generate, while other optimizations could be made more effective and/or speeded up with access to Mercury source-level semantic information. We could therefore speed up the overall compilation process for Mercury programs by compiling directly to object code. However, this would require us to expend time and effort on reimplementing well-understood techniques and would sacrifice the portability of the Mercury implementation for the sake of very small improvements. For the time being, we prefer to spend our time more productively by investigating higher level optimizations (usually source-to-source transformations) that can improve the big-O time complexity of the program. Eventually, however, we may write native code generators to supplement, not replace, the C code generator.

7 Conclusion

Traditionally most people have equated logic programming with Prolog, and concluded that logic programming is not suitable for writing application programs except in narrow domains. One major reason for this is that most implementations of Prolog are quite slow, and fast implementations of Prolog exist on only a few platforms. Prolog programmers usually attack efficiency problems by using non-logical constructs, e.g. by putting in cuts to prevent the exploration of parts of the search space. These constructs destroy the declarative reading of the program, and complicate the job of the compiler even further. Another major problem is that Prolog offers no support for the construction of large reliable software systems. For example, when a piece of code intended to always succeed fails instead, the programmer has no help in tracing the cause of the failure, and when an important data structure changes the programmer has no help in finding all the parts of the program that must be updated.

Our approach represents a clean break with logic programming tradition in that we designed Mercury according to the principles of software engineering. Since we wanted to realize the advantages promised by logic programming, Mercury is a purely declarative high level language with no non-logical features; even I/O is declarative. The module system helps groups of programmers

cooperate in the construction of large programs. The type, mode and determinism declarations we require provide important documentation that can be relied upon and help the compiler prevent a large fraction of program errors. They also provide information that the compiler can use to make the implementation much more efficient. Type information allows the compiler to specialize term representations; mode and determinism information allows it to specialize the code generated e.g. for parameter passing and for unifications. Our benchmarks confirm the superior speed of this approach compared with traditional approaches that cannot rely on access to such information.

We are writing the Mercury compiler in Mercury itself, using NU-Prolog for bootstrapping via an automatic translator from a subset of Mercury to NU-Prolog. The compiler consists of 40,000 lines of Mercury code so far, and is thus representative of the program size range for which Mercury was designed (thousands to millions of lines of code). Our experience with writing this code confirms our expectations about the superior software engineering qualities of the language. When we program in C, C++ or Prolog, we are used to having to chase down most bugs fairly laboriously. This also happened in the early part of our development of the Mercury compiler. However, once we got Mercury's type, mode and determinism checks working, the number of bugs we had to chase ourselves, as opposed to the compiler pointing them out, dropped dramatically. These days, it is not unusual for us to add a significant piece of functionality, and find that after fixing the errors detected by the compiler, the code runs correctly the first time. The first person outside the Mercury development team to use Mercury, David Kemp, reports the same experience.

Mercury is in some respects a less expressive language than Prolog. For example, the mode system requires Mercury programs to have fixed dataflow patterns, and prevents them from using unifications that would make one free variable an alias for another. While writing 43,000 lines of Mercury code, (the compiler and some small applications, including a scanner generator) we have bumped into these limitations very rarely. Even then, we found them easy to code around, and found the resulting code to be easier to understand than the original. This is consistent with the experiences reported in [4] and [19]. We consider Mercury's limitations to be more than worthwhile considering the benefits they bring in terms of program reliability, understandability, maintainability, and efficiency.

We would like to thank Will Winsborough, Jeff Schultz and Andrew Taylor for our discussions, David Kemp, James Harland, Jayen Vaghani, Kotagiri Ramamohanarao, Jeff Schultz, Lee Naish and Philip Dart for their comments on drafts of this paper, and the Australian Research Council and the Centre for Intelligent Decision Systems for their support.

References

[1] H. Ait-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, 1991.

[2] M. Carlsson and J. Widen. Sicstus-Prolog user's manual. Technical Report 88007B, Swedish Institute of Computer Science, Kista, Sweden, 1988.

[3] S. K. Debray and D. S. Warren. Detection and optimisation of functional computations in Prolog. In *Proceedings of the Third International Conference on Logic Programming*, pages 490–504, London, England, June 1986.

[4] W. Drabent. Do logic programs resemble programs in conventional languages? In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 389–396, San Francisco, California, August 1987.

[5] R. C. Haygood. Native code compilation in SICStus Prolog. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 190–204, Santa Margherita Ligure, Italy, June 1994.

[6] F. J. Henderson. Strong modes can change the world! Technical Report 93/25, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1993.

[7] P. M. Hill and J. W. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[8] A. Marien, G. Janssen, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretations: an experiment in code generation. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 33–47, Lisboa, Portugal, June 1989.

[9] R. Milner. A proposal for standard ML. In *Conference Record of the ACM Symposium on LISP and Functional Programming*, pages 184–197, Austin, Texas, July 1984.

[10] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programs. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, Israel, June 1990.

[11] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[12] F. Pfenning, editor. *Types in logic programming*. MIT Press, 1992.

[13] H. Seki and K. Furukawa. Notes on transformation techniques for generate and test logic programs. In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 215–223, San Francisco, California, August 1987.

[14] G. Smolka. Making control and data flow in logic programs explicit. In *Conference Record of the ACM Symposium on LISP and Functional Programming*, pages 311–322, Austin, Texas, July 1984.

[15] Z. Somogyi. Stability of logic programs: how to connect don't-know nondeterministic logic programs to the outside world. Technical Report 87/11, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1987.

[16] Z. Somogyi. *A parallel logic programming system based on strong and precise modes*. PhD thesis, University of Melbourne, Australia, January 1989.

[17] Z. Somogyi. A system of precise modes for logic programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 769–787, Melbourne, Australia, June 1987.

[18] A. Taylor. LIPS on a MIPS: results from a Prolog compiler for a RISC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185, Jerusalem, Israel, June 1990.

[19] A. Taylor. *High performance Prolog implementation.* PhD thesis, University of Sydney, Australia, June 1991.

[20] J. Thom and J. Zobel. NU-Prolog reference manual. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[21] P. van Roy et al. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.

[22] P. Wadler. Linear types can change the world! In *Proceedings of the IFIP TC2 Conference on Programming Concepts and Methods*, pages 547–566, April 1990.

[23] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, October 1983.

[24] J. Wielemaker. SWI-Prolog reference manual. Technical report, Department of Social Science Informatics, University of Amsterdam, Netherlands, 1993.