The execution algorithm of Mercury, an efficient purely declarative logic programming language

Zoltan Somogyi, Fergus Henderson and Thomas Conway
{zs,fjh,conway}@cs.mu.OZ.AU
Department of Computer Science, University of Melbourne
Parkville, 3052 Victoria, Australia

▷ 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, yielding very efficient code. The Mercury compiler uses this execution model to generate portable C code. Our benchmarking shows that the code generated by our 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 (e.g. when memory is allocated or freed) to the language implementation. This declarativeness also means that pure logic programs are ideal subjects for the application of formal mathematical techniques. Such uses include proofs of correctness, automatic program transformation and optimization, automatic parallelization, and the use of declarative debugging.

Despite these advantages, logic programming languages have not been successful in capturing more than a niche market. The reasons have to do with the pragmatic requirements of application programmers:

- 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 by one programmer do not propagate unnecessarily to other parts of the program, which may be the responsibility of other programmers.
- Application programmers need 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 redundancy often turns out to be useful documentation; since the compiler's checks ensure that declarations are accurate, declarations are much more useful than comments with the same content would be.

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

Current logic programming languages do not rate very highly when evaluated using these criteria. Only a few Prolog dialects support modules, and their module systems tend to 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 more 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, and make the program hard to maintain and more difficult to debug due to the inapplicability of techniques such as declarative debugging (see e.g. [17, 22, 25]).

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 the Prolog system's typically 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*, is designed explicitly to support teams of programmers working on large application programs. It has a modern module system, detects a large fraction of program errors at compile time, and has an implementation that is both efficient and portable. To ensure that programmers can actually enjoy the benefits claimed for logic programs, Mercury has no non-logical constructs that could destroy the declarative semantics that gives logic programs their power. Mercury thus represents a clean break with previous logic programming tradition; its purity makes it is necessarily incompatible with existing logic programming languages such as Prolog.

Mercury's polymorphic type system, modelled after ML's [19], is very expressive. The type system uses declarations supplied by the programmer to detect a large fraction of program errors. The type system is also the basis for a novel strong mode system that lets the compiler prevent errors such as floundering and allows the omission of the occur check without compromising soundness. The mode system is in turn the basis for a strong determinism system that 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 will 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.

The Mercury compiler is written in Mercury itself. (Initially we used NU-Prolog and SICStus Prolog to execute the compiler, but this is now unnecessary except as backup since we have succeeded in bootstrapping the compiler.) Our experience with using the language has been entirely positive. The module system has proven that it can do its job by letting us cooperate in writing the compiler without stepping on each others' toes, and we have found that the compiler's type, mode and determinism checkers find the vast majority our errors. The strong type system is more effective than the type systems of languages such as Modula-2 because of its polymorphism and the absence of unchecked conversions. Many bugs that would be classified logic errors in imperative languages or in Prolog turn out in Mercury to be mode errors or determinism errors that can be detected by the compiler, leaving only a small minority of real logic errors to chase down using manual debugging methods. This explains how we were able to create more than 100,000 lines of Mercury code in 27 months with about 5 person-years of effort, These figures indicate sustained productivity at the level of about 400 lines of code per person per week, which is excellent, especially for a research project.

The Mercury execution model is significantly simpler than previous models for logic programming languages; for example, it does not have full unification, and it never does any dereferencing or trailing. The Mercury compiler generates code that follows this execution model. We have tested the performance of the code generated by the compiler on a suite of standard logic programming benchmarks. We have found it to have much higher performance than all the logic programming implementations we have access to. Mercury is almost twice as fast as Aquarius Prolog, more than five times as fast as SICStus Prolog fastcode, and about ten times as fast as Quintus Prolog or the wamcc system. The fastest of these Prolog systems achieve their speed by directly generating native code, and since significant effort is required to retarget them for different architectures, they are available on only a few platforms. The Mercury compiler avoids this problem and achieves portability by generating C code.

The Mercury execution model relies on the guaranteed presence and guaranteed correctness of the information derived from type, mode, and determinism declarations. A language without declarations could use program analysis to obtain the same information and hence could use the same execution model. However, to the best of our knowledge no-one has yet demonstrated a mode inference algorithm that is guaranteed to find accurate mode information for every predicate in the program. (Building such an algorithm is very difficult, not least because of the probable presence of errors in any large program.) In any case, our choice of declarations over inference is dictated mainly by software engineering concerns.

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, higher-order predicates, and polymorphism. Section 4 compares our execution model with the standard logic programming implementation model, the Warren Abstract Machine or WAM. Section 5 describes some optimizations and shows how Mercury handles I/O. Section 6 gives the current state of the Mercury system while section 7 presents performance results.

2. The Mercury language

Syntactically, Mercury is similar to Prolog with additional declarations, partly because Prolog syntax is standard in the logic programming community and partly because this made it simple to execute Mercury programs using Prolog systems early in our project. Semantically, however, Mercury is very different from Prolog. Mercury is a pure logic programming language with a well-defined declarative semantics. Like Gödel [14], Mercury provides declarative replacements for Prolog's non-logical features. Unlike Gödel, Mercury does not retain *any* non-logical features; in Mercury even I/O is declarative.

Mercury is designed to appeal to at least two groups of programmers. One group is those with backgrounds in imperative languages such as C who are looking for a higher level and more expressive language. The other group 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. This section gives a brief overview which provides the background required for the rest of the paper; sections 3.7 and 5.1 introduce some further aspects of the language. For more information about Mercury, see the Mercury home page at `http://www.cs.mu.oz.au/mercury/mercury.html`.

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 [21], and very similar to the type system of Gödel [14]; the major difference from Gödel is that Mercury supports higher-order predicates (see section 3.7). We borrow our syntax from the NU-Prolog [31] 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 variables occurring on the right hand side of a type definition also occur on the left hand side. (This "transparency" condition is the same as in Gödel.)

In principle all types can be defined just like this, but in practice types such as integers 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 suitable language for the development of large programs.

For first-order Mercury programs, type-correctness is defined the same way as in Gödel; the extension to higher-order programs is done in a fairly natural way. Although we have not yet published any formal mathematical description of the Mercury type system, a reasonably detailed description can be found in the Mercury Reference Manual [10]. The details are not important for this paper. What is important is that the type system guarantees that type-correct programs can never "go wrong" at runtime, e.g. by passing to a predicate an actual parameter whose type is not an instance of the type of the corresponding formal parameter. Since the compiler rejects programs which are not type-correct, the execution algorithm need only deal with type-correct programs.

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 similar type systems (see e.g. [21, 23]).

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 [28], 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 out_listskel :: free -> listskel.
:- mode in_listskel :: listskel -> listskel.
```

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 [6]. 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, one 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. 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 bindings of those argument terms will in turn be approximated by the final instantiatedness trees of the mode declaration and all whose free variables will be distinct. 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.

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 definition 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 the definition 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.

In Prolog systems that have them, mode declarations are *descriptive*. In those systems, their equivalent of an "in" mode states that when execution arrives at a call to this predicate using the usual left-to-right computation rule, this argument will be bound. If in some call the argument is not bound, the program usually dumps core. In Mercury, mode declarations are *prescriptive*. An "in" mode means that the compiler must make sure that the argument is bound at all calls, reordering some conjunctions of the program if necessary, and if it cannot do so, it must report a mode error. Therefore Mercury modes are declarative in a pragmatic sense; they give design-level information, i.e. which arguments are should be computed from which other arguments.

The usual definition of append is not well moded with respect to declarations such as

```
:- mode append(in_listskel, in_listskel, out_listskel).
```

because when a call to append in this mode succeeds, it establishes links between the elements of the third list and the elements of the other two lists, and this violates the "free variables must be distinct" part of the mode declaration constraint. Some programs (e.g. serialise) cannot be well moded without being rewritten because they depend on such links in an essential way. Others (e.g. the zebra puzzle) cannot be well moded because different solutions to a call leave the call's arguments in different states of instantiation. However, most logic programs can be well moded even with a significantly less expressive mode system than ours [6].

The bodies of clauses in Mercury may contain negations, disjunctions and if-then-elses; the compiler converts any predicate definition with two or more clauses into a single clause with an explicit disjunction. The type and mode analysis algorithms work with such rich clauses that have been converted into *superhomogeneous form*, where each atom is of one of the forms

$\texttt{p(X}_1\texttt{, ..., X}_n\texttt{)}$
$\texttt{Y = X}$
$\texttt{Y = f(X}_1\texttt{, ..., X}_n\texttt{)}$

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 the definition of the predicate, keeping track of the instantiatedness of each variable, and selecting a mode for each call and unification in the definition. To ensure that the mode set constraints of called predicates are satisfied, it may reorder the elements of conjunctions; it reports an error if no satisfactory order exists. Finally it checks that the resulting instantiatedness of the predicate's arguments is the same as the one given by the predicate's mode declaration. For further details we refer the reader to [13, 27, 28].

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. In fact, after mode analysis, the various modes of a predicate are treated completely independently. We call each mode of a predicate a *procedure*.

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 such *assignment unifications* 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 such *test unifications* as e.g. `Y == X`.

- instances of `Y = f(`X_1`, ..., `X_n`)` where Y is output and the X_is are either input or void (i.e. this is their only appearance in the predicate); we indicate such *construction unifications* as `Y := f(`X_1`, ..., `X_n`)`.

- instances of `Y = f(`X_1`, ..., `X_n`)` where Y is input and the X_i are output or void; we indicate such *deconstruction unifications* as `Y == f(`X_1`, ..., `X_n`)`.

(These classifications are similar to the ones used in [16].) For other instances of unification we call an out-of-line procedure whose code is automatically generated 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`. Our execution model allows both approaches; see section 3.8.

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 [13, 27, 28].

2.3. Determinism

For each mode of a predicate, the programmer should categorise that mode according to the maximum number of solutions it can produce (zero, one, or more than one) and whether or not it can fail before producing its first solution:

- If all calls to a particular mode of a predicate have exactly one solution, then that mode of the predicate is *deterministic* (`det`).

- If all calls to a particular mode of a predicate have at most one solution but may have none, then that mode of the predicate is *semideterministic* (`semidet`).

- If all calls to a particular mode of a predicate have at least one solution but may have more, then that mode of the predicate is *multisolution* (`multi`).

- If some calls to a particular mode of a predicate have no solution but other calls may have more than one solution, then that mode of the predicate is *nondeterministic* (`nondet`).
- If calls to a particular mode of a predicate cannot succeed but may fail, then that mode of the predicate has a determinism of `failure`.
- If calls to a particular mode of a predicate can neither fail nor succeed (i.e. they either loop forever or abort), then that mode of the predicate has a determinism of `erroneous`.

The last two are rarely used.

Whenever the programmer declares a mode for a predicate, Mercury requires that the declaration also state the determinism of that mode. For example, here are the declarations of the predicate member/2.

```
:- pred member(T, list(T)).
:- mode member(out, in) is nondet.
:- mode member(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 [5, 26, 27], and they work well in practice. The algorithm must be conservative; it must never underestimate the maximum number of solutions of a predicate, and it must never report that a procedure always has at least one solution if it can fail without producing a solution.

Our algorithm [12] looks for disjunctions in which all arms contain a test unification of the same variable against different function symbols. We call such disjunctions *switches*, since only one arm of the disjunction can possibly be executed. If some arms of a switch contain further disjunctions, the compiler will try to turn them into switches too. Because it works with disjunctions and not clauses, our algorithm for detecting switches is not affected by programmers moving unifications into or out of clause heads.

Our determinism analysis algorithm also looks for goals which have no output variables, or whose output variables are all locally existentially quantified; the compiler ensures that such goals will succeed at most once. Mercury does not have (or need) any non-logical explicit pruning constructs.

Determinism is a declarative concept. The declaration

```
:- pred max(int::in, int::in, int:out) is det.
```

states that for any combination of values of X and Y, there is exactly one value of Max for which max(X, Y, Max) is true. (This example shows that the pred and mode declarations can be coalesced if a predicate has only one mode, which is frequently the case.) However, the compiler's code generator wants to know whether calls to max can succeed more than once, which is a subtly different concept. In most cases, this conflict is avoided by the determinism analysis algorithm of the compiler, which transforms the internal form of the program so that the declarative and procedural interpretations will agree. In the remaining cases, such as the definition below, where the algorithm is not smart enough to perform such a transformation,

```
max(X, Y, X) :-
        X >= Y.
max(X, Y, Y) :-
        X < Y.
```

the compiler reports an error, and the programmer must either use a less precise determinism declaration, or modify the program to avoid the conflict (e.g. by using an if-then-else).

Our current 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 call 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 called 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 `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 all 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.

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

3. The Mercury execution algorithm

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, multideterministic or nondeterministic. The code generator takes advantage of this fact and uses three specialized execution algorithms, one for deterministic procedures, one for multi- and nondeterministic procedures, and one for semideterministic procedures. Sections 3.3 to 3.5 describe these algorithms. But before we go onto these algorithms, we start with a discussion of our target language and a description of Mercury's approach to data representation.

3.1. The target language

We want our Mercury implementation to be fast yet portable. Our speed objective is incompatible with bytecode 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 labels defined in one C function visible in other C functions by emitting `.globl` directives and labels into the assembly code, using statements such as `__asm__(".globl entry_append");` and `__asm__("entry_append:");`
- the ability to make direct use of the machine registers using global register variable declarations such as `register int r1 __asm__("s1");`

These extensions are a standard part of gcc and should therefore be available on all gcc ports.

Several existing systems also generate C code. The logic programming systems that fall into this category include Turbo Erlang [8], wamcc [4], Janus [7] and KL1. These also exploit various combinations of GNU C extensions. As far as we know, our compiler is the first to emit C code that, with appropriate definitions of macros, can be compiled either to exploit or not to exploit each particular GNU C extension. When not using any GNU C features, the macros yield portable ANSI C.

When not exploiting GNU C's nonlocal gotos, the macros turn each labelled piece of code into a function; they also turn abstract machine gotos into return statements that tell a driver program which function to call next. Since this technique, which is a standard trick of threaded implementations, hides the flow of control, all our examples will use nonlocal gotos.

[1] According to the GNU C manual, "totally unpredictable things will happen" if computed gotos jump to code in a different function, as they can in the code we generate. However, considerable inspection of the gcc source code by our gcc internals expert shows that in most circumstances we can use such jumps quite safely provided we take suitable precautions, which we do. The most important precautions are (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 x86, which is very short on registers) has confirmed our assessment. For more details, please refer to [11].

When exploiting GNU C's nonlocal gotos, one must decide how much code should be inside each C function. The Mercury compiler has an option whereby the programmer can request that everything in a Mercury module should be put inside one C function. This is good for speed, since (depending on which combination of GNU C extensions is in use) jumps within a C function may be faster than jumps between C functions. However, C compilers are likely to behave badly on large functions. This is why the default behaviour of the Mercury compiler is to emit several C functions for each Mercury module, with each C function containing up to a configurable (but usually small) number of Mercury procedures.

Using GNU C nonlocal gotos to jump between different C functions requires code in each function to know the addresses of labels in other functions. This can be done in one of two ways. First, one can arrange for each C function to start with code that takes the addresses of its own labels, puts them into global variables, and then returns. Cross-function jumps can get the addresses they need from these variables after an initialization routine has called all these functions at startup. This what we do in the absence of assembler labels. The second approach is of course to generate the `.globl` directives and labels directly into the assembly code.

When they are available, we exploit global register variables by using them to store the most frequently used abstract machine registers. The number of registers we can use for this depends on the hardware architecture and the configuration of gcc. To simplify calls to C library functions, at the moment we exploit only registers that gcc designates to be callee-save. We use 3 global register variables on x86s, 6 on Alphas, 8 on MIPS and HP-PA processors, and 10 on SPARCs.

3.2. Data representation

Since we know all types at compile time, we 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. (On 64-bit machines such as the Alpha, Mercury can use up to three tag bits, giving eight tag values, but the discussion below assumes four.)

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. 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 identity of the function symbol is stored in the low-order bits of the pointer to the cell, not in an extra word at the start of the cell. The difference is that in our model, this scheme 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. If all function symbols sharing the same primary tag value are constants, we use the rest of the word as a 30-bit local secondary tag. If at least one of them is not a constant, we store an extra word at the start of the argument block as a 32-bit remote secondary tag. To save both space and time, we prefer to share a primary tag value only between several constants or several non-constants. 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. We don't need to box values of primitive types that fit in a word.

Relatively few types in real programs need secondary tags. For simplicity of presentation, we do not present any such types in this paper.

The data representation we have just described is our preferred one, but it does not work on word-addressed machines, since they do not have spare low-order bits. This is not a problem since our model can use two other representations as well: one stores the tag in the most significant bits of a word, while the other does not use tag bits as such but instead always uses a full word as a remote tag. Depending on command-line options, the Mercury compiler can emit code that uses any of these representations. Our preferred representation uses the following tag-manipulation macros:

```
#define mktag(t)          (t)
#define mkbody(i)         ((i) << TAGBITS)
#define tag(w)            ((w) & ((1 << TAGBITS) - 1))
#define body(w, t)        ((w) - (t))
#define mkword(t, p)      ((Word)(t) + (Word)(p))
#define field(t, p, i)    ((Word *) body((p), (t)))[i]
```

The macros mktag and mkbody position an integer in the tag and body parts of a word respectively. The tag macro extracts the tag of a word, while body extracts the non-tag part. The mkword macro puts the tag t on the body b, which must either be an aligned pointer or have been constructed by mkbody. The field macro 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 almost 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 in general be represented differently for each type. The type checker will determine the type of each occurrence of the function symbol. The different representations do not cause any problems, since terms of different types cannot be compared directly.

3.3. 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 can grow. The order of these areas in the address space does not matter, nor does the direction of growth of the stack and the heap. For this paper we assume that both the heap and the stack grow upward.

The abstract machine 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.

At the start of execution, the runtime system sets the stack pointer and the heap pointer to point to the start of their respective data areas, sets the succip register to point to code that will terminate execution of the abstract machine, sets up the registers to reflect the input arguments of the main predicate, and jumps to the code of this predicate.

Consider the call `append([1,2], [3], L)`. This calls append in its first (forward) mode, whose form after mode analysis is:

```
:- pred append(list(int), list(int), list(int)).
:- mode append(in, in, out) is det.

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

(We postpone consideration of polymorphism until section 3.8.) 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.

```
append_3_0:
        incr_sp(2);
        detstackvar(2) = succip;
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_0_i2;
        detstackvar(1) = field(mktag(1), r1, 0);
        r3 = r1;
        r1 = field(mktag(1), r3, 1);
        localcall(append_3_0, append_3_0_i3);
append_3_0_i3:
        tag_incr_hp(r1, mktag(1), 2);
        field(mktag(1), r1, 0) = detstackvar(1);
        field(mktag(1), r1, 1) = r3;
        r3 = r1;
        goto append_3_0_i1;
append_3_0_i2:
        r3 = r2;
append_3_0_i1:
        succip = detstackvar(2);
        decr_sp(2);
        goto *succip;
```

FIGURE 3.1. C code for the forward mode of append, unoptimized

Figure 3.1 shows the code emitted by the Mercury compiler for this mode of append without optimization. For clarity of exposition we have turned off the compiler option that automatically generates a comment for each line of C code (!), and we have changed the actual code generated by the compiler in minor ways: we have shortened label names, eliminated cast operators, expanded several macros, and removed extern declarations and their braces.

Each label begins with the name of the predicate, followed by the arity of the predicate and the number of the mode concerned (in this case mode 0 is the forward mode). For the label that starts an abstract machine procedure, this is all. Labels internal to a predicate also have a label number preceded by "i".

The procedure prologue invokes the incr_sp macro to create a stack frame containing two words, and saves the success continuation in the bottom word. It then tests whether A is equal to nil or not; if it is, we branch to append_3_0_i2. When using both global registers and nonlocal gotos, gcc will compile the entire if statement into a single conditional branch instruction, since the right hand side expression is the constant zero. The code at append_3_0_i2 implements the base case. It copies B in r2 to r3, where the caller will look for C. It then falls through to the epilogue code that restores the success continuation from the stack, and then jumps to the success continuation after popping the stack frame.

The code after the if statement implements the recursive case. 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; the mode system guarantees that A is not an unbound variable. Therefore the code does not need to check to know that A has the cons tag and that the body points to a two-word cell containing H and T. 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 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. (Without optimization this takes two instructions because the code generator does not know that A is dead after the move, and therefore attempts to save its value in r3.) The other input argument of the call, B, is already in r2. The localcall macro puts the address of the append_3_0_i3 label into the succip register and then branches to the append_3_0 label to begin execution of the recursive invocation of append. (There is a separate macro for calling predicates defined in other C functions. When using GNU C's nonlocal gotos without assembler labels, this other macro takes the address to be branched to from a global variable associated with the called predicate. At startup, each C function assigns the addresses of the Mercury procedures within it to the global variables associated with them.)

When the recursive invocation has been completed, execution will continue at append_3_0_i3, at which time r3 will contain the value of NT. The tag_incr_hp macro increments the heap pointer by two words, tags the old value of the heap pointer with the tag 1 (for cons) and assigns the tagged value to r1; r1 now points to a new two-word cons cell on the heap. The next two assignments fill in the first word in this cell with the value of H from the stack and the second word with the value of NT in r3. The assignment to r3 puts the tagged pointer to the cons cell, i.e. C, where the caller expects it to be, and the goto branches to the procedure epilogue.

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.

The low-level optimizer in the Mercury compiler turns the code in figure 3.1 into the code in figure 3.2. The optimizer can and does find out that the base case doesn't contain any calls and hence does not need a stack frame. It therefore delays the creation of the stack frame until execution enters the code of the recursive case; this way the base case does not have to destroy the stack frame. However, on most current RISC machines branch instructions have a delay slot, an instruction after the branch instruction that is executed whether or not the branch is taken. The store instruction that saves succip on the stack can be put into the delay slot of the branch instruction, but the C compiler will only do this if it knows that the store can be done in both branches. In situations like this where there is nothing else

```
append_3_0:
        detstackvar(0) = succip;
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_0_i1000;
        incr_sp(2);
        detstackvar(1) = field(mktag(1), r1, 0);
        r1 = field(mktag(1), r1, 1);
        localcall(append_3_0, append_3_0_i3);
append_3_0_i3:
        tag_incr_hp(r1, mktag(1), 2);
        field(mktag(1), r1, 0) = detstackvar(1);
        field(mktag(1), r1, 1) = r3;
        r3 = r1;
        succip = detstackvar(2);
        decr_sp(2);
        goto *succip;
append_3_0_i1000:
        r3 = r2;
        goto *succip;
```

FIGURE 3.2. C code for the forward mode of append, optimized

to do before the if statement, the Mercury optimizer therefore puts the assignment before the if statement. By allowing the C compiler to put the store in the delay slot, this speeds up the recursive case and does not harm the base case. The stack offset of the store has to be adjusted to account for its timing (before the increment of the stack pointer); after the creation of the stack frame by the incr_sp macro, detstackvar(2) will refer to saved copy of succip.

The reason why the code generator reorders the two arms of the switch even without optimization is also related to pipelining. On most current machines taken branches cause pipeline breaks and thus can be significantly more expensive than untaken branches. Since the recursive case is more likely to be executed, the code generator puts it first. (In general the compiler may reorder the arms of disjunctions because Mercury is a pure language and programmers may not depend on the order of solutions. However, this consideration is not required here, since exactly one of the base and recursive cases will be executed.)

The optimizer has replaced the goto at the end of the recursive case with the code at the target of the goto, the procedure epilogue, eliminating the jump and the pipeline disruption it would cause. Although this optimization may increase the size of the object code, the cost is minimal in this instance, since the base case uses a cut-down version of the epilogue (one that does not touch the stack) created by the previous optimization.

The last optimization is the removal of the redundant copy of r1 into r3 just before the recursive call. This makes the code smaller as well as faster.

In the following sections we will show optimized code because it is shorter and frequently clearer to humans.

3.4. The nondeterministic execution algorithm

The execution algorithm of the previous section does not work for multisolution and nondeterministic predicates because execution may backtrack into such predicates. When that happens, continued execution requires the values of the variables occurring in the predicate, 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 whose frames are popped not on success but on failure. Like our other dynamic areas, this *nondet stack* can grow in any direction and its placement in the address space does not matter. The new stack is addressed by two new abstract machine registers:

curfr Points to the nondet stack frame of the currently executing nondeterministic 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 alternative fails; points to the code for the next alternative 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 alternatives (input arguments) or between calls in an alternative (variables local to an alternative).

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, as it can be recalculated as frameaddr - prevfr - 4 words if necessary. The disadvantage is that it is very expensive to change the size of the frame after the frame has been set up. However, we feel that the time cost of changing the size of a stack frame would be unjustified even if we used the frame layout best suited for this. Therefore we never adjust the size of frames on either stack after setup. To reduce stack usage we instead use a graph-colouring algorithm for stack slot allocation on both stacks.

As an example of the execution algorithm for nondet and multidet predicates (which for brevity's sake we will lump together as nondet predicates in the rest of the paper), consider the call `append(A, B, [1,2])`. This invokes append in its backward mode, whose form after mode analysis is:

```
:- pred append(list(int), list(int), list(int)).
:- mode append(out, out, in) is multidet.

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

(Note that mode analysis rearranges the two modes of append in different ways; compare the above with the internal form of the forward mode of append from section 3.3.) For this Mercury code the compiler generates this C code:

```
append_3_1:
        mkframe(2, append_3_1_i2);
        mark_hp(framevar(1));
        framevar(0) = r3;
        r1 = mkword(mktag(0), mkbody(0));
        r2 = r3;
        succeed();
append_3_1_i2:
        restore_hp(framevar(1));
        modframe(do_fail);
        r1 = framevar(0);
        if (r1 == mkword(mktag(0), mkbody(0)))
                fail();
        r3 = field(mktag(1), r1, 1);
        framevar(0) = field(mktag(1), r1, 0);
        localcall(append_3_1, append_3_1_i3);
append_3_1_i3:
        tag_incr_hp(r3, mktag(1), 2);
        field(mktag(1), r3, 0) = framevar(0);
        field(mktag(1), r3, 1) = r1;
        r1 = r3;
        succeed();
```

Following the pattern of all nondeterministic procedures, the code of append_3_1 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 arguments give the number of framevar slots to reserve in the frame and the initial value of the redoip slot. 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 the current value of the heap pointer in framevar 1 and the current value of C in framevar 0 for use by later alternatives. The first alternative consists of `A := []` and `B := C`, so two assignments implementing these unifications follow next. At the end of the alternative (as at the end of every alternative) 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.

Sometime later some other part of the program will want to find the next solution, which it does by executing the redo macro. 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.

Eventually the nondet stack frame we created in this invocation of append will become the top nondet stack frame. When that happens, an invocation of the redo macro will set curfr to point to this frame and execution will continue at append_3_1_i2. The code there starts by invoking the restore_hp macro to recover all the space allocated since the start of the previous alternative. The mark_hp and restore_hp macros usually just copy the heap pointer to and from the named locations, but they and incr_hp are implemented as macros to allow us to explore other memory management strategies as well. For example, by redefining incr_hp to call gc_malloc, we can avoid the need for our own garbage collector and simply link in Hans Boehm's conservative garbage collector for C [2]. In such a setup, we could put our own wrappers around gc_malloc and gc_free to allow a restore_hp to free all blocks allocated since the corresponding mark_hp without invoking the general garbage collector, but this has quite high overhead during forward execution. A simpler solution is to define mark_hp and restore_hp to do nothing and let the garbage collector recover all memory. Of course, if the Mercury compiler knows this will happen, it should not reserve a framevar slot for the saved hp. To reduce C compilation time, it should also avoid emitting the mark_hp and restore_hp macros.

The code for the second alternative continues after the restore_hp by updating the redoip slot via the modframe macro. Since we are entering the last alternative, the new value of redoip should cause this call to fail if it is ever asked for more solutions. Do_fail is a global variable that contains the address of a label in the Mercury runtime system. 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.

The next statement copies the saved value of the argument C from framevar slot 0 into the 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 alternative does not need it or if the value is used only once. 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.

Even though the source for the second alternative contains a test of C against cons, the compiler generates code that compares C to nil, since this does not require masking off of the non-tag parts of r1. If C is nil, the second alternative cannot succeed, so the code invokes the fail macro directly. (A redo would invoke the fail macro indirectly through the redoip slot, which is slightly slower.) If C is not nil, we extract H and NT from the cons cell C 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. Since the lifetimes of C and H do not overlap, the compiler uses the same framevar slot for them. The recursive call may succeed several times. In our example, the value of NT will be [2], and the recursive call will succeed twice, the solutions being T = [], B = [2] and T = [2], B = []. When the recursive call returns one of these solutions, the code at append_3_1_i3 will create a new cons cell containing H (which is 1) and T and return the result as the value of A in r1, together with the untouched value of B in r2. Therefore the second alternative succeeds twice with the solutions A = [1], B = [2] and A = [1,2], B = [].

Suppose we wanted to discard nondet stack frames as soon as possible. Since we cannot leave a hole in the nondet stack, before we could discard this frame we would have to check whether this frame was on top of the stack or not (the test is simply curfr == maxfr). The problem is that although this test is cheap, so is an invocation of the fail macro, and the test will fail some of the time. For example, after returning the solution A = [1,2], B = [] the frame of the original invocation of append_3_1 will be on top of the nondet stack, but after returning A = [1], B = [2] it will not be. Therefore in this case leaving the frame on the nondet stack until failure is faster than trying to discard it on the last success, and this is what the code above does. The current Mercury compiler discards the nondet stack frame on the last success only if the last alternative contains no nondeterministic code, because only in this case can the runtime test be avoided.

3.5. 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 nondet 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 success/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 the general purpose register r1. This requires argument i to be stored in register $\mathrm{r}n+1$, which makes the calling convention for semidet predicates different from the calling convention for det or nondet predicates. This is not a problem, however, since the code generator always knows the determinism of the predicates it is dealing with.

3.6. 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. This effect can increase the runtime of a program by an arbitrarily large factor; if the goals checked for groundness include large terms, the checks can be prohibitively expensive.

The real requirement for the safety of a negated goal is that the negated goal not export any bindings to the rest of the computation. The Mercury mode system can ensure this at compile time, removing the need for any runtime checks. The mode system also allows increased flexibility, by allowing the negated goal to contain unbound variables that are instantiated by the goal, as long as these variables are not visible outside the negation. For example, if one wants to test whether two lists are disjoint, one may use the goal `not (member(E, Xs), member(E, Ys))`, where the variable E occurs only inside the negation.

The rules for if-then-elses are somewhat different. Since `(`*Cond* `->` *Then* `;` *Else*`)` is logically equivalent to `(`*Cond*`,` *Then* `;` `not` *Cond*`,` *Else*`)` the condition may export its bindings to the then part of the if-then-else, but not to the else part or to the rest of the computation.

For an example of the code generated for negations and if-then-elses (the two are closely related), consider the predicate addelement, whose Mercury code is shown in figure 3.3, together with the C code generated from this by the compiler.

Since the body of addelement contains a call, we start by saving the succip register. The compiler doesn't know that member cannot allocate space on the heap, so next we save the current heap pointer. 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 copy List0 from the stack into r3, returning it as the value of List.

```
:- pred addelement(int, list(int), list(int)).
:- mode addelement(in, in, out) is det.

addelement(Elem, List0, List) :-
        ( member(Elem, List0) ->
                List = List0
        ;
                List = [Elem | List0]
        ).

addelement_3_0:
        incr_sp(4);
        detstackvar(4) = succip;
        mark_hp(detstackvar(3));
        detstackvar(1) = r2;
        detstackvar(2) = r1;
        r3 = r2;
        r2 = r1;
        localcall(member_2_0_1, addelement_3_0_i2);
addelement_3_0_i2:
        if (!r1)
                goto addelement_3_0_i1;
        r3 = detstackvar(1);
        succip = detstackvar(4);
        decr_sp(4);
        goto *succip;
addelement_3_0_i1:
        restore_hp(detstackvar(3));
        tag_incr_hp(r3, mktag(1), 2);
        field(mktag(1), r3, 1) = detstackvar(1);
        field(mktag(1), r3, 0) = detstackvar(2);
        succip = detstackvar(4);
        decr_sp(4);
        goto *succip;
```

FIGURE 3.3. Mercury and C code for addelement

If member fails, we branch to addelement_3_0_i1, where we recover any storage member may have allocated on the heap; storage allocated by failed computations will never be used. We then 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 compiler can recover the space allocated on the heap even by a successful computation if that computation has no output variables; our compiler does not yet do so. In this case, however, it would be better to notice that member cannot possibly allocate any new storage on the heap, and therefore both the mark_hp and restore_hp instructions and the framevar they refer to can be optimized away. We have not yet implemented this optimization, primarily because in the general case it requires information about predicates defined in other modules.

The condition of an if-then-else may be nondeterministic; 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 generates code that 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. (The failure continuation of a frame can be reached only if that frame is on the top of the nondet stack, so a similar restoration in the failure continuation would not only not be able to prune away any frames but may also put back some frames that have already been removed.)

The compiler generates code to discard unnecessary frames from the nondet stack whenever a multi- or nondeterministic computation has no outputs, not just in the conditions of if-then-elses. The only difference between the treatment of multideterministic and nondeterministic predicates by the code generator is that if a multideterministic computation has no outputs, the generated code does not need and thus does not have a failure continuation.

Our implementation of negation is very similar to the implementation of if-then-else. This is not surprising, since one can transform goals such as

```
not member(Elem, List0)
```

into

```
( member(Elem, List0) -> fail ; true )
```

3.7. Higher order predicates

Mercury supports higher-order predicates; that is, a predicate can be passed as an argument to another predicate. Actually, it would be more accurate to say that Mercury supports higher-order procedures, since you can only pass a single mode of a predicate; if you want multiple modes, you have to pass multiple higher-order procedure terms. A higher-order predicate term (sometimes known as a "closure") can be the name of a predicate, the name of a predicate with some of the initial arguments supplied (currying), or a lambda expression. The family of built-in of predicates `call/1`, `call/2`, ... are used to call higher-order predicate terms.

For example, the following predicate takes as input a predicate and a list, and applies that predicate to all the elements of the list to produce a new list.

```
:- pred map(pred(T1, T2), list(T1), list(T2)).
:- mode map(pred(in, out) is det, in, out) is det.
```

```
map(Pred, [], []).
map(Pred, [X | Xs], [Y | Ys]) :-
        call(Pred, X, Y),
        map(Pred, Xs, Ys).
```

If you have defined a predicate called double,

```
:- pred double(int::in, int::out) is det.

double(X, Y) :- Y is 2 * X.
```

you can pass it to map like this:

```
        map(double, [1,2,3], List)
```

This goal will bind List to [2,4,6]. The same effect can be achieved through a lambda expression:

```
        Double = lambda([X::in, Y::out] is det, Y is 2 * X),
        map(Double, [1,2,3], List)
```

The compiler implements lambda expressions by translating them into auxiliary predicates, so the above two examples would result in equivalent object code.

Since Mercury supports currying, programmers can specify the first few arguments when constructing higher-order predicate terms. For example, given

```
:- pred multiply(int::in, int::in, int::out) is det.

multiply(X, Y, Z) :- Z is X * Y.
```

the three goals

```
        map(multiply(2), [1,2,3], List)

        Multiply = multiply,
        Double = call(Multiply, 2),
        map(Double, [1,2,3], List)

        Factor = 2,
        Double = lambda([X::in, Y::out] is det, Y is Factor * X),
        map(Double, [1,2,3], List)
```

are three more ways of doing the same thing. As shown in the last example, lambda expressions may refer to variables such as Factor which occur outside of the lambda expression; when transforming a lambda expressions to an auxiliary predicate, the compiler passes any such non-local variables as additional curried arguments to the introduced predicate.

The compiler represents higher-order predicate terms as a pointer to a structure containing the address of the procedure to call, the number of curried arguments (if any), and then the curried arguments. To call a higher-order predicate term, the compiler generates a call to a piece of code in the Mercury runtime system which positions the curried arguments and the input arguments to the `call` builtin in the appropriate registers, calls the procedure, and then positions the output arguments in the appropriate registers.

Higher-order predicates are considerably more efficient in Mercury than in existing Prolog systems, because the address of the procedure to call is determined at compile/link time, not at runtime, and higher-order predicate terms are always compiled, never interpreted.

The Mercury compiler has an optimization that looks for calls to higher-order predicates in which the higher-order argument terms have a known value (e.g. all the calls to `map/3` above), and generates a version of the called predicate specialized for these argument values. In many important cases this can eliminate all the penalties usually associated with higher-order predicates, which means programmers can use them much more freely.

3.8. Polymorphic predicates

One consequence of specializing the data representation of each type is that implementing polymorphic predicates becomes a little more complicated. Our type system allows polymorphic predicates to do three things with values whose type is not known until runtime: to copy such values, to unify two such values, and to compare two such values for less than, equal or greater than. In all three cases, the values concerned must be ground.

Copying is not a problem, since in our execution model values of all types are the same size (with one exception; see below). To handle unifications and comparisons, the compiler introduces an extra argument for each type variable in the type declaration of the predicate in question, giving information about the value of the type variable. Calls that instantiate type variables set these arguments to point to structures containing information about the types being assigned to the type variables. In the current implementation, this `type_info` structure contains the addresses of the `unify/2` and `compare/3` procedures for the type. These are ordinary predicates whose definitions are automatically generated by the compiler from the type definition, as we have shown in an example in section 2.2. (We could implement unification as a comparison followed by a test for equality, but the current version of the compiler can generate somewhat better code for specialized unification procedures.)

When a predicate p1 passes an argument of type list(tree(int)) to a predicate p2 that expects an argument of type list(T), it passes a type_info structure describing the type tree(int) to p2, which then uses it whenever it wants to manipulate values of type T. If p2 wants to pass a value of type list(T) to a predicate p3 that expects a value of type U, where U is another type variable, p2 must package up the type_info for T into the type_info for U. Therefore type_info structures can contain pointers to other type_info structures, and the unification and comparison routines of one type_info structure will invoke the unification and comparison routines of other structures when necessary. The way we do this is similar to the way we handle currying and closures for higher-order predicates (see section 3.7).

Many polymorphic predicates do not need these additional arguments. We have therefore implemented an optimization that looks for predicates that have arguments that are used only in recursive calls, creates a specialized version of the predicate without the argument, replaces the original predicate definition with a call to the specialized version, and adjusts all calls to the predicate accordingly. (The original predicate interface may still be needed if the predicate is exported from the module or if it is referred to from a higher-order construct.)

This optimization eliminates any unneeded type_info structures, and sometimes eliminates unnecessary arguments put there by the programmer. The compiler generates a warning in the latter case, even if the eliminated argument appears more than once; this can happen if the argument appears in the clause head and in recursive calls but nowhere else.

We could optimize polymorphic predicates that do need the type_info structure by creating specialized versions of the C code for the required combinations of the values of type variables. Since only some combinations will have a significant impact on runtime, this optimization could clearly benefit from feedback from profiling runs to identify these combinations. We have not yet implemented this optimization, but specialization is certainly one possible way of handling double-precision floating-point numbers, which do not fit into 32-bit words. The other way is to pass around pointers to such numbers, which is what our current implementation does.

4. Comparison with the WAM

It is interesting to compare our execution model with the Warren Abstract Machine [1, 34]. 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 the absence of polymorphism, or if the polymorphism overhead is eliminated through specialization, such predicates are the fastest way to perform such unifications.

Indexing is much simpler in our model. We do not need operations such as switch_on_term because the mode system tells the compiler exactly which parts of which arguments will be bound at each point in the program. We can generate smaller and faster code because we do not need to prepare for the possibility of having to execute all alternatives. Strong typing and our specialized data representation let us use dense jump tables 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 can usually avoid this test because 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 whether they contain disjunctions after indexing and the nature of the calls they contain:

- predicates with no disjunctions containing calls only to deterministic predicates
- predicates with no disjunctions containing some calls to nondeterministic predicates
- predicates with disjunctions

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. (The designers of NTOAM [36] also merge environments and choice points into one structure, but their technique and their motivation have little in common with ours.)

Last call optimization is easier with the WAM. Some 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 must reorder the superhomogeneous form of the conjunction 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 construction 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 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 in cases where a cell construction follows the last call; section 5.2 shows how to do this. That section also introduces a recursion optimization that works just fine with the 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. When compiling the Mercury compiler, the Mercury compiler recognizes about 9% of calls as last calls. By contrast, with its more general notion of LCO, the NU-Prolog compiler turns about 24% of all calls into last calls (these counts are static counts). Therefore Mercury finds about one third of the last call opportunities Prolog does.

Our execution model does not need a trail. 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.

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.

Our execution model does not need to box integers and floats. Since we don't need to reserve tag bits for indicating whether a variable is bound or not and what its type is, we can use the full size of the word for storing data. This means that our model can store values of primitive types such as integers and single-precision floating-point numbers in their native format. This avoids loss of precision to the tag bits, and the absence of boxing and unboxing code speeds up arithmetic to the level where it is exactly as fast as in imperative languages. On 32-bit machines we do have to box double-precision floating-point numbers because they do not fit into a single word. (Our current implementation boxes all floats, i.e. represents them as a pointer to a memory location containing the number, because this allows the compiler to treat all registers as equivalent. This restriction will go away in the future.)

Our execution model relies much less on global analysis for good performance. High performance Prolog systems must perform global analysis to find out whether they can omit a particular trailing or dereferencing operation and whether all callers can pass unboxed integers or floats. In the Mercury execution model none of these operations is ever required. A compiler for the Mercury execution model may still use global analysis to gather information such as whether a particular predicate can ever allocate space on the heap or whether a particular predicate can ever clobber a particular register.

5. Optimizations

Since Mercury is a purely declarative language, 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 automatic inlining, common subexpression elimination, and the short-circuiting of jumps to jumps. For presentation in this section we have selected the optimizations that we believe to be most significant in raising the efficiency of Mercury programs to the level of programs written in imperative languages. This represents work in progress; the current Mercury implementation implements only one of these optimizations.

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/retract and setarg, or implementing an update as the creation of an almost-identical copy. Programs written in the second style have many predicates 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 [13, 18, 20, 29]. 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. Type information helps e.g. in finding out that in a tree of items, all the tree cells can be reused even though the space of the items cannot. This is of course all that is usually required for optimizing the implementation of predicates that manipulate the tree.

If reuse analysis finds that an input argument to a call is the last reference to a structure, i.e. the structure will never be needed again, the implementation of the called predicate is free to reuse the space of that structure. If reuse analysis finds that the structure will not be accessed during forward execution but may be needed after backtracking, the implementation of the called predicate is free to modify the contents of the structure, provided that the old values are restored on backtracking. The standard way to arrange for this to happen is to store the addresses and original values of any modified words on a trail, to save the trail pointer in a nondet stack frame when entering a disjunction, and to unwind the trail to this saved point when entering the second and later disjuncts. This must be done even for disjunctions in predicates that do not themselves use trailed structure reuse. In any case, since value-trailing requires two trail words for each updated word, its use improves space utilization only if reused cells are big, or if the reuse of one cell allows the reuse of other cells without change. The latter happens e.g. when one inserts a new element into a tree; with a trailed update of one node to point to the new element, the cells on the path leading to the insertion site can remain unchanged.

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 forward mode of the append predicate. 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, which will frequently be the case. The specialized version of append is in figure 5.1. This version differs from the one in figure 3.2 in that the recursive case 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.

Even given some calls to a predicate that can exploit structure reuse, it is quite unlikely that *all* calls to a given predicate will have last references to structures

```
append_3_0_sr:
        detstackvar(0) = succip;
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_0_sr_i1000;
        incr_sp(2);
        detstackvar(1) = r1;
        r1 = field(mktag(1), r1, 1);
        localcall(append_3_0_sr, append_3_0_sr_i3);
append_3_0_sr_i3:
        r1 = detstackvar(1);
        field(mktag(1), r1, 1) = r3;
        r3 = r1;
        succip = detstackvar(2);
        decr_sp(2);
        goto *succip;
append_3_0_sr_i1000:
        r3 = r2;
        goto *succip;
```

FIGURE 5.1. Append with structure reuse

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 append_3_0_sr; other calls jump to the general version append_3_0.

In some programs, structure reuse becomes so important as to be indispensable. If you are writing a number crunching program, 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 memory consumption and execution time. Mercury's mode system therefore provides a concept called *unique modes* [13], analogous to linear types in functional programming [33]. Unique modes essentially allow the programmer to declare that there should only ever be one reference to a particular object, thus guaranteeing that the compiler will be able to apply untrailed structure reuse for updates to that object. Mercury also provides a related concept, called *mostly unique modes*, which provides a weaker guarantee: that the compiler will be able to apply *trailed* structure reuse for updates to that object. Like all the other declarations in Mercury, both forms of uniqueness annotations on 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 states 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.

Unique mode analysis, which relies on determinism information, will ensure that the implementation never has to backtrack past any I/O operation.

As an example, consider the hello predicate:

```
:- pred hello(io_state, io_state).
:- mode hello(di, uo) is det.

hello(S0, S) :-
        write_string("Hello, ", S0, S1),
        write_string("world\n", S1, S).
```

The first argument of hello, S0, describes the I/O state of the computation on entry to hello, while the second argument, S, describes the I/O state of the computation on exit from hello. The modes di and uo stand for destructive input and unique output respectively. The di mode of S0 requires the caller of hello to supply as first argument an I/O state to which no other references remain; the uo mode of S promises that hello will return as second argument the unique reference to the new I/O state. In the example this new I/O state is reached in two stages, with the intermediate stage S1 reflecting the I/O state after "Hello" has been written but "world" has not.

5.2. Recursion optimizations

Most Prolog systems implement recursion optimization (usually as a special case of last call optimization). For the first, deterministic mode of append, the one discussed in section 3.3, mode reordering puts the construction unification `C := [H | NT]` after the recursive call. To exploit tail recursion optimization, we must reverse this order. The new order of the recursive case is

```
        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.

Figure 5.2 shows both the interface procedure and the tail recursive procedure. The code for the recursive case 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_3_0_tr, which will be the only call to that procedure in the program.

The base case of append_3_0_tr is different only in where it puts the output argument. The recursive case 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

```
append_3_0:
        detstackvar(0) = succip;
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_0_i1000;
        incr_sp(2);
        tag_incr_hp(r4, mktag(1), 2);
        detstackvar(1) = r4;
        field(mktag(1), r4, 0) = field(mktag(1), r1, 0);
        r1 = field(mktag(1), r1, 1);
        r3 = &field(mktag(1), r4, 1);
        localcall(append_3_0_tr, append_3_0_i3);
append_3_0_i3:
        r3 = detstackvar(1);
        succip = detstackvar(2);
        goto *succip;
append_3_0_i1000:
        r3 = r2;
        goto *succip;

append_3_0_tr:
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_0_i1000_tr;
        tag_incr_hp(r4, mktag(1), 2);
        *r3 = r4;
        field(mktag(1), r4, 0) = field(mktag(1), r1, 0);
        r1 = field(mktag(1), r1, 1);
        r3 = &field(mktag(1), r4, 1);
        localtailcall(append_3_0_tr);
append_3_0_i1000_tr:
        *r3 = r2;
        goto *succip;
```

FIGURE 5.2. Append with tail recursion optimization

sets up the arguments for the recursive call to fill in the second field of the new cons cell and performs a tail call (if debugging is not turned on, this is just a goto). Eventually control will reach the base case of append_3_0_tr, which will return to append_3_0_i3.

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

```
append_3_0:
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_0_i1;
        r5 = sp;
append_3_0_i4:
        incr_sp(1);
        detstackvar(1) = field(mktag(1), r1, 0);
        r1 = field(mktag(1), r1, 1);
        if (r1 != mkword(mktag(0), mkbody(0)))
                goto append_3_0_i4;
        r3 = r2;
append_3_0_i5:
        tag_incr_hp(r4, mktag(1), 2);
        field(mktag(1), r4, 0) = detstackvar(1);
        field(mktag(1), r4, 1) = r3;
        r3 = r4;
        decr_sp(1);
        if (sp > r5)
                goto append_3_0_i5;
        goto *succip;
append_3_0_i1:
        r3 = r2;
        goto *succip;
```

FIGURE 5.3. Append with middle recursion optimization

written inline containing the original code for `A == [H | T]` and `C := [H | NT]` respectively, with the code for `C := B` in the middle. The resulting code is shown in figure 5.3.

The code for the base case is unchanged. The code for the recursive case uses the stack pointer to count n, and therefore starts by saving the current value of sp in r5. The body of the down loop sets up the stack and registers as for a call, but instead of jumping to the start of the procedure to test r1, we perform the test directly. This replaces one unconditional jump and one conditional branch with just one conditional branch. When the condition fails, we execute the base case and fall into the up loop, which uses the same scheme to execute the code for `C := [H | NT]` as many times as the down loop executed `A == [H | T]`. The exact parallel between the two loops ensures that the up loop finds the various incarnations of H on the stack where the down loop left them, and that the comparison of sp against r5 stops the up loop at the right time.

If the recursive case contains any calls apart from 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

```
append_3_0:
        if (r1 == mkword(mktag(0), mkbody(0)))
                goto append_3_0_i1;
        r3 = r1;
append_3_0_i4:
        r4 = r1;
        r1 = field(mktag(1), r1, 1);
        if (r1 != mkword(mktag(0), mkbody(0)))
                goto append_3_0_i4;
        field(mktag(1), r4, 1) = r2;
        goto *succip;
append_3_0_i1:
        r3 = r2;
        goto *succip;
```

FIGURE 5.4. Append with destructive assignment

call is made.

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

Accessing the stack for loop counters or alternative indicators will certainly reduce the performance gain to be had from middle recursion optimization, and will in fact usually slow the program down. However, the middle recursion transformation may be worthwhile even in such cases if it 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. Consider the original code for append again:

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

With structure reuse optimization, the recursive case replaces T with NT in the cons cell of A, which it reuses as the value of C. When a recursive call reaches the

```
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;
        }
}
```

FIGURE 5.5. Append from the GNU C++ library

recursive case again, 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 recursive case, this means that in the pattern $D^n A U^n$, 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 (r3) and one assigned every time through the loop (r4). The third use of the stack, as a counter, is no longer needed. The resulting code is shown in figure 5.4. This is very close to the version that a C programmer would write from scratch. Figure 5.5 shows the code for append from the GNU C++ library, cleaned up for publication. In our code r4 in our version plays the role of tmp while r1 plays the role of tmp–>next.

These optimizations can produce similarly good code for other predicates besides append. Consider the predicate that inserts a key/value pair into a binary search tree, whose code appears in figure 5.6. 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 base case, and that the code for the recursive case is mostly a loop. In this case the loop has two termination conditions: we exit the loop if either Init is a leaf or if it is a tree node whose key is equal to Key. The loop body puts the address of Init 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 to either Left or Right depending on the result of the

```
:- pred insert(tree(int, T), int, T, tree(int, T)).
:- mode insert(in, in, in, out) is det.

insert(Init, Key, Val, Final) :-
        Init == leaf,
        Final := tree(leaf, Key, Val, leaf).
insert(Init, Key, Val, Final) :-
        Init == tree(Left, Curkey, _, Right),
        compare(Key, Curkey, Result),
        (
                Result = lt,
                insert(Left, Key, Val, Nleft),
                Final := tree(Nleft, Key, Val, Right)
        ;
                Result = eq,
                Final := tree(Left, Key, Val, Right)
        ;
                Result = gt,
                insert(Right, Key, Val, Nright),
                Final := tree(Left, Key, Val, Nright)
        ).
```

FIGURE 5.6. Mercury code for binary search tree insertion

comparison. If we exited the loop because Init is a leaf, the code following the loop allocates a new tree node, fills it with Key, Val 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, the code simply overwrites the value field of the last Init with Val.

6. Status of the Mercury system

We designed the Mercury execution algorithm in October 1993. We started working on a Mercury compiler in November 1993. Semantic analysis started working around May 1994. We started generating code around August 1994; we started work on optimizations very soon after while continuing to expand the class of predicates we generated code for. The compiler successfully compiled itself to a fixpoint on 24 February 1995. The first beta test release of the system (version 0.1) was on 8 April 1995. The first public release of the system (version 0.3) was on 18 July 1995. Version 0.4, which was released on 14 September 1995, added support for higher-order predicates and included a preliminary version of the Mercury profiler. Version 0.5, released on 15 February 1995, added checking of uniqueness annotations on mode declarations.

Since Mercury is a pure language, the compiler cannot use failure to recover memory. Without garbage collection, it can correctly compile some but not all of the 141 modules in the compiler and the standard library. To allow the compiler to compile the remaining modules, we linked our runtime system with Hans Boehm's conservative garbage collector for C [2]. This requires that we define incr_hp to call gc_malloc. We are also working on a native garbage collector for Mercury, since a compacting collector may lead to better locality.

As of version 0.5, the Mercury compiler does not yet implement structure reuse, and hence it cannot implement destructive assignment either. Although the compiler does not yet attempt to create opportunities for last call optimization and hence tail recursion optimization by adapting the argument passing convention suggested in section 5.2, it does perform last call optimization whenever the last call leaves the right values in the right registers. This happens surprisingly frequently in real programs due to the popularity of the accumulator-passing style of programming; for example, three of the ten benchmark programs from section 7 contain predicates that the compiler can optimize in this way. The Mercury compiler also implements middle recursion optimization. The code for append with this optimization in section 5.2 was generated by the compiler (the other examples in section 5 being handwritten); we disabled this optimization when generating the examples in section 3.3, since the optimization had not been covered up to that point in the paper. The compiler also implements several other high-level optimizations, including automatic inlining, common subexpression elimination, predicate specialization to eliminate unused arguments, specialization of higher-order predicates when their higher-order arguments are known, and the reuse of the storage of terms that occur more than once in a predicate.

The compiler implements a whole host of low-level optimizations. These include constructing ground terms at compile time, stack slot allocation using graph colouring, shortcircuiting of jumps to jumps and to calls, shortcircuiting of call return addresses pointing to jumps, replacement of jumps by the target code, replacement of calls that perform only the procedure epilogue after return by tail calls, elimination of useless jumps and jumps around jumps, elimination of dead labels and code, delaying the creation of stack frames as long as possible, and value numbering on extended basic blocks to eliminate useless assignments and common subexpressions and to put values used more than once into registers. These low-level optimizations were switched on for all of the examples in section 3 except the first.

7. Performance results

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

7.1. Background

Mercury allows programmers to choose what combinations of GNU C extensions they would like to use by specifying what we call a *grade* as a command line option. There are six main grades, none, reg, jump, fast, asm_jump and asm_fast, whose characteristics are shown in table 7.1. Not all grades are available on all systems. Grade none yields portable ANSI C code. Each grade can be used without a garbage collector or with Hans Boehm's conservative garbage collector for C, as we have described earlier; programmers can ask for Boehm's garbage collector by adding the suffix ".gc" after the grade option.

GNU C extension	none	reg	jump	fast	asm_jump	asm_fast
global register variables	no	yes	no	yes	no	yes
non-local gotos	no	no	yes	yes	yes	yes
assembly labels	no	no	no	no	yes	yes

TABLE 7.1. The use of GNU C extensions by Mercury grades

The Mercury runtime system is implemented as a shared library on machines that support shared libraries (really shared objects) in a way that does not interfere with our use of GNU C extensions, and as a traditional library on machines that do not. The run-time system contains the driver routine for grades that do not use nonlocal gotos (see section 3). 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. This reduces collisions in machines with set-associative caches as well. 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, OSF/1 and many other modern versions of Unix, the run-time system sets up "red zones" towards the ends of the heap and the stacks. Access to a red zone will result in a signal being delivered to the process. Our native Mercury garbage collector is based on the idea that the signal handler that is notified on overflow will unprotect enough reserved memory in the redzone to let the computation reach its next jump point, but it will override both the success and failure continuations to redirect them to the garbage collector. The garbage collector will thus get control at a point where the "shapes" of the contents of registers and stack slots (if any) are known. (This approach had problems with earlier versions of Unix, but works just fine with more modern versions.) This garbage collector is not finished yet, but this is not a problem since the benchmark programs do not use enough memory to require any garbage collection.

7.2. Speed tests

We have tested the speed of the Mercury implementation on a set of standard Prolog benchmarks which we have translated to Mercury. In most cases the translation was simply a matter of figuring out and adding the proper declarations. We made only two other kinds of changes. First, in one case we had to compensate for Mercury's lack of the "`=\=`" operator. Second, in some benchmarks, the Prolog code has predicates that either enforce determinism through cuts, or don't enforce determinism even though it is required; for Mercury, we converted these predicates to use switches or if-then-elses.

We ran the benchmarks on Mercury version 0.5-beta and on six other logic programming systems: SWI-Prolog 1.9.0 [35], NU-Prolog 1.6.4 [31], wamcc 2.21 [4], Quintus Prolog 3.2, SICStus Prolog 2.1 [3] and Aquarius Prolog 1.0 [32]; these versions are the latest we had access to. We would have liked to run the benchmarks on some other systems as well, PARMA [30] and Turbo Erlang [8] 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 the benchmarks do not benefit significantly from any optimizations requiring interprocedural analysis, our performance results indicate directly the effectiveness of the execution model we described in section 3.

SWI-Prolog, NU-Prolog and probably Quintus 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 [9], 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 perform global analysis on the program; we tested Aquarius both with and without this option. Wamcc compiles to C and then to object code; the results we report use the -fast_math option.

NU-Prolog, SICStus Prolog and Aquarius Prolog allow programmers to supply declarations, but do not require it. We therefore tested each of these systems 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. The declarations cause checks to be made at runtime, so they are a source of overhead. However, with the help of these declarations, programmers can write programs that use coroutining, which substantially speed up some programs. The NU-Prolog compiler also uses the declarations to generate better code for indexing.

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 declarations yield significant speedups for only two of our ten benchmarks in the presence of program analysis.

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 by 1 Mb of unified 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. 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 eight or more trials. (The Aquarius compiler does not run under Solaris 2 due to differences in assembler formats, so we compiled Aquarius benchmarks on another machine running SunOS 4.1.2, although of course we ran the resulting executables on the SPARCserver 1000.)

Table 7.2 contains summaries of our timing results, using the arithmetic, geometric and harmonic means respectively. These numbers are derived from table 7.3, which shows the speed of each variant of each system on each benchmark. (The first two columns mean the same thing in the two tables, although table 7.3

system	variant	arith	geom	harm
SWI		1.00	1.00	1.00
NU	no declarations	1.84	1.77	1.72
	declarations	4.92	2.17	1.67
wamcc		3.35	3.25	3.14
Quintus		4.22	3.92	3.74
SICStus	compactcode, no declarations	2.09	2.05	2.02
	compactcode, declarations	4.10	2.26	1.88
	fastcode, no declarations	7.45	7.12	6.83
	fastcode, declarations	5.77	3.70	3.15
Aquarius	no declarations, no global analysis	12.76	11.23	9.87
	declarations, no global analysis	16.09	13.80	12.26
	no declarations, global analysis	24.79	21.49	18.42
	declarations, global analysis	28.92	22.96	19.00
Mercury	none grade, Boehm gc	10.99	8.08	6.36
	none grade, no gc	16.11	14.64	13.39
	asm_fast grade, Boehm gc	23.41	12.97	9.24
	asm_fast grade, no gc	43.86	39.33	36.52

TABLE 7.2. Averages of benchmark speed ratios

has the system and variant names in compressed form.) All speeds in these tables are normalized to the speed of SWI-Prolog, a freely available system. The raw timing data, user-mode times required to produce all solutions for one iteration of each benchmark on each system variant, is accessible through our World Wide Web page. 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.

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. Prime finds all primes up to 100. Qsort quicksorts a list of 50 integers using difference lists. Queen finds all safe placements of 9 queens on a 9x9 chessboard. Cqueen is the same benchmark after it has been put through the source-to-source transformation of Seki and Furukawa [24]. 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. Deriv, nrev, primes, qsort and tak are entirely deterministic, poly is mostly deterministic with some semideterministic code, and cqueen, crypt, queen and query have significant amounts of both nondeterministic and either deterministic or semideterministic code.

We currently support Mercury on six platforms: SPARC/Solaris 2.x, MIPS/IRIX 5.x, MIPS/Ultrix 4.3, Alpha/OSF/1 3.x, x86/Linux and x86/BSDI 1.1. All six platforms support GNU C global register variables. Five support GNU C nonlocal gotos (the exception is the Alpha), although under IRIX the use of nonlocal gotos interferes with shared libraries. These five also support assembler labels. Given this wide availability of these three gcc extensions, the asm_fast grade is much more representative of Mercury's performance than the none grade, which we include for purposes of comparison in tables 7.2 and 7.3.

None of the benchmarks require garbage collection when run just once. The Prolog systems don't need garbage collection even when they are run many times, since the failure driven loop in the benchmark harness recovers all memory. Boehm's garbage collector has no similar facility for cheap recovery of memory on backtracking, which leads to repeated garbage collections when running the allocation intensive benchmarks many times. This effect reduces the apparent performance of Mercury variants using Boehm's garbage collector in tables 7.2 and 7.3 in a way that does not occur in realistic programs (see section 7.3). In any case, given that our design for the native Mercury garbage collector does not suffer from this problem, and that the behavior of a system with this native collector is identical during forward execution with the behavior of a system with no gc, the results without garbage collection are much more representative of Mercury's potential than the results with Boehm's conservative garbage collector.

Table 7.2 shows Mercury to be the fastest system overall with all three averaging methods. Using the harmonic mean, the asm_fast grade of Mercury is 92% faster than the fastest variant of Aquarius. It outperforms SICStus fastcode by a factor of 5.3, Quintus Prolog by a factor of 9.8, wamcc by a factor of 11.6, and all the other systems we measured by factors ranging from 18 to 36. Using the geometric mean, Mercury is 71% faster than Aquarius, while using the arithmetic mean, Mercury is 52% 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, table 7.3 shows that declarations slow down SICStus compact code on every benchmark except one (queen), 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.

Mercury is the fastest system on all ten benchmarks, Aquarius with analysis and declarations is the next fastest system. Its speed comes closest to Mercury on the two smallest benchmarks, nrev and tak, while its relative performance is worst on the two largest benchmarks, crypt and poly.

Mercury's speed on nrev depends on either its two predicates being declared without polymorphism, or the optimization to remove unused arguments being switched on. If the predicates are declared polymorphic and the optimization is not switched on, the overhead of passing around a pointer to a type_info structure which is never used slows down the asm_fast grade of Mercury by about 16% on this benchmark. Only two of the other benchmarks, queen and cqueen, have any predicates that are naturally polymorphic; the impact of unoptimized polymorphism on Mercury's speed on these benchmarks is less than 2%.

NU-Prolog and SICStus both perform very well on the nine-queens program when declarations are turned on. The complexity of this benchmark is factorial when executed left-to-right, but the declarations enable coroutining which reduces this to polynomial. Nevertheless, the Mercury execution algorithm is so much faster than the execution algorithms of NU-Prolog and SICStus, that for nine queens the constant factor outweighs the big-O advantage, which asserts itself slowly. For ten queens, NU-Prolog with coroutining is less than twice as fast as Mercury; for eleven queens, it is slightly over four times as fast.

Comparing the raw execution times for queen with cqueen shows that even on

		cqu	cry	der	nrv	pol	pri	qst	qun	qry	tak
SWI		1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
NU	-	1.5	1.4	1.5	3.4	1.8	1.6	1.6	1.6	1.9	2.2
	d	1.6	1.2	0.9	3.4	1.6	1.7	1.2	33.8	1.9	1.9
wcc		4.3	4.3	2.2	3.7	3.0	3.5	2.8	4.1	2.0	3.7
Qui		3.8	3.5	2.8	10.4	3.6	3.5	3.5	4.3	3.4	3.4
SI	c	2.1	2.1	1.6	3.0	2.2	1.7	1.8	2.1	1.8	2.5
	cd	1.9	1.8	1.3	1.9	1.9	1.5	1.4	25.3	1.7	2.2
	f	6.1	6.0	6.8	12.1	7.9	5.7	6.9	6.8	4.4	11.6
	fd	3.5	3.1	2.4	2.2	3.2	2.7	2.2	30.8	3.9	3.8
Aq	-	16.2	7.3	7.2	15.7	9.8	8.1	14.7	15.9	4.5	28.0
	d	17.5	7.4	7.9	19.4	12.6	9.0	16.7	17.4	8.6	44.3
	a	25.9	7.5	15.7	36.9	18.8	11.7	32.6	26.1	18.2	54.5
	ad	25.9	7.6	15.8	36.8	18.9	12.9	32.8	26.1	18.2	94.2
Mer	ng	13.4	6.1	2.7	4.1	7.2	6.4	4.7	7.7	24.1	33.6
	n	19.0	12.5	6.6	14.2	15.0	12.3	10.3	12.8	24.4	33.9
	ag	22.8	7.3	4.1	4.9	10.1	8.4	8.5	12.2	39.9	116.0
	a	49.1	29.1	25.8	40.2	40.0	23.6	39.8	33.9	40.2	116.8

TABLE 7.3. Benchmark speed ratios on SPARCserver 1000, SWI-Prolog = 1

systems that can coroutine, 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 prefer source-level program transformations [15, 24] to coroutining. We are working on a stream AND-parallel implementation of Mercury that will subsume the functionality offered by coroutining systems. The parallel version of Mercury will work for programs with circular data dependencies, which the current system does not allow.

With NU-Prolog and SICStus Prolog, it is striking that although declarations can speed up the program dramatically by inducing coroutining (as in queen) or slightly by allowing better indexing (e.g. NU-Prolog on cqueen), 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 its use of the GNU C extensions. The most important extension is the use of global register variables to keep the most important abstract machine registers in the registers of the physical machine. The payoff from this 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. 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 jump instruction and zero instructions respectively. For jumps within a single C function, these jumps are always simple

System	compile time	file size	run time
NU-Prolog	8.4 minutes	1.9 Mb	116 s
SICStus compactcode	7.2 minutes	4.4 Mb	101 s
SICStus fastcode	13.6 minutes	7.4 Mb	71.1s
Mercury asm_fast.gc	61.8 minutes	2.2 Mb	18.5s

TABLE 7.4. The Mercury compiler as a large benchmark

direct jumps. When not using assembler labels, jumps between different C functions are indirect jumps in which the address to jump to is fetched from a variable in memory (see section 3.1). The use of assembler labels improves performance by turning jumps between C functions into direct jumps as well, but a more important benefit is that it avoids long startup times. (Without assembler labels, every module of the program must be paged in at startup to initialize the variables used by indirect jumps.)

The speeds reached by the asm_fast and none grades of Mercury on the SPARC-server 1000 on the nrev benchmark correspond to 4.68 Megalips and 1.65 Megalips 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 by hand raises the speed of nrev to 10.00 Megalips with the asm_fast grade, which is as fast as a handwritten C program compiled with full optimization.

7.3. Program size and compile time

The Mercury compiler is a large program, approaching a hundred thousand lines of code. It is mostly written in the intersection of three languages, Mercury, NU-Prolog and SICStus Prolog (some primitives have three separate implementations). We have therefore used the compiler as a very large test case to compare the performance of these three implementations. The results of these comparisons are summarized in Table 7.4. The first column gives the time to compile the Mercury compiler and link it with the Mercury library. The second column gives the size of the resulting save file or stripped executable (statically linked). The third column gives the running time of the executable for a typical compilation task: compiling a medium-sized (400 line) module from the compiler.

The third column shows that Mercury retains its efficiency for large programs. In fact, the speed advantage of Mercury asm_fast.gc over SICStus fastcode is much greater for this large program (3.8) than it is for the small benchmarks (1.4). We see similar effects when we compare Mercury asm_fast.gc with SICStus compactcode (5.5 vs 4.6) and NU-Prolog (6.3 vs 5.4).

The second column shows that unlike SICStus fastcode, Mercury does not have a problem with code size. 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; the frequency of predicates with more than one declared mode in the Mercury compiler and standard library is about 5%. Second, the code for each mode is small because it is specialized, whereas the code emitted by a conventional Prolog compiler has to be prepared to handle all possible dataflow patterns (e.g. indexing code typically has to be prepared for the possibility that the variable to be indexed on is not bound). When we ran these benchmarks, the Mercury compiler and standard library had a total of 2.4 Mb of source code. Compiling 2.4 Mb of source code into 2.2 Mb of object code is good performance even for compilers for imperative languages such as C, and it is even better when one considers the high level nature of Mercury code.

The first column shows that the Mercury compiler takes roughly six times longer to compile the compiler than NU-Prolog or SICStus Prolog. The larger compilation time is understandable, given that the Mercury compiler does much more work than the compilers for the Prolog dialects. It is usually not a problem in practice, since Mercury supports separate compilation and the recompilation of a typical size module is quite quick. Nevertheless, reducing compilation times is still worthwhile. The steps we have taken to this end include limiting the number of Mercury procedures in a single C function (to limit the impact of quadratic algorithms in gcc's optimizer), choosing better data representations in the compiler, and reimplementing some predicates shown to be expensive by the profiler. Of course, since the Mercury compiler is written in Mercury, any new general-purpose optimization we implement will also speed up the compiler itself (if we discount the time taken by the new optimization itself).

At the moment about one third of the Mercury to object code translation time is accounted for by gcc –O2 converting C code to object code. We could therefore speed up the overall compilation process for Mercury programs by compiling directly to object code. This could get several benefits beyond the obvious one of avoiding an extra pass involving file I/O. 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. However, generating native code directly 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. Eventually, however, we may write native code generators to supplement, not replace, the C code generator.

8. 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 nonlogical 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 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 by maintenance programmers and help the compiler prevent a large majority 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. Even though our compiler does not yet implement any global optimizations, our benchmark results confirm the superior speed of this approach compared with traditional approaches that cannot rely on access to such information.

The Mercury compiler and standard library are written in Mercury itself. They consist of about 88,000 lines of code so far. Of these lines about 56% contain code and about 14% contain declarations, with the rest being comments (12%) and whitespace (18%). The system 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, one example being our ability to develop this much high-level code in little more than 4 person-years.

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. Now, 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. Some of the people outside the core Mercury development team who have used Mercury report 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 more than 100,000 lines of Mercury code (the compiler and some small applications, including a scanner generator and an interpreter for pure Prolog), 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 [6] and [30]. We consider Mercury's limitations to be more than worthwhile considering the benefits they bring in terms of productivity, reliability, understandability, maintainability, and efficiency.

We would like to thank Peter Ross for his work on the Mercury profiler, Andrew Bromage and Simon Taylor for implementing some optimizations, Tyson Dowd for his work on the native Mercury garbage collector, David Jeffery for his work on the Mercury/C interface, Jane Langley for her work on the front end of the compiler, Will Winsborough, Jeff Schultz and Andrew Taylor for our discussions, Peter Schachte for running the Quintus Prolog benchmarks for us, 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.

For directions on how to obtain the Mercury system, please consult our Web page at `http://www.cs.mu.oz.au/mercury/mercury.html`.

REFERENCES

1. H. Ait-Kaci. *Warren's abstract machine: a tutorial reconstruction.* MIT Press, 1991.
2. H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.
3. M. Carlsson and J. Widen. SICStus-Prolog user's manual. Technical Report 88007B, Swedish Institute of Computer Science, Kista, Sweden, 1988.
4. P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 317–331, Kanagawa, Japan, June 1995.
5. 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.
6. 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.
7. D. Gudeman, S. K. Debray, and K. DeBosschere. jc: an efficient and portable sequential implementation of Janus. In *Proceedings of the International Conference and Symposium on Logic Programming*, pages 399–416, Washington, D.C., November 1992.
8. B. Hausman. Carpe diem, some implementation aspects of Turbo Erlang. In *Proceedings of the Workshop on Practical Implementations and Systems Experience in Logic Programming*, pages 1–12, Budapest, Hungary, June 1993.
9. 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.
10. F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.
11. F. Henderson, Z. Somogyi, and T. Conway. Compiling logic programs to C using GNU C as a portable assembler. Technical Report 96/12, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.
12. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. Technical Report 95/25, Department of Computer Science, University of Melbourne, Melbourne, Australia, August 1995.
13. F. J. Henderson. Strong modes can change the world! Technical Report 93/25, Department of Computer Science, University of Melbourne, Melbourne, Australia, October 1993.
14. P. M. Hill and J. W. Lloyd. *The Gödel programming language.* MIT Press, 1994.
15. D. B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming*, pages 981–998, Cleveland, Ohio, October 1989.

16. F. Kluzniak. Type synthesis for ground Prolog. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 788–816, Melbourne, Australia, June 1987.
17. J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
18. 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.
19. 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.
20. 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.
21. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
22. L. M. Pereira. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, pages 203–210, London, England, June 1986.
23. F. Pfenning, editor. *Types in logic programming.* MIT Press, 1992.
24. 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.
25. E. Y. Shapiro. *Algorithm program debugging.* MIT Press, 1983.
26. 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.
27. Z. Somogyi. *A parallel logic programming system based on strong and precise modes.* PhD thesis, University of Melbourne, Australia, January 1989.
28. 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.
29. 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.
30. A. Taylor. *High performance Prolog implementation.* PhD thesis, University of Sydney, Australia, June 1991.
31. J. Thom and J. Zobel. NU-Prolog reference manual. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.
32. P. Van Roy and A. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.
33. 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.
34. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, October 1983.
35. J. Wielemaker. SWI-Prolog reference manual. Technical report, Department of Social Science Informatics, University of Amsterdam, Netherlands, 1993.
36. N.-F. Zhou. On the scheme of passing arguments in stack frames for Prolog. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 159–174, Santa Margherita Ligure, Italy, June 1994.