Code generation for Mercury

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

Abstract

Mercury is a new purely declarative logic programming language that requires programmers to write declarations for every predicate in the program. Although the main motivation for this requirement is that it allows the compiler to catch most programmer errors, it also allows the Mercury code generator to rely on the presence of type, mode and determinism information about every predicate in the program. The code generator exploits a new execution algorithm based on the availability of this information as well as some novel techniques (lazy code generation, follow-code migration, the use of a compile-time failure continuation stack) to produce very high quality code. This code is in C and is therefore quite portable. Benchmarks show that the Mercury implementation produces much faster code than wamcc, Quintus Prolog, SICStus Prolog and Aquarius Prolog.

1 Introduction

Mercury is a new logic programming language with two unusual properties. First, it is purely declarative; even I/O is done within a declarative framework. Second, it requires the programmer to declare several aspects of the specifications of the predicates they write. These aspects are the types of the arguments; the mode of the predicate, i.e. which (parts of) which arguments are input to the predicate and which are output from it; and the determinism of the predicate, i.e. whether it may succeed more than once and whether it may fail without producing a solution. The first characteristic ensures that programs have a declarative semantics; the second helps the compiler detect predicate definitions whose declarative semantics does not match the programmer's intended meaning.

We designed Mercury with two main objectives in mind. The first objective was to improve programmer productivity on large programming projects. The experience we gathered while writing 80,000 lines of Mercury code has shown us that we have achieved this objective. The declarations enable the compiler to catch most of our errors at compile time, and make it much easier to understand code written by other programmers.

Our second objective for Mercury is high performance High performance Prolog implementations generally get their speed from global analysis of the program. By contrast, the presence of declarations allows the Mercury compiler to derive all the information it wants about the program via simple local analysis, and there are no non-logical constructs to obstruct this analysis. The Mercury compiler we have developed, which is written in Mercury itself, demonstrates the effectiveness of this approach. The code it generates has significantly higher performance than all the other logic programming implementations we have tested it against, which include wamcc [2], Quintus Prolog, SICStus Prolog [1] and Aquarius Prolog [7]. In this paper we show the code generation techniques (including some novel techniques such as *lazy code generation*, *follow-code migration*, and the use of a compile-time *failure continuation stack*) that enable us to achieve this level of performance. These techniques are based on a new abstract machine and the complete information provided by declarations.

The Mercury compiler achieves portability by generating C code (as do Janus, KL1, wamcc [2] and Turbo Erlang [3]). Depending on compile-time flags, our C code can be portable ANSI C or it can exploit GNU C extensions on platforms on which they are available. The first public release of Mercury on 18 July, 1995 supported Suns, SGIs, DECstations and PCs running Linux; we have since added support for 64-bit Alpha-based machines.

The structure of this paper is as follows. Section 2 gives a brief introduction to the Mercury language, while section 3 is an overview of the Mercury implementation. The largest part of the paper is section 4, which presents the algorithms used by the Mercury compiler's code generator. Section 5 evaluates the performance of the generated code.

Due to space limitations, this paper necessarily omits many details. A full length version of this paper and other papers and documentation on Mercury are obtainable from `http://www.cs.mu.oz.au/~zs/mercury.html`. That page also contains directions for obtaining the system's source code; this should be useful for readers interested in the exact algorithms we use.

2 The Mercury language

Syntactically, Mercury is similar to Prolog with additional declarations. Semantically, however, it is very different. Mercury is a pure logic programming language with a well-defined declarative semantics. Like Gödel [4], Mercury provides declarative replacements for Prolog's non-logical features. Unlike Gödel, Mercury provides replacements for *all* such features, including I/O.

Mercury's type system is based on a polymorphic many-sorted logic. It is essentially equivalent to the Mycroft-O'Keefe type system [5], and to the type system of Gödel [4]. The definition of a type lists the function symbols (functors) to which variables of that type may be bound, and gives the types of the arguments of those functors. Types may be polymorphic, as in

```
:- type tree(T) ---> empty ; node(T, tree(T), tree(T)).
```

Programmers must declare the types of the arguments of all predicates:

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

Mercury also requires programmers to declare the modes of their predicates. In their simplest form, mode declarations specify which arguments to a predicate are input (ground on entry to the predicate), and which arguments are output (free on entry to the predicate and ground on success). Predicates may have more than one mode:

```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
```

All declarations may be given on one line if the predicate has one mode:

```
:- pred flatten(list(list(T))::in, list(T)::out) is det.
```

Mercury also allows arguments to be partially input and partially output, as long as the programmer specifies which part is input and which is output. In such modes, the descriptions of the initial and final instantiation states are based on the structure of the type concerned.

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 in the given mode, a predicate has exactly one solution, then it is deterministic (`det`). If it has at most one solution, then it is semideterministic (`semidet`). If it has at least solution, then it is multisolution (`multi`). If it may have multiple solutions or none, then it is nondeterministic (`nondet`). If it always fails, it should be declared `failure`. If it neither fails nor succeeds (i.e. it either loops or aborts), it should be declared `erroneous`. The last two are rarely used.

The compiler checks all type, mode and determinism declarations. In the process, it infers the types of all local variables, the modes of all calls, and the determinism of all goals and subgoals, including any missing determinism declarations. It also reorders conjunctions as necessary to ensure that goals that consume a variable come after the goal that produces that variable; since logically the order does not matter, the compiler is free to pick any valid ordering. The compiler must reject the program if any declaration is not satisfied, or if the flow of data in any predicate is circular. Programs that pass the compiler cannot flounder, and they do not need the occur check. (The parallel version of Mercury, now under development, will allow circular dataflows, and reintroduces the need for the occur check.)

Predicates that perform I/O must be deterministic, and they have an extra pair of arguments (usually hidden by DCG notation) representing the old and new states of the world. The in,out modes of these arguments must have "uniqueness" annotations that allow the compiler to verify that the old state of the world will not be referred to again.

By allowing only pure, type-correct, mode-correct, determinism-correct programs, Mercury is in some respects a less expressive language than Prolog. However, we do not find Mercury's requirements to be restrictive; on the contrary, we find them liberating. While certain Prolog idioms are not possible in Mercury, alternative idioms are available, and these idioms are usually easier to understand, usually more efficient, and invariably declarative. This reflects the different characteristics of the two languages. Prolog is well suited for exploratory programming, while Mercury is intended for the development of large systems by teams of programmers. The extra declarations required by Mercury make maintenance easier, especially in team projects, and enable the compiler to catch the vast majority of program errors, making programmers more productive and their programs more reliable. They also allow the compiler to generate very efficient code. We therefore consider Mercury's limitations to be more than worthwhile considering the benefits they bring.

3 System overview

The Mercury compiler processes Mercury programs one module at a time. It uses two main internal representations of the module being compiled. The first is an annotated form of the source code, which we call the High Level Data Structure, or HLDS; it also contains the declared interfaces of imported modules. The second is a representation whose primitive elements correspond fairly directly to the C statements the compiler emits; we call this representation the Low Level Data Structure, or LLDS.

When creating the HLDS, the compiler transforms predicates into what we call *superhomogeneous form*. This involves replacing multiple clauses with an explicit disjunction, so that the predicate body is a single goal, and introducing new variables and new unifications as necessary until every atom (including the head) is of one of the forms $\texttt{p(X}_1\texttt{, ..., X}_n\texttt{)}$, `Y = X`, or $\texttt{Y = f(X}_1\texttt{, ..., X}_n\texttt{)}$, where all the X_i are distinct variables. This simplifies much of the work of the following phases.

Predicates in Mercury may have several declared modes; we refer to each declared mode of a predicate as a *procedure*. Type analysis works on one predicate at a time, but after that the various procedures of a predicate are handled completely separately. Mode analysis will in general reorder each version of the predicate differently and compute different modes for the calls in the body. (All calls must match exactly one of the declared modes of the called predicate. If the call has a non-variable in what should be an output position, mode analysis will introduce extra unifications to keep this invariant.) Later passes will in general also put different annotations on each procedure. The code generator generates separate code for each procedure. The various procedures of a predicate do not share any code, but this is not a problem because very few predicates (1 to 5%) have more than one mode.

Several aspects of a procedure's treatment by the code generator depend on its *code model*. Det and erroneous procedures share the det code model, semidet and failure procedures share the semidet code model, and nondet and multi procedures share the nondet code model. In the rest of the paper we will talk about code models more than determinisms.

The Mercury runtime system uses three main memory areas, which we call the heap, the det stack, and the nondet stack. Their relative placement and direction of growth do not matter. The heap is very similar to the heap of the WAM; it is the area in which all structured terms are created. Its memory is reclaimed on backtracking and by garbage collection. The det stack holds the stack frames of procedures that cannot succeed more than once; these frames are popped when their procedure succeeds. Predicates that can succeed more than once may be backtracked into, so we store their stack frames on the nondet stack, whose frames are popped on the failure of the procedure concerned.

These areas are managed by several abstract machine registers. The heap pointer hp points to the next free word on the heap. The stack pointer sp points to the next free word on the det stack. The maxfr register points to the top of the top frame on the nondet stack, while the curfr register points to the top of the current frame on the nondet stack. There are no explicit links between frames on the det stack; the code manipulating a det stack frame always knows how big the stack frame is, and generated code never refers to any det stack frame except the top one. Frames on the nondet stack are linked together in two ways, using fixed slots in every nondet stack frame. The prevfr slot of a nondet frame always points to the frame immediately below it. The succfr slot contains the value of curfr when the frame was created.

Calls place input arguments in abstract machine registers named r1, r2 etc, and pick up output arguments from the same registers. Calls also place the address of the code to return to in an abstract machine register called succip (success instruction pointer). The called procedure must save the value of this register across its own calls. Det stack frames may use any slot for saving succip, while nondet stack frames have a dedicated slot for this purpose. Nondet stack frames also have a slot dedicated to holding the address at which execution should resume if their procedure is backtracked into; we call this the redoip slot, for redo instruction pointer.

When a procedure using the nondet stack succeeds, it leaves its frame on the nondet stack, sets curfr from its succfr slot to point to the frame of its nearest nondet ancestor, and branches to its caller through its succip slot. Such a predicate is backtracked into when its frame is on the top of the nondet stack. When this happens, the backtracking mechanism (the redo macro) sets curfr and maxfr to point to this frame and branches to the procedure's next alternative, whose address is in the redoip slot. The last time this happens, the redoip slot will point to code that invokes the fail macro, which discards the top nondet stack frame, sets curfr and maxfr to point to the newly exposed frame, and branches through its redoip slot.

Stack frames on both stacks may contain the values of variables being saved e.g. across calls or disjunctions. We never generate any pointers to stack slots containing variables. Values that don't fit in stack slots are always stored on the heap; values that fit in stack slots also fit in registers and are therefore always passed by value. The reason why we can pass input arguments by value is that mode analysis always puts all consumers of a variable after its producer. This is also the reason why free variables will never be referred to and hence do not need initialization, why we never have to build reference chains, and why we don't need a trail.

We specialize the representation of terms for each type. Basic types like `int` are represented as a single untagged machine word. For discriminated union types we use tagged pointers. For types with up to four function symbols, the primary tag in the two low-order bits of the word specifies the function symbol; if the function symbol's arity is not zero, the rest of the word is a pointer to an aligned cell on the heap containing one word for each argument. For types with more function symbols, some primary tags values must be shared, either by several constants (which are distinguished by a (usually) 30-bit secondary tag in the rest of the word) or by several function symbols of non-zero arity (which are distinguished by a secondary tag word at the start of the pointed-to memory cell).

For further information on the Mercury execution model, see [6].

4 Code generation

The conversion of a Mercury procedure into compact and efficient code is surprisingly straightforward for the most part. The code generator performs a single left to right depth-first traversal of the annotated HLDS component representing the body of the procedure, producing a tree of LLDS code fragments along the way. This tree is then flattened and output as a list of C statements.

The code generator has a state threaded through most of its code using DCG notation. The most important part of the state is the exprn_info structure, which maps each bound variable to its current status. This status may be *cached*, which means that the code to evaluate the variable has not been emitted yet, but that the value may be obtained by evaluating an expression involving other variables. The other possible status is *available* directly in one or more rvalues; the code generator will always choose the cheapest of these when it wants the value of the variable. This technique allows the code generator to avoid redundant moves by computing variables directly into the lvalues where their values are needed. It also means that the code generator is often able to move code after tests that may fail, reducing the probability that it will be executed at runtime, and sometimes it can avoid emitting the code at all. We call this technique *lazy code generation*; the next few sections describe it in more detail.

4.1 Unifications

Mode analysis classifies unifications into five categories: assignments, constructions, deconstructions, simple tests and complicated unifications. The code generator handles each of these differently.

A unification is an assignment if it has the form X = Y and either of the modes (in, out) or (out, in); assume the former. The code generator emits no code directly for such unifications. Instead, it creates an entry in exprn_info for Y, giving its status as cached, with the cached expression being the variable X. When the code generator needs the value of Y, it will find this status in exprn_info and proceed to flush the cached expression. In this case this means finding rvalues that yield the value of X and copying this set to become the set of rvalues yielding Y. If X was not already evaluated, then its cached expression must also be flushed, and so on; the flushing process is recursive. If it finds that the value of the variable being flushed must be computed, the flushing process will generate the value directly into the lvalue where the code generator requires it to be.

A unification is a construction if it has the form X = f(Y_1, ..., Y_n), where X is output and the Y_i are distinct variables that are either input or void. The compiler does not directly emit code for such unifications either. If $n = 0$, the code generator records the status of X as evaluated, with its single rvalue being the constant f; the rest of the code generator can use this rvalue directly. If $n > 0$, the code generator records the status of X as cached with the expression being the term on the right hand side. When the time comes to flush such an expression, the code generator checks whether the value of the expression is known at compile time. If it isn't, it emits code to allocate a new cell on the heap, to tag the pointer to the cell with the right primary tag value, to fill the remote tag slot if any, and to fill the slots corresponding to those Y_i that are input to the construction. The slots corresponding to Y_i that are void are left uninitialized; this is how partially instantiated data structures are created.

A unification is a deconstruction if it has the form X = f(Y_1, ..., Y_n), and X is input and the Y_i are output or void. The compiler may know that X must be bound to f at this point; the type of X may have only one alternative, or X's value may already have been tested. If this is not the case, the code generator emits code to test the top-level functor of X and to fail if it isn't f. If $n > 0$, the test extracts and compares the primary tag, and the secondary tag if there is one; if $n = 0$, the test compares the entire word. The three forms of test are illustrated by the following three if statements, the bodies of which are dictated by the top element of the compile-time failure continuation stack (see section 4.5):

```
if (!((tag(r2) == mktag(3)) && (field(mktag(3),r2,0) == 0))) ...
if ((tag(r2) != mktag(1))) ...
if ((r2 != mkword(mktag(0), mkbody(0)))) ...
```

After emitting the test, the code generator updates exprn_info with an entry for each non-void Y_i giving its cached expression as the i'th field of X.

A unification is a simple test if it has the form X = Y and the mode (in, in), where X and Y are of atomic types (i.e. builtins or enumerations). For these the compiler generates a test very similar to the test for X = a.

The only other kinds of unifications left by the transformation to super-homogeneous form are unifications of the form X = Y that do not qualify as simple tests or assignments. The code generator implements such complicated unifications by calling a compiler-generated unification predicate that is specific to the type shared by X and Y. The compiler derives the structure of the unification predicate from the structure of the type. Here is an example:

```
intlist_unify([], []).
intlist_unify([H1 | T1], [H2 | T2]) :-
        H1 = H2,
        intlist_unify(T1, T2).
```

This code is specific to lists of atomic types. For our solution to the problem of unifications involving polymorphic types, see the full version of this paper.

4.2 Predicate calls

The Mercury parameter passing convention is that all arguments are passed and returned by value in abstract machine registers. For most calls, the n'th argument will be passed or returned in abstract register rn; for calls to procedures with a semidet code model, it will be passed or returned in abstract register r$n+1$, with r1 being reserved for a success/failure indication. An argument is an input argument if its top function symbol is bound at the time of the call; it is an output argument if its top function symbol is bound by the call. Procedure calls must put input arguments in their registers before the call; the called procedure will put the output arguments in their registers. The calling procedure need not put anything in the registers occupied by non-input arguments, and the called procedure may destroy the contents of the registers occupied by non-output arguments.

When the code generator encounters a call, its first job is to emit code to save variables that are live after the call onto the stack. (The code model of the procedure decides which stack this is.) An HLDS annotation on the call tells the code generator which variables need saving, and another annotation in the caller's HLDS tells it in which stack slot each variable should be saved. (The pass that generates these annotations builds an *interference graph*, in which a link connects two variables if they have to be stored on the stack at the same time; it then uses a graph colouring algorithm to compute which groups of variables can share a stack slot.) The code generator consults the exprn_info structure for each variable that needs saving. If the variable is already in its stack slot, it need not emit any code. If the variable is not there but it is evaluated, the code generator emits code to copy it from its cheapest current location to its stack slot. If it is cached, the code generator flushes its cache entry by emitting code to evaluate the variable directly into the stack slot.

The next job of the code generator is to place the input arguments in their registers. Since the program is in superhomogeneous form, all the arguments are distinct variables, and therefore the code generator can use the same algorithm as it used for saving variables on the stack. During both of these operations, the code generator is careful not to overwrite the last copy of the value of a live variable. When it would otherwise overwrite such a value, it emits code to copy the value to a spare register.

To make the call, the code generator allocates a new label, emits an invocation of the call() macro, and emits the new label. One of the arguments of the call macro is the address of the called procedure; the code generator derives the name of its label from the predicate's name and arity and the number of the mode in which the predicate is called. The other argument of the call macro is the success continuation or return label, for which the compiler passes the new label. The call macro assigns the address of the return label to the succip register and then branches to the called procedure. When control reaches the return label, the only registers holding useful values are those containing the output arguments of the called procedure, and the code generator updates the exprn_info structure to reflect this fact. (It updates the exprn_info structure on the fly as it saves variables on the stack and places input arguments in their registers.)

For most procedures, the code generator need do nothing else. Deterministic procedures cannot fail; nondeterministic procedures do not return to the success continuation when they fail. Semideterministic procedures, however, always return to this continuation. For them, the code generator must emit code to examine r1 to see whether the call succeeded or failed. If it failed, the code must branch away; the destination is dictated by the top element of the compile-time failure continuation stack (see section 4.5).

4.3 Conjunctions

Most conjunctions involve only unifications and procedure calls; we describe how we handle these in the previous two sections. Some conjunctions contain more complicated goals such as disjunctions and/or if-then-elses; we discuss how these fit into conjunctions when we discuss these types of goals. The only remaining interesting aspect of generating code for conjunctions is where values get stored when control reaches the end.

If the conjunction is the entire procedure body, we would like the code generator to put the output variables into their assigned registers. If the conjunction is part of a branched goal, e.g. one arm of a disjunction or an if-then-else, then we would like similarly precise information about where values should go. If the branched goal is followed by a call, then the code generator can use information about the arguments of the call to guide its placement of values. If the branched goal is followed by some unifications and/or builtins, then an earlier part of the compiler actually pushes these unifications and/or builtins into the end of each arm of the branched goal; we call this *follow-code migration*. In most cases this will leave the branched structure followed by a call. If it isn't, the code generator gets its guidance from the first call that will be executed after the end of conjunction (or the end of the procedure, if there are no calls there), although the usefulness of this information is degraded by distance. In no case does the code generator leave a conjunction (or an arm of a branched structure even if isn't a conjunction) with the exprn_info structure still containing cached variables.

4.4 Switches

Indexing in Mercury is based on the notion of switches, which are disjunctions in which each disjunct tests the same variable against a different functor. (Some Prolog systems, e.g. Aquarius [7], also look beyond clause heads.) The front end of the Mercury compiler contains a pass that looks at disjunctions to see if they fit this pattern, and transforms them into switches in the HLDS.

The compiler has several ways to generate code for switches. The simplest technique is to generate a chain of if-then-elses, one functor-test for each case of the switch, although in switches whose cases cover all the functors in the type of the variable being switched on, the compiler omits the test before the last case, since it is bound to succeed. Normally the compiler puts the cases with the cheapest tests first. However, for switches on types with only two alternatives (e.g. lists, trees) the compiler reverses the sense of the test and puts the likely recursive case first, since this reduces the number of taken branches and hence the number of pipeline breaks in modern CPUs.

```
    if ((r1 == mkword(mktag(0), mkbody(0))))
        GOTO_LABEL(append_3_0_i2);
    <code for cons>
    GOTO_LABEL(append_3_0_i1);
append_3_0_i2: ;
    <code for nil>
append_3_0_i1: ;
```

For switches containing a small number of alternatives, a chain of tests is the fastest possible implementation. For switches containing many alternatives, the Mercury compiler uses dense jump tables or hash tables.

4.5 Disjunctions

Disjunctions that are not switches are nondeterministic constructs, since each disjunct may generate solutions. To improve the efficiency of the code generated for disjunctions that produce no bindings visible from outside, and whose solutions are therefore all equivalent, we transform such disjunctions into semidet if-then-elses (if disjunct1 then true else if disjunct2 then true ... else disjunct$n$).

To generate code for a disjunct, the code generator must know what code to emit for failures occurring within that disjunct. We employ a compiler data structure that we call the *failure continuation stack* for this purpose. Each element of this stack describes a logical place in the HLDS at which forward execution should resume on failure. For each such logical continuation, there may be several labels in the generated LLDS that differ in where they expect the live variables to be stored. At the moment we store two such labels for each continuation. The first expects the live variables to be wherever they happened to be at the time the failure continuation was created (usually some will be in registers), while the second expects all live variables to be in their stack slots. Each logical continuation also contains a flag that indicates what the code generator should do on failure. If the flag is not set, the code generator scans the list of labels in the continuation for the first one whose requirements for the locations of variables are met by the current exprn_info structure, and generates code to branch to it. If the flag is set, it generates an invocation of the redo() macro to invoke the general backtracking mechanism. The latter is necessary (and the flag is set) if the code generator encounters any goal that may push its own frame on the nondet stack, since the alternatives offered by inner goals (typically, those created by a call in one disjunct) must be explored before the alternatives offered by outer goals (e.g. the next disjunct).

Before starting to generate code for the first disjunct, the code generator pushes a new continuation containing two new labels onto the failure continuation stack. The first will be used by failures that occur while the registers have their initial contents; such failures do not need any stack accesses. The second label will be used by failures that occur after the first call or nonatomic construct in the disjunct. Before it generates a branch to the second label, and in any case before control leaves the disjunct, the code generator emits code to save all variables in their stack slots. To ensure that an invocation of the general backtrack mechanism finds the proper label for resumption of forward execution, on entry to the disjunct the code generator emits code to set the redoip slot of the top nondet stack frame to point to the second label; at all points where such failures can happen, the live variables will be in their stack slots.

At the start of later disjuncts, the code generator emits the second label, code to move the live variables from their stack slots into the locations they had on entry to the disjunction, and then the first label. The code for the later disjunct will begin there, and the exprn_info structure is set up to reflect this. The code generator pops the top failure continuation off the stack, and unless this disjunct is the last one, it pushes on a new failure continuation reflecting resumption of execution at the next disjunct. The code generator then modifies the redoip slot according to the new top element of the failure continuation stack. The initial contents of the stack depends on the code model of the procedure concerned. For nondet procedures, it contains one element with the flag set. For semidet procedures, it has one element with the flag clear and two labels with the same requirements (no live variables) pointing to the failure epilogue code, which sets the success indication register (r1) to FALSE and returns. For deterministic procedures, the stack is effectively empty.

These techniques together allow us to handle procedure bodies that contain nested disjunctions. Each nested disjunction adds an element to the failure continuation stack. When the inner disjunction reaches its last disjunct, its popping of the failure continuation stack exposes the element left there by the outer disjunction.

When the code generator has finished generating code for a disjunction, and is about to proceed to the rest of the procedure body, it sets the flag in the top element of the failure continuation stack. If any goals following the disjunction fail, the code generator must emit code to perform a redo. This is required because when processing the code following a disjunction, the code generator can't know where execution should resume on the next failure; it could be the start of any one of the disjuncts, or it could be somewhere inside a procedure called in one of the disjuncts.

If the flag in the top failure continuation is set when the code generator starts processing a disjunction, the code generator knows that there may be other nondet stack frames on top of the current one. Since backtracking within the current disjunction will always refer to the redoip slot of the top nondet stack frame, the code generator would have to in effect *hijack* the redoip slot of a frame that may belong to another call. Since the contents of this slot are unknown to the code generator yet may be needed later, the code generator would have to emit code to save this value at the start of the disjunction and restore it at entry to the last disjunct. However, the second and later disjuncts would be entered by backtracking, and the macros that implement backtracking (redo and fail) both set curfr to point to frame of the nondet stack from which they take the redoip, not to the frame of the current procedure. Creating an auxiliary procedure for the disjunction would avoid the problem, because the disjunction would then use the redoip slot of the auxiliary procedure. The way we handle such situations, pushing a new nondet stack frame and using only its fixed slots, is equivalent in most ways to creating an auxiliary procedure but avoids storing variables in two places.

4.6 If-then-else and negation

The Mercury compiler handles if-then-elses differently depending on whether the condition can succeed more than once or not. If the condition can succeed at most once, the code generator can follow a relatively simple algorithm. It creates a new element on top of the failure continuation stack in the same way as it does for disjunctions. It then generates code for the condition, pops the failure continuation stack, generates code for the then case, emits a goto to the code following the if-then-else, emits the entry points for the else case and generates code for the else case. It need not modify any redoip slots because the semidet code of the condition will not refer to such slots.

If the condition can succeed more than once, then backtracking may cause the computation to reenter the condition. We want execution to continue at the start of the else case if the condition fails before producing a solution. On the other hand, once the condition has succeeded once, we want the entire if-then-else to fail if the condition fails. We arrange for this to happen by emitting code that on entry to the condition saves and then overwrites the contents of the redoip slot with the address of the label starting the else case, and resets the slot to its previous value at the start of the then case. We must make sure that this reset is a soft cut, i.e. it prevents backtracking to the else case but not to the condition. If the condition contains calls to nondet procedures, this is easy, since they will have their stack frames above the one being modified. If the condition contains a disjunction, we must arrange for it to create its own stack frame as well, which we do by setting the flag on the top failure continuation. As with disjunctions, the code generator emits code to create a new frame for a nondet if-then-else if the flag in the top failure continuation is set when arriving at the if-then-else.

Mode analysis ensures that variables bound in the condition of an if-then-else are visible only in the then part; this lets us guarantee soundness without runtime checks. Since negations can be considered a form of if-then-else ("not G" is equivalent to "if G then fail else true"), this goes for negation as well.

4.7 Implicit pruning

If a nondet goal produces no bindings visible from outside the goal, then all the solutions produced by the goal are equivalent, and all solutions after the first should be pruned away. Therefore the determinism analysis treats the goal as semidet, even though its evaluation actually requires nondeterminism. This requires careful handling, because the Mercury execution model cannot tolerate a procedure succeeding more than once unless the compiler has anticipated this possibility and generated code accordingly.

A nondet goal may push frames onto the nondet stack before succeeding for the first time. Pruning away any further solutions requires throwing away these frames, and so the code generator emits code to save the maxfr register before the goal and to restore it when the goal succeeds. If the goal fails without producing a solution, backtracking will resume at the point indicated by the redoip of the nondet stack frame that was on top at the time the goal was entered. If the goal is inside a semidet context such as

System	cqu	cry	der	nrv	pol	pri	qst	qun	qry	tak	mean
SWI	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.00
NU	1.6	1.4	1.5	3.4	1.8	1.7	1.6	33.8	1.9	2.2	1.95
wamcc	4.3	4.3	2.2	3.7	3.0	3.5	2.8	4.1	2.0	3.7	3.14
Quintus	3.8	3.5	2.8	10.4	3.7	3.5	3.5	4.3	3.4	3.4	3.75
SICStus	6.1	6.0	6.8	12.1	7.9	5.7	6.9	30.8	4.4	11.6	7.41
Aquarius	25.9	7.6	15.8	36.9	18.9	12.9	32.8	26.1	18.2	94.2	19.01
Mercury	47.5	29.1	25.4	40.2	39.3	23.3	39.8	31.4	39.0	116.8	35.83

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

the condition of a deterministic if-then-else, this is a problem, since semidet code does not set the redoip. We can arrange for this branch to happen by hijacking the redoip of the frame that is on top when the goal being pruned is entered. Saving and restoring the old redoip is easy if the current procedure has its frame on the det stack. It is more difficult for a procedure that has its frame on the nondet stack, because the failure of the goal being pruned will in general set curfr to point to a frame not controlled by the current procedure. To enable the procedure to continue execution after this failure, we must save curfr on the det stack before entering the goal and restore it on failure. We could then restore the hijacked redoip from either stack; for simplicity we always store the saved values of the hijacked redoip, curfr and maxfr together on the det stack. This works because it is an invariant of our execution model that *any* call that returns, whether with success or failure, always restores the det stack pointer to its original value.

5 Performance results

We have tested the speed of the Mercury implementation on a set of standard Prolog benchmarks which we have translated to Mercury. These benchmarks cover several types of code, containing semideterministic and nondeterministic as well as deterministic predicates. We ran the benchmarks on Mercury and on six other logic programming systems: SWI-Prolog 1.9.0, NU-Prolog 1.6.4, wamcc 2.21 [2], Quintus Prolog 3.2, SICStus Prolog 2.1 [1], and Aquarius Prolog 1.0 [7] (these versions are the latest we have access to). For each system we report the best results we can achieve with that system. For wamcc, this means using the -fast_math option. For Aquarius, it means asking the compiler to perform global analysis on the program, and providing declarations specifying details such as 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. For NU-Prolog and SICStus Prolog, it means providing declarations to enable coroutining whenever doing so improves performance.

Unlike the code generator for Mercury, 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

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	68 minutes	2.2 Mb	18.5s

Table 2: The Mercury compiler as a large benchmark

SPARCserver 1000 with four 50 MHz SuperSPARC processors each rated at 60.3 SPECint92, although none of our tests used more than one processor.

Table 1 contains a summary of our results, with all speeds in the table normalized to the speed of SWI-Prolog. Mercury is clearly the fastest system, being beaten by only one system on one benchmark. Aquarius 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. NU-Prolog and SICStus Prolog both perform very well on the nine-queens program. The complexity of this benchmark is factorial when executed left-to-right, but coroutining reduces the complexity to polynomial. Nevertheless, the Mercury execution algorithm is so much faster than the execution algorithms of NU-Prolog and SICStus Prolog that for nine queens Mercury can hold its own, beating SICStus and being only slightly slower than NU-Prolog. The cqueens benchmark shows how Mercury does on the polynomial algorithm; it is a source-to-source transformed version of nine-queens with the coroutining compiled away.

Using the harmonic mean, which is the appropriate way to average rates, Mercury is 88% faster than Aquarius, and outperforms SICStus Prolog fastcode by a factor of 4.8, Quintus Prolog by a factor of 9.6, and all the other systems we measured by factors ranging from 10 to 36.

The Mercury compiler consists of 94 modules; it has 60,000 lines of code totaling slightly over 2 megabytes. It uses the 35 modules that constitute the Mercury standard library, which have another 14,000 lines totaling about 400 kilobytes. The compiler is mostly written in the intersection of three languages, Mercury, NU-Prolog and SICStus Prolog (some primitives have three separate implementations), so we can use 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 2. 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. 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 results show that unlike SICStus fastcode, Mercury does not have a problem with code size, and that Mercury retains its efficiency even for large programs. (The smaller speed ratio compared to the one from Table 1, 3.8 vs 4.8, is due to our use of Boehm's C garbage collector, which is not native to Mercury.) The larger compilation time is understandable given that the Mercury compiler does much more work than the compilers for the Prolog dialects, but we are working on reducing it. Since we limit the number of procedures compiled into one C function, only one third of the 68 minutes is taken by gcc -O2, so there is room for improvement.

6 Conclusion

Generating efficient code for Mercury is much easier than for Prolog, because the compiler can obtain the accurate and complete information it needs from the declarations. Since this information is guaranteed to be available, the basic execution model can rely on it.

Type information allows the compiler to specialize term representations, reducing the size of terms on the heap and therefore the cost of manipulating them. Mode information allows the code generator to specialize the code it emits for parameter passing and for unifications, and avoids the need for dereferencing and trailing operations and for explicit initialization of free variables before they are bound; it also allows the compiler to generate faster and more compact indexing code. Determinism information allows the compiler to generate specialized code for procedures that succeed at most once. This specialized code uses a smaller stack frame that takes less time to set up, and always removes the stack frame as soon as possible (on success) without having to check at runtime whether the frame is at the top of its stack.

While the basic execution model for Mercury is already more efficient than the best Prolog implementations, our more novel techniques such as lazy code generation and the use of the failure continuation stack contribute further to the speed of our implementation.

We would like to thank Peter Schachte for running the Quintus Prolog benchmarks for us, Tyson Dowd for his comments on a draft of this paper, and the Australian Research Council, the Key Centre for Knowledge Based Systems and the Centre for Intelligent Decision Systems for their support.

References

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

[2] 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.

[3] B. Hausman. Turbo Erlang: approaching the speed of C. In E. Tick, editor, *Implementations of logic programming systems*. Kluwer, 1994.

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

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

[6] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*. To appear.

[7] P. Van Roy and A. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.