The University of Melbourne

Department of Computer Science

Code generation for Mercury

Thomas Conway

supervisor: Zoltan Somogyi

November, 1994

Abstract

We describe a compiler for *Mercury*, a new purely declarative logic programming language designed to provide the support that groups of application programmers need when building large programs. Mercury requires programmers to provide type, mode and determinism declarations. The front end of the compiler uses the information in these declarations to catch many errors that would otherwise require tedious debugging. The back end uses novel techniques to generate efficient code by taking advantage of this information and the clean semantics of Mercury. We show that the code generated by our implementation is significantly faster and more portable than the code generated by mature optimising implementations of other logic programming languages.

1 Introduction

Conventional languages require programmers to pay attention to many low level details such as storage management. Mistakes in these details are often hard to find. Programming languages that take care of these details automatically would significantly improve the productivity of teams of programmers working on large application programs. Predicate logic provides a good basis for the design of programming languages to fulfill this promise.

Unfortunately, current logic programming languages such as Prolog are not widely used in the computer industry. There are several reasons for this:

- They are oriented towards exploratory programming and do not support team programming or the creation of large programs.
- They do not provide the redundancy that is essential for the automatic detection of program errors.
- Their implementations are significantly less efficient than the implementations of conventional languages.

In an effort to improve the efficiency of their programs, Prolog programmers make use of non-logical constructs which destroy the declarative semantics of their programs. This is a pity, because programs with a clean semantics have several advantages over programs without a clean semantics. These advantages include the ability to reason about the program automatically, to transform the program in useful ways, and to apply powerful debugging techniques to the program.

Zoltan Somogyi, Fergus Henderson, and myself have designed a new logic programming language, Mercury, to improve the productivity of teams of programmers working on large application programs. In the design of Mercury we addressed the shortcomings of Prolog.

- Mercury is a pure logic programming language with a clean semantics, and with no non-logical constructs.
- Mercury has a module system to support teams of programmers cooperating in the construction of large programs.
- Mercury has strong type, mode and determinism systems that enable a compiler to detect large classes of programmer errors.
- The Mercury compiler also uses these declarations to achieve what non-logical constructs failed to achieve in Prolog: high performance.

Current high performance Prolog compilers expend large amounts of computational effort to find as much information as they can about the run-time behaviour of the program. The type, mode and determinism systems of Mercury transform the job of the compiler, because they provide the compiler with guaranteed access to complete and accurate information along these lines. The type information allows the Mercury compiler to specialise the representation of terms for each type. The mode information allows the compiler to emit specialized code for each unification and for each mode of use of multimode predicates. The determinism information allows the compiler to limit backtracking to the predicates where the programmer expects it to happen. (In Prolog, this would often require the use of cuts, a particularly nasty non-logical construct.)

We are currently implementing a compiler for Mercury along these lines. The subject of this thesis is the code generator of this compiler. This code generator is differs significantly from code generators for Prolog because it is designed to make the best use possible of the complete and accurate information provided by declarations in Mercury. To meet this challenge, we developed some novel techniques such as *lazy code generation*, *switch detection*, *follow-code migration* and the use of a *failure continuation stack*.

The Mercury compiler is being written in Mercury itself, using NU-Prolog for bootstrapping. It is almost complete. Our benchmarks show the code generated by our experimental Mercury implementation, which as yet lacks many standard optimizations, to be significantly faster than the code generated by mature optimizing implementations of other logic programming languages. Since the Mercury compiler compiles to C, using it as a portable assembly language, our fast implementation can easily be made available on many hardware platforms.

The structure of this paper is as follows. Section 2 gives a brief introduction to the Mercury language. Section 3 presents our execution model. The heart of this thesis is section 4, which presents the structure of the compiler, concentrating on the aspects related to code generation. Section 5 discusses performance results. Section 6 discusses software engineering aspects of the compiler. We have three appendices. Appendix A presents an example tree module written in Mercury, appendix B shows the actual code generated by the compiler for the example predicates presented in section 3, while appendix C explains how to access the compiler sources.

Small parts of this thesis are derived from two publications on Mercury of which I am a co-author. One [18] has appeared in the proceedings of the ILPS '94 Postconference Workshop on Implementation Techniques for Logic Programming Languages, Syracuse, New York, November 1994; the other [17] will appear in the proceedings of the Eighteenth Australasian Computer Science Conference, Adelaide, South Australia, February 1995.

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 [8], Mercury provides declarative replacements for Prolog's non-logical features. Unlike Gödel, Mercury provides replacements for *all* such features, including I/O.

The rest of this section is necessarily brief, and does not describe many of the details of the language. The text of this section is largely based on the corresponding section in [18], and we refer the reader to it for more detail.

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 [9], and to the type system of Gödel [8]. We borrow our syntax from the NU-Prolog [20] type checkers.

The basic method of defining types is with declarations such as

```
:- type bool ---> true ; false.
:- type tree(T) --->
            empty
        ;   tree(T, tree(T), tree(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. 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 tree(T) above). To ensure that no run-time type checking is needed, we require that all type parameters occurring on the right hand side of a type definition also occur on the left hand side.

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

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

```
:- pred tree__contains(tree(T), tree(T)).
```

The compiler then automatically infers the types of all variables.

A program is *type correct* if there is a unique most general assignment of parametric polymorphic types to the variables and function symbols occurring in the program.

In the rest of the paper, we assume that the program is type correct, and that the type of all symbols is known. Type systems of this nature have been the subject of much research, so we omit a description of the type checking algorithm and refer the reader to papers describing type checking algorithms for equivalent and related type systems (see e.g. [9, 10]).

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 [16], we attach mode information to the or-nodes type trees.

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

A term is *approximated by* an instantiation tree if for every node in the instantiation 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 instantiation tree tells us that a variable is bound, there may be several alternative function symbols to which it could be bound. The instantiation 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.

As a shorthand, our mode system provides `free` and `ground` as names for instantiation 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 instantiation tree to a final instantiation tree, with the constraint that no node of the type tree is transformed from bound to free.

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 [5]. Nevertheless, Mercury's mode system is sufficiently expressive to handle more complex data-flow patterns, including those involving partially instantiated data structures.

A *predicate mode declaration* assigns a mode mapping to each argument of a predicate. For example, we can write the following type and mode declarations for tree insertion an tree membership:

```
:- pred tree__depth(tree(T), int).
:- mode tree__depth(in, out).

:- pred tree__member(T, tree(T)).
:- mode tree__member(in, in).
:- mode tree__member(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 instantiation trees of the mode declaration and all of whose free variables are distinct, if the predicate succeeds then the resulting binding of those argument terms will in turn be approximated by the final instantiation trees of the mode declaration. These assertions are checked by the compiler, which rejects programs if it cannot prove that their mode declaration constraints are satisfied.

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

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

- p satisfies its mode declaration, and
- p satisfies a mode (either declared or implied) of every predicate that it calls.

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

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

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

The mode analysis algorithm checks one mode of one predicate at a time. It abstractly interprets each clause of the predicate, keeping track of the instantiation of each variable, and selecting a mode for each call and unification in the clause body. To ensure that the mode set constraints of called predicates are satisfied, it may reorder the literals in the clause; it reports an error if no satisfactory order exists. Finally it checks that the resulting instantiation of the predicate's arguments matches that given by the predicate's mode declaration. For further details we refer the reader to [7, 15, 16].

The mode analysis algorithm annotates each call with the mode used. This is necessary because our implementation generates separate code for each mode of each predicate, and therefore the code generated for a call depends on the mode in which the predicate is being called. We generate inline code for some built-ins such as the arithmetic predicates and for most 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 refer to this as assignment, and denote it 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 refer to this as a simple test and denote it 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 clause); we refer to this as construction and denote it as e.g. `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 refer to this as deconstruction and denote it as e.g. `Y == f(X`$_1$`, ..., X`$_n$`)`.

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

```
unify_tree_t(A, B) :-
    A = empty,
    B = empty.
unify_tree_t(A, B) :-
    A = tree(AE, AL, AR),
    B = tree(BE, BL, BR),
    unify_t(AE, BE),
    unify_tree_t(AL, BL),
    unify_tree_t(AR, BR).
```

The predicate we call to unify AE and BE depends on their type, which is the type of the elements of A and B. We can implement polymorphic unifications like the above either by specialisation of the unification procedure for each instantiation of the type variables, or pass the address of a procedure to unify values of the variable type as a hidden argument.

2.3 Determinism

For each mode of a predicate, we categorise that mode according to the maximum number of solutions it can produce, and whether or not it can fail. For this categorisation we can draw the following table of determinism annotations.

Can fail	Max Number of Solutions		
	0	1	> 1
no	erroneous	det	nondet
yes	failure	semidet	nondet

If in the given mode, a predicate has exactly one solution and cannot fail, then it is deterministic and has the annotation `det`. If a predicate has at most one solution, but may fail then we call it semideterministic and give it the annotation of `semidet`. If the mode of a predicate may have more than one solution then it is nondeterministic and is given the annotation `nondet`.

The three annotations (det, semidet, nondet) suffice for the vast majority of predicate modes, but there are a very small number of predicate modes which require other determinism. The determinism `failure` is used for predicate modes which always fail. In practice this annotation is never used, since a predicate which fails cannot export any bindings, so can be replaced by the simple goal `fail`.

The important example of where the three main determinism annotations are inadequate is where we want a predicate that operationally corresponds to aborting execution. The right logical semantics for such a predicate is that it evaluates to bottom, that is, it loops forever. Thus it does not fail, but it has no solutions. If we want to express this behaviour to the compiler, we use the annotation of `erroneous`. It is important to note that from an external point of view such as that of the user, the program may terminate when a predicate is called in a mode with a determinism annotation of "erroneous", but from the point of view of the calling predicate, such predicates never terminate.

There are many real situations when we want to call an erroneous predicate. Take for example a predicate to return the top node of a tree: tree__top_element/2.

If it is given an empty tree then the obvious thing to do is fail. However, sometimes this is not what we want.

In some situations where we are sure that the tree will never be empty and it is a program error for it to be empty, it may be desirable for tree__top_element/2 to be deterministic, in which case it must have a clause to handle the empty tree. If that clause ever gets executed then we want to abort execution. To do this we call the builtin predicate error/1 which prints an error message and aborts.

Here is tree__top_element/2:

```
:- pred error(string).
:- mode error(in) is erroneous.

:- pred tree__top_element(tree(T), T).
:- mode tree__top_element(in, out) is det.

tree__top_element(empty, _E) :-
    error("No top element").
tree__top_element(tree(E, _L, _R), E).
```

The reason that we must state that error/1 never returns to the calling procedure is so that mode analysis can accept this kind of predicate in which a branch of a branched structure doesn't terminate, but doesn't produce a variable that it should. In this case, the first clause doesn't produce _E, but since it doesn't ever return it does not violate the mode constraint that the second argument of tree__top_element/2 be ground when it returns. The mode analysis can therefore accept the clause as being valid.

The compiler analyses the bodies of predicates to check that their determinism declarations are correct. Although this problem is undecidable in general, several fast approximate solutions are known [4, 12, 13], and they work well in practice.

In fact the compiler needs determinism declarations only for exported predicates — it can (and does) infer the determinism of predicates local to a module. However, determinism declarations are very easy to write, provide important documentation, and help the compiler to detect quite a significant number of program errors, so there is no advantage in making them optional.

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 predicates exported by the module, as well for all local types and predicates. The module may optionally end with an `end_module` declaration.

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

For an example, see appendix A.

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

3 The execution algorithm

We want our Mercury implementation to be fast yet portable. Our speed objective is incompatible with interpretation, yet we do not want to generate native code directly as this would tie us to a particular machine architecture. What we would like to do is generate code in a portable assembly language. The best such language we know of is GNU C. It exists on many platforms, and it extends standard ANSI C with several features that let us access the underlying machine. The extensions we can exploit are

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

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

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

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

3.1 Data representation

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

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

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

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

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

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

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

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

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

3.2 The deterministic execution algorithm

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

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

sp
: The stack pointer: it holds the address of the next free word on the stack.

hp
: The heap pointer: it holds the address of the next free word on the heap.

succip
: The success instruction pointer: it holds the address of the label to return to when the current predicate succeeds.

r1, r2, ...
: General purpose registers for parameter passing and for temporary values.

f1, f2, ...
: Floating point registers for parameter passing and for temporary values.

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

To demonstrate how these are used, consider the following goal:

```
append([1,2], [3], L)
```

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

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

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

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

For our example, the system will put pointers to the lists [1,2] and [3] into registers r1 and r2 respectively before calling append, which is implemented by GNU C code like the following. (The actual code generated by the compiler differs from the following code in minor ways. The actual code can be found in appendix B.)

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

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

The code between points a0 and a1 implements the test `A == []`. In general, we have to strip off the non-tag bits of r1 before the comparison, but since we are comparing against a constant and this constant's primary tag value is not shared, we would be stripping only zeroes anyway. (If the function symbol we are comparing against had a shared primary tag, the condition of the if statement would test the secondary tag after testing the primary tag.) The code between points a1 and a2 implements the assignment `C := B`. The proceed macro then branches to the success continuation, the label whose address was put into the succip register by the caller (i.e. when compiling for GNU C the macro proceed() is defined as goto *succip).

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

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

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

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

3.3 The nondeterministic execution algorithm

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

curfr Points to the nondet stack frame of the currently executing 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 clause fails; points to the code for the next clause if there is one.

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

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

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

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

The framevar slots are at the lowest addresses within each frame while the curfr and maxfr registers and the pointers between frames all contain the address of the highest word within the frame they point to. This arrangement lets us avoid storing the number of saved values in each frame.

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

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

The corresponding C code is:

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

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

Figure 1: Nondet stack near start

```
   r2 = field(TAG_CONS, r1, 0);
   r3 = field(TAG_CONS, r1, 1);
   framevar(1) = r2;
   call(append_2, append_2_b2);
append_2_b2:
   r1 = mkword(TAG_CONS,
      create2(framevar(1), r1));
   succeed();
```

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

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

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

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

The code for the second clause starts by updating the redoip slot via the modframe macro. Since we are entering the last clause, the new value of redoip will cause this call to fail if it is ever asked for more solutions. Dofail is a global variable that contains the address of a label. The code at this label executes the fail macro, which removes the topmost frame on the nondet stack (by setting maxfr to the value of the prevfr slot of the frame being removed), sets curfr to point to the newly exposed frame, and branches to the label whose address is in the redoip slot of that frame. In this case, the failure of append will cause forwards execution to resume in the appropriate clause.

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

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

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

Figure 2 shows the state of the nondet stack just after the last of these successes; different invocations of append are denoted by different suffixes. The usual redo from the interpreter sets curfr to &nondetstack[21] and causes a branch to append_2_b0. The code there sets redoip to dofail and tests the value of C saved in nondetstack[17] to see if it has a cons tag. It does not, so the if statement executes the redo, which now causes a branch to an invocation of the fail macro. This will remove the

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

Figure 2: Nondet stack near end

topmost frame by setting maxfr and curfr to &nondetstack[15] and branch through the redoip slot of the new topmost frame. All remaining frames corresponding to invocations of append have dofail in their redoip slots, so they are discarded one by one. Execution will continue at point given by the redoip slot of the next remaining nondet stack frame.

3.4 The semideterministic execution algorithm

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

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

The other approach is to conceptually transform semidet predicates into deterministic predicates that return a 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 a general purpose register (r1). We plan to eventually implement all the approaches we have described, and to compare their efficiency for large programs.

3.5 If-then-else and negation

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

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

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

The code corresponding to this is:

```
add_element:
   push(succip);
   push(r1);
   push(r2);
```

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

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

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

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

Our implementation of negation bears strong resemblance to the implementation of if-then-else. This is not surprising, since one can transform goals such as `not member(Elem, List0)` into `if member(Elem, List0) then fail else true`. For reasons of space, we defer a more detailed discussion until section 4.11.10.

4 The compiler

The Mercury compiler uses two main internal representations of the programs it compiles. The first is an annotated form of the source program, which is used mainly for the analysis passes of the compiler. Code generation translates this representation into a low level form close to the C level which is used by the peephole optimisation pass and is used to represent the output of the compiler. We call the first of these representations the High Level Data Structure (HLDS), and we call the second the Low Level Data Structure (LLDS).

The front end of the compiler first reads in the program text, and converts it to the HLDS, and then performs a series of passes over the HLDS to do type, mode and determinism checking, and to fill out various annotations needed by the code generator. The back end of the compiler takes a fully annotated HLDS structure and translates it to the LLDS structure (code generation). It then does some optimisation on the LLDS before emitting it as C code.

The use of two intermediate representations greatly improves the capacity of the compiler to support static analysis and low level optimisation. Analysis passes are able to annotate and restructure the program at a high level and provide information to aid code generation, and low level optimisation passes are able to manipulate instructions, while retaining enough information about the program to operate effectively.

The compiler has a number of passes: some for analysis, some for code generation and others for optimisation. The compiler data structures are threaded through the passes — each pass producing the input to the next. When one of the analysis or checking passes detects an error in a predicate, that predicated is marked as containing an error, and no further processing is performed on it by later passes. If errors occur in any predicates, then no code is generated.

It is difficult to split the compiler neatly into a front end and back end because some of the passes which logically belong to the back end are executed before passes that logically belong in the front end. When this occurs it is because the back end pass affects annotations other than those which it computes and the ordering used avoids recomputing them.

Most of the code examples in this section, including the comments, are unmodified output from the compiler. It will be clear from the context where the code is not compiler generated.

The rest of this section discusses the passes of the compiler in the order that they are executed, but first we introduce an example module to help illustrate many of the points that are made later in the discussion of the compiler. Predicates from this module will be presented showing many special situations that occur.

The complete example module is presented in appendix A. It is written differently to how a programmer would usually write a real library module because the intent is different: it is written primarily to exemplify the special cases that arise in programs which the code generator must be able to handle. The module does not include important predicates such as those for constructing trees.

The interface of the tree module looks like this:

```
:- module tree.

:- interface.

:- type tree(T).

:- pred tree__member(tree(T), T).
:- mode tree__member(in, in) is semidet.
:- mode tree__member(in, out) is nondet.

:- pred tree__contains(tree(T), tree(T)).
:- mode tree__contains(in, in) is semidet.

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

In the implementation section of the module, the type tree/1 is defined:

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

4.1 Lexical analysis and parsing

For the purposes of bootstrapping, the compiler currently uses NU-Prolog syntax [20]. This syntax has various flaws that a dedicated Mercury parser (when written) will correct. For example, NU-Prolog syntax defines a total ordering on operator precedences. It makes a lot more sense to define a partial ordering, because it allows the parser to detect more errors, and it is more flexible.

Prior to bootstrapping, the Mercury compiler uses the NU-Prolog lexical analysis and parsing code. In NU-Prolog, lexical analysis is done at the C level for efficiency. The NU-Prolog parser has undergone several generations of hacking, and as a result is difficult to work with. It contains non-logical code to make it as deterministic as possible, so that it can be re-implemented in C. Since we want the compiler to be almost entirely written in Mercury itself (leaving only the lowest levels of I/O to be written in C), this is unsatisfactory.

A lexical analyzer written in Mercury could be achieved by one of two routes: crafting one by hand, or writing a lexical analyzer generator for Mercury similar to Lex, and using that. We have written such a tool called Merl, but it is not yet sufficiently well developed for the task of scanning Mercury programs. Consequently, the initial implementation will be hand crafted.

The Mercury parser will be an operator precedence parser closely based on the algorithm presented in [1]. Some modifications to the basic algorithm are required to handle binary prefix operators, but these modifications have no fundamental impact on the design and implementation of the operator precedence parser.

The operator precedence parser takes the tokens provided by the lexical analysis, and constructs terms. These correspond to what is referred to in most compilers as the parse tree. The top levels of the parser take these terms and yield a list of parsed items which describe the declarations and clauses in the source file. Declarations that import modules are processed as they occur, so that the appropriate interface files are read. This procedure requires the processing of transitive dependencies also.

It should be emphasized that the current syntax for Mercury has been chosen largely for the pragmatic reason that it greatly simplifies the process of bootstrapping the compiler. Opinions vary widely on what good syntax looks like — it is possible to argue endlessly about separators versus terminators, and so on. We believe it to be more important to concentrate our energies on the semantics and execution of Mercury.

4.2 Constructing the HLDS

The list of parsed declarations and clauses is used to construct the HLDS describing the program module. During this construction process a number of transformations are performed on the program clauses to convert the program into a form that is useful for later parts of the compiler.

The HLDS contains a number of tables which hold the various components of a module. Predicates, and type, inst, and mode declarations are all copied into the HLDS from the list of parse items. Other components of the HLDS such as the procedure table are derived from these more basic components.

The translation of clauses into predicate goals involves considerably more than simply copying the structure. Occurrences of the universal quantifier (`all`) are replaced with equivalent expressions using negation (`not`) and existential quantification (`some`). This is primarily to simplify the later passes in the compiler by reducing the number of kinds of goal that have to be handled.

There are more source level constructs which are not currently implemented, but which would be translated in a similar way at this point in the compilation process (for example implication).

Conjunctions and disjunctions are converted from nested binary terms to lists. The goal `true` becomes a conjunction containing an empty list of goals, and the goal `fail` becomes a disjunction containing an empty list of goals.

All the clauses of a predicate are combined into a single disjunction containing a disjunct for each of the original clauses. Fresh variables are introduced as described in section 2.2 to convert the clause to super-homogeneous form.

Consider the predicate in the example tree module: tree__depth/2

```
:- pred tree__depth(tree(T), int).
```

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

tree__depth(empty, 0).
tree__depth(tree(_E, L, R), D) :-
    tree__depth(L, LD),
    tree__depth(R, DR),
    int__max(LD, DR, D0),
    D is D0 + 1.
```

When translated into super-homogeneous form, its goal reads:

```
tree__depth(H_1, H_2) :-
    (
        H_1 = empty,
        H_2 = 0
    ;
        H_1 = tree(_E, L, R),
        H_2 = D,
        tree__depth(L, LD),
        tree__depth(R, DR),
        int__max(LD, DR, D0),
        V_167611 = 1,
        builtin_plus(D0, V_167611, D)
    ).
```

At this point it is worthwhile to annotate each subgoal with those variables which it contains that are *nonlocal* to that goal. A variable is nonlocal to a goal if there is another occurrence of that variable outside the goal.

For example, in the predicate tree__depth/2 above, the occurrence of L in the call tree__depth(L, LD) is nonlocal to that call, because it occurs in the deconstruction unification that produces it.

Note that a goal may contain two variables that have the same name, but which denote different values. Here is an example:

```
p(T1, T2) :-
    (
        tree__depth(T1, D),
        D > 0
    ;
        tree__depth(T2, D),
        D > 0
    )
```

One of the variables denoted by D is local to the first disjunct and the other is local to the second disjunct. T1 is nonlocal to the first disjunct, but not the second, and T2 is nonlocal to the second disjunct but not the first.

The scope information is used later by the mode analysis and the liveness analysis passes.

4.3 Type analysis

The type analysis pass of the compiler processes each predicate and tries to find a unique most general type assignment for each variable. If it cannot do this, the predicate is rejected.

For instance, the following predicate contains a type error:

```
:- pred tree__make_tree(tree(T)).
:- mode tree__make_tree(out) is det.

tree__make_tree(T) :-
    T = tree(42, empty, empty).
```

The error message reported by the compiler for this error is:

```
tree.nl:74: In clause for predicate ‘tree__make_tree/1’:
tree.nl:74:   in argument 1 of functor ‘tree/3’:
tree.nl:74:   type error in unification of argument
tree.nl:74:   and term ‘42’.
tree.nl:74:   argument  has type ‘T’,
tree.nl:74:   constant ‘42’ has type ‘int’.
```

In the current implementation, equivalence types are expanded during the construction of the HLDS rather than during type analysis. This simplifies the type analysis, but does mean that some error messages are less clear than they would otherwise be, since the type name used in the program source is different to the type reported in the error message. Our experience has been that this does not cause many problems.

The following type assignments are given to the variables in tree__depth which was shown in its HLDS form in the previous section:

Variable	Type
H_1	tree(T)
H_2	int
V_167611	int
_E	T
L	tree(T)
R	tree(T)
LD	int
DR	int
D0	int
D	int

The annotations generated by type analysis are used in subsequent passes in several ways. The type of a variable indicates what storage should be allocated for it, how the available tag bits of pointers should be assigned to functors (as described in section 3.1) and how switches should be generated. An important component of the annotations produced by the type checker is the assignment of concrete types to type variables in polymorphic calls. This is because implicit parameters are needed for polymorphic calls to provide unification procedures for variables of polymorphic type.

4.4 Mode analysis

Mode analysis is a crucial stage in the front end of the compiler. During mode analysis, each predicate is transformed from a declarative form, with no execution ordering specified, to what we term a *procedure*, which has a definite execution order. For each declared or implied mode of a predicate a separate procedure is created in the procedure-table in the HLDS.

To create each procedure, the goal from the predicate is copied to the procedure structure and the goal is abstractly interpreted with respect to instantiation. The way in which this is done is described in section 2.2.

As well as finding a well moded ordering of the conjunctions within a goal, the mode analysis fills out a number of annotations. Each subgoal in which any variables have a change in instantiation, is annotated with a mapping from variable to instantiation for all instantiated variable; procedure calls are annotated with the mode number of the called procedure; and unifications are categorized into one of: assignment, atomic test, term construction, term deconstruction, and complicated unifications as described in section 2.2. Mode analysis introduces fresh variables for construction and deconstruction unifications to ensure that all of the sub-unifications are either unifications with a (dead) free variable, or assignments. Complicated unifications ultimately become out of line calls to specialised unification procedures.

There are three forms of unification which we call *test unifications*. These are simple tests, deconstruction unifications and complicated unifications. These unifications may all be semideterministic, whereas assignments and construction unifications are always deterministic.

In section 4.2 we showed the super-homogeneous form of tree__depth/2. Let us consider the mode analysis of the second disjunct in the only declared mode of the predicate, in which H_1, is input and H_2 is output.

The first unification is between H_1 (which is known to be ground because its mode was `in`, from the mode declaration), and a ground term, so the unification is classed as a deconstruction. The second unification (between H_2 and D) cannot be scheduled because neither H_2 or D is bound, and both occur elsewhere, so neither is dead after the unification. Since it cannot be scheduled, it is delayed, and mode analysis proceeds through the conjunction. The two recursive calls to tree__depth/2 can both be scheduled since L and R are both ground (which we know from the mode of the deconstruction unification with H_1), and the calls produce LD and DR respectively. The call int__max (in the mode int__max(in, in, out)) can also be scheduled since LD and DR are both ground. The construction unification which binds the introduced variable V_167611 to the constant 1 is scheduled and the call to builtin_plus/3 which produces D. Now that D has become ground, the unification between H_2 and D is scheduled as an assignment producing H_2.

Having reached the end of the conjunction, the mode analyser checks that there are no unscheduled goals, and finding that there are none accepts the conjunction as mode correct.

The resulting conjunction is:

```
H_1 = tree(_E, L, R), % deconstruction
tree__depth(L, LD), % mode 0
tree__depth(R, DR), % mode 0
int__max(LD, DR, D0), % mode 0
V_167611 = 1, % construction
builtin_plus(D0, V_167611, D), % mode 0
H_2 = D % assignment
```

The change in instantiation information (*delta-inst*) computed during mode analysis and attached to each goal in the procedure is used by liveness analysis, switch detection and the code generator where they need to know the instantiation of particular variables. The information about the category of each unification is used in the code generator to determine what code needs to be generated for each unification. It is also used by some of the passes that compute annotations to distinguish unifications that are inline from those that become calls to out-of-line code. The mode number annotations for calls are used by the code generator to generate a call to the appropriate procedure, and by the determinism analysis to compute the determinism of the procedure call.

4.5 Switch detection

Many disjunctions are composed of mutually exclusive disjuncts. The switch detection pass finds disjunctions which have disjuncts that are distinguished by deconstruction unifications of an input variable to the different functors of the type of that variable. These are then converted into switch structures which are generally more deterministic than the (nondeterministic) disjunctions from which they are derived. If all the possible functors of the type of the variable on which a switch is being performed are present then exactly one of the branches of the switch will match, and the switch is locally deterministic. If only a subset of the possible functors are covered, then the switch will be locally semideterministic since it is possible for none of the disjuncts to match.

The Mercury compiler does a better job of detecting switches than other logic programming implementations because it operates on disjunctions not just clause heads. This more general approach removes the need for programmers to introduce numerous auxiliary predicates just to coerce the language implementation into performing efficient indexing. Moreover, switch detection can be and is applied recursively, so our approach is also more general because it supports arbitrarily deep indexing on multiple arguments.

The two clauses of tree__depth/2 each contain a deconstruction unification of H_1. Switch detection converts the disjunction in the HLDS representation to a switch:

```
tree__depth(H_1, H_2) :-
    ( % det switch on `H_1'
        H_1 has functor empty/0,
        H_1 = empty,
        H_2 = 0
    ;
        H_1 has functor tree/3,
        H_1 = tree(_E, L, R),
        tree__depth(L, LD),
        tree__depth(R, DR),
        int__max(LD, DR, D0),
        V_167611 = 1,
        builtin_plus(D0, V_167611, D),
        H_2 = D
    ).
```

Detecting switches benefits programs in several ways. Most important, they increase the class of deterministic programs that the compiler can prove to be deterministic. Without switches, the class of programs that could be proved deterministic would be small enought to create significant problems for programmers. The use of switches also results in programs that require less baggage to handle "this can never happen" cases where the programmer knows that a piece of code is deterministic but has to put it in the condition of an if-then-else because the compiler cannot prove it to be deterministic.

Using switches rather than sequences of if-then-else-if constructs improves the maintainability of programs. We have found that it is not uncommon to add a new alternative to a discriminated union type after writing a body of code that operates on that type as a whole. If the handling of each alternative of the discriminated union is done with clauses or disjunctions the compiler can give determinism warnings if the programmer modifying his code neglects to include a clause or disjunct for handling the new alternative. In if-then-else constructs the last else case catches the new alternative, and bugs may result.

Another benefit of switches is that they are more efficient than disjunctions. They do not require the allocation of frames on the nondeterministic stack, and a variety of optimizations can be performed during code generation such as the use of jump tables (possibly in conjunction with hashing), rather than using nested if-then-else-if constructs.

4.6 Follow-code migration

Branched sub-goals complicate code generation considerably. The code generator must have some way of ensuring that variables are placed in consistent places at the end of each branch. The follow-vars pass (described in section 4.9) addresses this problem directly.

There is however a way of mitigating, and in some cases entirely removing the problem. If we move builtin calls between the end of the branched subgoal and the next non-builtin subgoal (or the end of the procedure) into the end of each branch of the branched subgoal, we often have a structure for which it is clear where each live variable should be stored at the end of the branch. If the first non-builtin subgoal after the branched subgoal is a procedure call, every live variable will have a fixed location: if a variable is an input argument to the called procedure, then it will have to be located in the appropriate register determined by its argument position, otherwise it will need to be saved onto the stack.

As an example, let us consider tree__member/2 in the nondeterministic mode where the first argument (the tree) is input and the second argument (the element of the tree) is output. After mode analysis, the HLDS structure for this procedure is:

```
tree__member(H_1, H_2) :-
    (
        H_1 = tree(E, _L, _R),
        H_2 = E
    ;
        H_1 = tree(_E0, L, R),
        (
            tree__member(L, E)
        ;
            tree__member(R, E)
        ),
        H_2 = E
    ).
```

The important thing to consider is in the second disjunct (containing a disjunction). Here we have a conjunction containing a branched goal (the inner disjunction), followed by a builtin goal (an assignment unification).

E is live at the end of a branched goal (the inner disjunction) and therefore will need to be stored in a known location at the end of each branch. In this example it is obvious by observation that it should be stored in the register allocated to the output argument of tree__member, however computing which register E should be stored in is not trivial for the compiler since it would in general require considerable effort to propagate argument locations both forwards and backwards until we can decide the best place.

Since H_2 is an output variable, it is easy to figure out which register its value should be stored in. Therefore by moving the unification of E with H_2 into each disjunct, E is no longer nonlocal to the disjunction, so there is no need to retain it. H_2 is nonlocal, but we can easily decide where it should be stored. The resulting HLDS goal looks like this:

```
tree__member(H_1, H_2) :-
    (
        H_1 = tree(E, _L, _R),
        H_2 = E
    ;
        H_1 = tree(_E0, L, R),
        (
            tree__member(L, E),
            H_2 = E
        ;
            tree__member(R, E),
            H_2 = E
        )
    ).
```

The code migration is applied recursively (top-down) so that builtin goals are pushed in as far as possible. This increases the likelihood that the migrated code will be merged with an existing basic block.

Even in the case that the first non-builtin subgoal is not a call, this approach of migrating what we call *follow-code* will still often improve the quality of the generated code. It tends to increase the length of basic blocks, so register allocation and optimisation techniques such as value numbering and peephole optimisation will generally be able to perform better.

There is a possibility that the code expansion caused by the follow-code transformation will cause slowdowns that outweigh its benefits. This is unlikely for two reasons.

First, in our experience the increase in code size is only very small, being less than 1%. Second, as already suggested the use of the follow-code transformation tends to enable the code generator to make better use of registers which leads to an increase in performance.

The follow-code migration pass requires the recomputation of some of the goal annotations. The lexical scoping can be altered, so the whole goal must have its quantification recomputed, and the delta-instantiation annotations of the branched goals will generally change too, so that information must be updated.

Finally, we note that the follow-code migration process can change the computed determinism of a goal. For example, consider the predicate:

```
:- pred p(int, int).
:- mode p(in, out) is ???.

p(X, Y) :-
    (
        q(X, Z)
    ;
        r(X, Z)
    ),
    Z > 0,
    s(Y).
```

We can assume for simplicity that q/2 and r/2 both deterministically produce Z and s/1 deterministically produces Y.

The current determinism analysis pass would consider the procedure to be nondeterministic because the disjunction has output bindings. However, if we apply the follow-code migration technique, we end up with the following version of the procedure:

```
:- pred p(int, int).
:- mode p(in, out) is ???.

p(X, Y) :-
    (
        q(X, Z),
        Z > 0
    ;
        r(X, Z),
        Z > 0
    ),
    s(Y).
```

Now the disjunction has no output bindings, so the determinism analysis would conclude that the procedure is semideterministic.

4.7 Liveness analysis

Liveness information is used by the code generator to help it optimize its use of storage. Only those values which are live, that is those that have a value and will be referenced again, need to occupy storage, so determining exactly when variables become live and when they become dead can have a big impact on the quality of generated code. In principle, liveness information need only be conservatively approximated — designating a variable as live before it in fact becomes live or after its last occurrence, should not cause the code generator to generate incorrect code, though it may well cause the generated code to use registers and stack less efficiently than it might otherwise. In practice, we compute the liveness information as exactly as possible, and allow the code generator to depend on its correctness.

To compute when a variable becomes live, we perform a simple forwards traversal of the goal. At each atomic subgoal we determine which nonlocal variables have occurred for the first time in a binding occurrence (we check the delta-instantiation map computed by the mode analysis), and put these in the set of variables that become live at that goal. This set, for reasons that shall be explained, is called the "pre-birth" set. The variables in this set are considered to become live during the goal. We also keep an accumulator of all the variables that have become live so far.

The accumulator of all the variables that have become live is initialized with the input variables of the procedure.

As a simple example of the computation of pre-birth sets, consider tree__contains/2. Here we show it with the accumulated liveness sets immediately before each sub-goal:

```
tree__contains(T1, T2) :-
    % { T1, T2 }
    not (
        % { T1, T2 }
        some [E] (
            % { T1, T2 }
            tree__member(T1, E),
            % { E, T1, T2 }
            not (
                % { E, T1, T2 }
                tree__member(T2, E)
            )
        )
    ).
```

Because the pre-birth for a goal only contains the variables that become live at that goal, all the pre-birth sets are empty except for the goal tree__member(T1, E) which has a pre-birth set containing only E.

Branched goals are handled by processing each branch independently, giving each the same initial set of live variables. If a branched goal is mode-correct, then the same set of variables should be live at the end of each branch.

If one of the branches of a branched goal contains a call to a predicate with a determinism of "erroneous" or "failure" then it is possible that it doesn't make all the variables that it should become live. To maintain the invariant that the branches should make the same set of variables live, any variables which should have been produced by a branch but were not are put into a "post-birth" set attached to the branch as a whole.

Consider a predicate such as tree__top_element/2 which was introduced in section 2.3:

```
tree__top_element(H_1, H_2) :-
```

```
    ( % det switch on `H_1'
        H_1 has functor empty/0,
        H_1 = empty,
        _E = H_2,
        V_726493 = "No top element",
        error(V_726493)
    ;
        H_1 has functor tree/3,
        H_1 = tree(E, _L, _R),
        H_2 = E
    ).
```

The difficulty here is that the first branch does not produce H_2, but the second branch does. To solve this problem, the pre-births are computed as usual, and we introduce post-birth sets (for conciseness, empty pre- and post-birth sets are omitted):

```
tree__top_element(H_1, H_2) :-
    ( % det switch on `H_1'
        H_1 has functor empty/0,
        H_1 = empty,
        _E = H_2,
        % pre-births: {V_726493}
        V_726493 = "No top element",
        error(V_726493)
        % post-births: {H_2}
    ;
        H_1 has functor tree/3,
        % pre-births: {E}
        H_1 = tree(E, _L, _R),
        % pre-births: {H_2}
        H_2 = E
    ).
```

Note that in the first disjunct, _E is local to the assignment, and is not considered to have become live. (Mode analysis treats free-free unifications as assignments to the dead variable so that problems do not occur.) H_2 is nonlocal to the assignment, but the liveness analysis will discover from the delta-inst information that this occurrence of H_2 is not a binding occurrence, so H_2 does not become live.

The presence of H_2 in the post-birth set at the end of the first disjunct indicates that H_2 should be considered to become live at this point, even though there was no producer for it.

To compute when a variable dies, that is when it has its last reference, we use much the same techniques as we did to compute liveness, except that instead of performing a forwards traversal of the goal, we traverse it backwards. The accumulator of variables for which we have seen a last occurrence is initialized with the output variables of the procedure. The variables that are nonlocal to a goal and do not already occur in the accumulator are put in the "post-death" set for the goal.

As an example, consider again the predicate tree__contains/2. To compute when the variables die, we perform a bottom up traversal, and accumulate nonlocal variable. Because tree__contains has no output variables, the initial accumulator is the empty set. Here we show the HLDS form of the goal and the accumulator at the end of each goal:

```
tree__contains(T1, T2) :-
    not (
        some [E] (
            not (
                tree__member(T2, E)
                % {E, T1}
            )
            % {E, T1}
            tree__member(T1, E),
            % {}
        )
        % {}
    ).
    % {}
```

This gives rise to the following post-death annotations (empty annotations are omitted):

```
tree__contains(T1, T2) :-
    not (
        some [E] (
            not (
                tree__member(T2, E)
                % post-death: {T2}
            )
            tree__member(T1, E),
            % post-death: {E, T1}
        )
    ).
```

For branched goals, we use a process very similar to that used for computing liveness. The accumulator is passed into the end of each of the branches and any variables that occur in some goals but not in a given goal are inserted into the pre-death set for that goal. As an example, consider the second clause of tree__member/2 (in the mode tree__member(in, out)):

```
    H_1 = tree(_E0, L, R),
    (
        tree__member(L, E),
        H_2 = E
    ;
        tree__member(R, E),
        H_2 = E
    )
```

If execution takes the first branch of the disjunction, R dies immediately after the deconstruction unification in which it is produced. On the other hand, if execution takes the second branch, L dies immediately. Clearly we need both to be live at the start of the disjunction.

As soon as we have chosen one disjunct or the other we can safely discard R (if we take the first disjunct) or L (if we take the second). Therefore, we give the following deadness annotations to the goal:

```
        H_1 = tree(_E0, L, R),
        % post-death: {H_1}
        (
            % pre-death: {R}
            tree__member(L, E),
            % post-death: {L}
            H_2 = E
            % post-death: {E}
        ;
            % pre-death: {L}
            tree__member(R, E),
```

```
        % post-death: {R}
        H_2 = E
        % post-death: {E}
    )
```

Notice that while it is only in programs containing calls to predicates with a determinism of 'erroneous' or "failure" that can contain variables in the post-birth sets, any program that contains a branched structure will usually have variables in its pre-death sets.

If-then-else is a special case of a branched goal. To understand how to correctly annotate an if-then-else we view it as a disjunction:

```
(
    Cond,
    Then
;
    not Cond,
    Else
)
```

rather than

```
(
    Cond
->
    Then
;
    Else
)
```

which is really an abbreviation for the disjunctive form.

Because the negated form of the condition cannot have any output variables, no variables that become live in the condition are considered live in the Else goal.

In practice, this means that to compute liveness, we thread the accumulator through the Cond goal and into the Then goal, and we thread it around the Cond goal and into the Else goal.

To compute deadness for if-then-else, we thread the accumulator backwards through the Then and Else goals separately, but merge them before threading through the Cond goal.

The terms "pre-birth" and "post-death" are used because if a variable becomes live during a goal then it will need storage during the goal, and if a variable becomes dead during a goal, it will still need storage for some portion of the execution of that goal.

4.8 Determinism analysis

The determinism analysis pass computes the determinism of each subgoal. The determinism of calls are given by the declarations of their modes, and the determinism of unifications are given by their class. The determinism of compound goals is computed from the determinism of their components according to the rules for the particular construct. Each goal is annotated with its computed determinism. Switches are also annotated with their local determinism which states whether or not the cases for the switch are exhaustive or not.

If the computed determinism does not match the declared determinism of a predicate mode, then an error is reported, and that mode of the predicate is rejected.

In certain instances it is possible to improve the estimate of the determinism of a goal computed by this simple algorithm. If a nondeterministic goal has no output variables, then its external behaviour is semideterministic — there is no way to distinguish the different successes of the goal, so the containing goal merely observes that the goal either succeeds of fails. When generating code for such goals, the code generator is able to insert code to prune the search tree without compromising the semantics of the program.

An example of where nondeterminism is contained in this way occurs in tree__contains. The HLDS form of it was shown in section 4.7. Here the first call to tree__member is in the nondeterministic mode where the first argument is input and the second output. However, the binding of E produced in the nondeterministic call does not escape from the existential quantifier, so the quantified expression

```
some [E] (
    tree__member(T1, E), % nondet
    not [] (
        tree__member(T2, E) % semidet
    )
)
```

has semideterministic behaviour. In this case the point is moot because if a negation is well moded, no bindings can escape it, so the enclosing negation ensures that the behaviour is semideterministic, regardless of the determinism of the negated goal.

The current determinism analysis algorithm only detects that nondeterministic goals have semideterministic behaviour if they have no output variables at all. An improved implementation would detect that if each output variable of a nondeterministic goal was deterministically produced, in which case multiple successes of the goal are also indistinguishable, and the search tree can be pruned.

The determinism annotations are used by the code generator to decide the appropriate way to generate code for each subgoal.

4.9 Follow-vars

Crucial to generating efficient code is the allocation of storage resources. If the registers are not effectively used, then it is impossible to achieve good efficiency. The follow-vars pass computes information to give hints to the storage allocator about where variables should be stored.

When execution reaches a predicate call, input argument variables must be located in specific registers. The follow-vars pass propagates information about these variables and in which registers they must be stored backwards through the predicate, so that at each point in the program the code generator knows which variables must be stored in specific places at the next call, and where. This information is then used during storage allocation to improve the annotations that it generates. The rationale is that if we can store a variable in the register where it will have to be at the next call, then we can avoid having to move it.

As an example, consider the HLDS form of a predicate to compute the depth of a tree:

```
tree__depth(H_1, H_2) :-
    ( % det switch on 'H_1'
        H_1 has functor empty/0,
        H_1 = empty,
        H_2 = 0
    ;
        H_1 has functor tree/3,
        H_1 = tree(_E, L, R),
        tree__depth(L, LD),
        tree__depth(R, DR),
        int__max(LD, DR, D0),
        V_167611 = 1,
        builtin_plus(D0, V_167611, D),
        H_2 = D
    ).
```

The follow-vars set initially contains the output variables of the procedure and where they should be stored. In this case there is only one: H_2 which should be stored in r2. Thus the initial set is {(H_2, r2)}.

Traversing the second case, for which H_1 has the tree/3 functor, when we traverse the conjunction backwards the first call we encounter is int__max(LD, DR, D0). It is annotated with the current follow-vars set {(H_2, r2)}, and the current follow-vars set is replaced with the input variables of the call and where they should reside, namely {(LD, r1), (DR, r2)}.

Continuing the traversal, we encounter the call tree__depth(R, DR), which is annotated with the follow-vars set {(LD, r1), (DR, r2)}, and the current follow-vars set becomes {(R, r1)}.

This process continues to the start of the conjunction, and then the first branch is processed in the same fashion using the same initial follow-vars set. To produce the initial follow-vars set which is stored as an annotation to the procedure as a whole, the two follow-vars sets that result from the processing of the disjunction need to be merged in some way. Since the follow-vars annotations are only advisory, we simply take the follow-vars set which results from the processing of the first disjunct. In the current procedure case this results in a sub-optimal annotation for the procedure because the information computed in the first disjunct is not as useful as that computed in the second. Future work may attempt to merge the follow-vars sets to make the computation more effective for branched goals.

4.10 Storage allocation

At predicate calls, the input arguments to the call must be stored in specific registers. Similarly, at the end of the procedure, the output arguments must be stored in specific registers. These specific locations are fixed as part of the current execution model. For all other situations where the code generator must generate code to store values, it is free choose where the values are stored. These situations can be divided into three kinds:

- At the start of disjunctions, all live variables must be allocated storage on the stack so that on backtracking their values are available.
- At procedure calls, variables that are live after the call must be saved, and are allocated stack locations accordingly.
- At the end of branched goals all live variables are allocated registers. This is so that variables are stored in consistent places in the different branches of a branched goal. This allocation is aided by the information from the follow-vars pass.

The current implementation of storage allocation for stack locations is somewhat naive — it makes no attempt to reuse stack locations for different variables; each variable is allocated a unique stack location. Additionally, the allocation of stack locations at disjunctions could be optimized by allocating storage only for variables that are used in disjuncts other than the first. Currently, these optimizations are not made in the interest of simplicity.

As an example of how storage is allocated, consider tree__member/2. The HLDS representation of the semideterministic mode in which both arguments are input is:

```
tree__member(H_1, H_2) :-
    (
        H_1 = tree(E, _L, _R),
        H_2 = E
    ;
        H_1 = tree(_E0, L, R),
        H_2 = E,
        (
            tree__member(L, E)
        ;
            tree__member(R, E)
        )
    ).
```

At the beginning of the outer disjunction, H_1 and H_2 are live and must be saved. Neither will be in the follow-vars set, so we assign a location in the nondeterministic stack frame for each. At the start of the inner disjunction L, R, and E are live, and since none has a stack location, each is assigned a unique stack location.

The information computed by the storage allocation pass is attached to each goal for which it was computed. Goals for which no annotations are produced are marked as having an empty annotation.

4.11 Code generation

Code generation for Mercury is surprisingly straight forward for the most part. Once all the analysis passes have been completed, the code generator is presented with an ordered goal containing annotations that give extensive information to aid the generation of compact and efficient code.

The design of the code generator is product of the set of constraints within which it must work. In this respect the code generator for Mercury is rather unusual — it is much less constrained than code generators for conventional languages. Some of the important factors in this are:

- Mercury programs have transparency of reference. Within the constraints of the immediate data dependencies, Mercury programs can be evaluated in any order.
- The execution model for Mercury provides an arbitrary number of virtual registers (subject to the constraint that lower numbered registers are more likely to have fast access than higher numbered ones).
- The target language is architecture independent (C), so apart from general strategies like increasing the length of basic blocks as much as possible, very low level optimisations such as scheduling for a particular pipeline do not need to be considered.

The choice of data structures can have a huge impact on the rest of the design of a program, and the code generator is no exception. The data structures used in the code generator represent a second generation of development. The initial design proved inadequate, and the present data structures build on the lessons learned from the weaknesses in the first design. Even so, the data structures will need modification to support the full implementation of our current approach to code generation.

The details of the data structures used in the current implementation are presented in section 4.11.1. The techniques used to generate each kind of goal in the HLDS are discussed in sections 4.11.3 to 4.11.13.

4.11.1 Data structures

The larger part of the code generator is written using DCG notation to thread the main data structure through the code generator.

DCG notation allows us to write a predicate like:

```
p(W, Z) -->
    q(W, X),
    { r(X, Y) },
    s(Y, Z).
```

which is expanded by the compiler into the following:

```
p(W, Z, V0, V2) :-
    q(W, X, V0, V1),
    r(X, Y),
    s(Y, Z, V1, V2).
```

Goals written inside braces are not expanded (like the call to r/2). All other goals are given two extra arguments and a series of variables are introduced so that the output argument from one call becomes the input argument to the next as is shown in the example by the numbering of the variables.

The main code generator data structure contains all the information that the code generator needs about such things as the contents of the registers, and the status of each variable. In many respects this data structure represents the "state" of the code generator.

The main components of the main code generator data structure (called the code_info structure in the source code, see appendix C) are:

- A mapping from registers to register status. For each register we maintain a set of variables whose value is currently stored in that register or we either record that it has been reserved. By reserved, we mean that although the register does not yet contain a value, one part of code generator wishes to prevent other parts of the code generator from allocating it.
- A mapping from variables to variable status. For each variable we record either the set of lvalues in which its value is stored, or we record a cached expression to generate its value. With each cached expression we also store whether or not we have a target location into which we should put the value of the expression when we generate it, and if so, the location.
- The current store-map indicating where variables should be stored on the stack and at the end of branched goals.
- The set of variables that are live at the end of the goal currently being processed.
- A counter for labels, so that new unique labels can be allocated.
- A counter for the number of stack locations used by the procedure.
- A copy of the HLDS structure for the current module.
- The identifiers for the current predicate and predicate mode.
- A stack of labels for failure continuations in semideterministic and nondeterministic code.

In some situations, the code generator needs to move the value stored in a register because that register is needed (e.g. for a procedure call). In these situations the mapping from register to status is used so that the code generator can correctly update the mapping from variable to status.

When the value in a register needs to be moved, we check to see if any of the variables that have their value stored in that register are live and do not currently have their value stored in any other location. Only in these circumstances do we need to actually generate instructions to copy the value of the register to a new location. The entries in the variable map for each of the variables stored in the register are then set to point to the new location of the value. Because Mercury is a single assignment language and variables cannot be updated, it is safe to change the storage location of variables in this way.

To generate a label, the code generator obtains a new label number from the code_info structure and increments the counter in the new copy of the structure. The new label is formed from the module name, the predicate name, the predicate arity, the number of the mode of the predicate being generated and the label number. Thus for the second declared mode of the predicate `p/3` in the module `m` we generate the labels `m_p_3_1_0`, `m_p_3_1_1`, `m_p_3_1_2` and so on. Given our current conventions for module and predicate names, this can give rise to labels that do not differ in the first 32 characters, which causes problems for some broken linkers. This problem could be solved by using name mangling to compress the names, however there is an increasing tendency for language tools to handle long identifier names correctly (this is due in part to the popularity of C++ which does name mangling to encode overloading and templates and sometimes generates long identifier names).

It is possible in the current Mercury syntax to include characters in predicate names that are not valid characters for a C identifier (for instance, it is valid to name a predicate "`'this is a predicate name'`"). To overcome this problem we perform name mangling on names that contain such characters.

The execution model as presented in section 3 pushed values onto the stack and popped them off as required. In practice it is simpler and more efficient to perform a single increment of the stack pointer in the procedure prologue, and to perform a single decrement in the procedure epilogue. This can result in greater stack consumption, but it simplifies the management of the stack, and so long as stack allocation is done reasonably carefully the extra cost is negligible in most cases. Like some modern compilers for imperative languages, Mercury stack frames do not use a frame pointer which decreases the overheads for procedure calls. This is particularly important in a language which makes heavy use of recursion.

Even though we use this approach of incrementing and decrementing the stack pointer once, the code generator maintains a counter of stack slot usage. This enables the code generator to conceptually push and pop things on and off the stack, though in practice this increments the stack slot usage, and causes a stack slot to be statically allocated. This is used only in places where the code generator needs to save a special purpose register such as the heap pointer or the nondet stack pointer, as all other stack usage is determined before the main part of code generation in the storage allocation pass.

The stack of labels in the code_info structure is used to determine where to branch to on failure in semideterministic and nondeterministic contexts. This is discussed further in sections 4.11.6, 4.11.7, and 4.11.8.

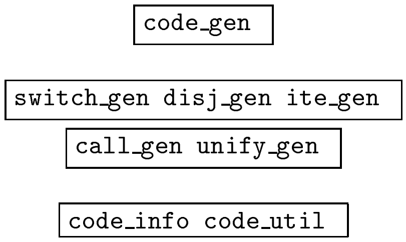

Figure 3: The structure of the code generator

4.11.2 Code generator organisation

The code generator is organised into eight modules in three conceptual layers. At the bottom level, the code_info module defines the main code generation data structures and provides predicate for managing them. It also provides predicates for producing many of the code sequences required by other parts of the compiler. Also at the bottom level, the code_util module provides various miscellaneous predicates used during code generation that do not depend on the code_info data structures themselves.

The top level of the code generator is the code_gen module, which as well as supplying the top level of the code generator which is called by the main compilation path, also provides predicates to generate arbitrary goals. These are called by the other code generation modules which each generate a particular kind of goal. The code_gen module generates code for the simplest compound goals directly — conjunctions, negations and quantifiers are all handled within code_gen, while other kinds of goal are forwarded to the module that generates code for them. These modules are: call_gen for generating predicate calls and out-of-line unification calls, unify_gen for generating all the in-line forms of unification (assignment, simple tests, constructions, and deconstructions), switch_gen for generating switches, disj_gen for generating disjunctions, and ite_gen for generating if-then-else constructs.

The last three of the modules listed above, switch_gen, disj_gen, and ite_gen generate compound goals. They contain calls back to code_gen to generate the component goals, and they import code_gen, which creates recursive dependencies. By having separate modules for working with each of the goal types, the task of generating efficient code for each becomes more manageable: each module can be enhanced independently of the others.

To generate code for a procedure, the code generator first initialises its data structures based on the modes of the arguments to the predicate. Arguments for which the top type node is in an input mode are marked as being live and stored in the register determined by the argument position and the procedure's determinism. A call to generate the goal is executed. This call produces the code for the body of the procedure. Once the body of the procedure has been generated, the procedure prologue and epilogue are generated.

4.11.3 Predicate calls

Procedure calls in Mercury have much in common with procedure calls in C or Pascal, and there is much in common between calls of different determinism.

In Mercury, procedure arguments are passed and returned in registers, and the registers are caller-save. To make a procedure call, the code generator must emit code to save the variables that are live after the call onto the stack, position the arguments in the appropriate registers and branch to the procedure by invoking a macro containing the address of the procedure to be called, and the address to which control should return. In these respects procedure calls in Mercury are similar to those in C (except that many C implementations pass arguments on the stack rather than in registers).

To determine into which stack location each live variable should be stored, the code generator consults the current store-map which will contain the stack location for each variable computed during the storage allocation.

Once the variables that are live after the call have been saved onto the stack, the input arguments to the procedure are generated and positioned in their appropriate registers (the allocation of arguments to registers is discussed below). These two operations are done in this order not the other way around so that more low numbered (fast) virtual registers will be available for the generation of the live values: if the input arguments were positioned first, the registers which they occupied would be unavailable, which would mean that higher numbered virtual registers would have to be used. This in turn would in general result in a more accesses to a larger number of virtual registers, which on machines with limited numbers of real registers available, would mean that even an optimal assignment of real registers to virtual registers would have more references to non-real registers (global variables in memory). Since an access to a non-real register is significantly more expensive than an access to a real register, reducing the number of references to non-real registers can result in a substantial improvement in performance.

Once the live values are saved and the input arguments are positioned, the call can be made. To make the call, the code generator must allocate a label to which control should return after the procedure call.

The call is executed at the C level by a macro which takes the addresses of the called predicate and a return continuation as parameters. It assigns the return continuation to the succip register, and branches to the called predicate.

The call macro has the following definition (simplified for presentation):

```
#define call(label, succcont) \
    do {                      \
        succip = (succcont);  \
        GOTO_LABEL(label);    \
    } while (0)
```

The code fragment is placed inside a do-while(0) construct so that the macro behaves correctly when used in contexts such as the statement following an if-then-else where the macro must behave as a single statement. Simply putting braces around the code fragment is insufficient, because the semicolon used after the macro no longer associates to the macro.

As an example of how calls are generated, consider the code for the predicate tree__depth:

```
tree__depth(H_1, H_2) :-
    ( % det switch on 'H_1'
        H_1 has functor empty/0,
        (
            H_1 = empty,
            H_2 = 0

        )
    ;
        H_1 has functor tree/3,
        (
            H_1 = tree(_E, L, R),
            tree__depth(L, LD),
            tree__depth(R, DR),
            int__max(LD, DR, D0),
            V_167611 = 1,
            builtin_plus(D0, V_167611, D),
            H_2 = D
        )
    ).
```

Just before the code fragment we show below, the only live variable is H_1. The code to perform the call tree__depth(L, LD) in tree__depth/2 as generated by the compiler (including the comments) looks like:

```
    { detstackvar(2) = (int) field(mktag(1),
            (int) r1, 2); }
        /* Assigning from R */
    { r2 = (int) r1; }
        /* Swap live variable to a new register */
    { r1 = (int) field(mktag(1), (int) r2, 1); }
        /* Assigning from L */
    { call(mercury__tree__depth_2_0,
            LABEL(mercury__tree__depth_2_0_i3)); }
        /* branch to det procedure */
mercury__tree__depth_2_0_i3:
    ;
        /* Continuation label */
```

The detstackvar macro has the following definition:

```
#define detstackvar(n)  sp[-n]
```

The first assignment in the generated code fragment copies the third argument of H_1 which is stored in r1 into the stack location offset two from the stack top (field arguments are numbered from zero). This assignment saves R onto the stack since it will be needed for the second recursive call to tree__depth/2.

The next two assignments place L in r1 as the input argument to the call. The assignment that copies the value of r1 into r2 is redundant, however it would be difficult to make the code generator avoid some such assignments. Once the value numbering pass has been completed, assignments such as this one will be optimised away.

In general it may be necessary to shuffle registers around while setting up the parameters to a call. The shuffling of registers is managed by a predicate in code_info called code_info__shuffle_register which produces code to place the value of a given variable in a given register, and takes a list of all the arguments to the current procedure call as auxiliary input. To see how it works, assume we have a deterministic call:

```
p(W, X, Y, Z)
```

in the mode p(in, in, in, out) and immediately before the call the status of variables is:

Variable	Location(s)
W	r4
X	r1, r3
Y	r2

To set up the input arguments to this call, we first try to position W and find that r1 already contains a value that we need (the value of X). However, closer examination reveals that since the value of X is stored somewhere else, we can overwrite r1 with the value of W. With this operation done, the status of variables is:

Variable	Location(s)
W	r1, r4
X	r3
Y	r2

As we setup the second argument X, we find that r2 is occupied, and that the value that is stored there (the value of Y) is not stored anywhere else, and is still needed. Therefore we need to emit code to copy it to somewhere else. Since it is an argument that we have not yet placed, we look to see if it can be placed in the right register straight away, or if we will need to do more shuffling. We notice that the register into which we would like to put it is occupied by a value that we cannot overwrite, so we find the lowest numbered free register which is r5, and copy the value of r2 there. Now we can safely copy the value of X from r3 to r2 and the status of the variables is:

Variable	Location(s)
W	r1, r4
X	r2, r3
Y	r5

Now we try to position Y, and find that r3 is occupied, but that the value that is stored there is duplicated elsewhere, so r3 can be overwritten.

The code sequence that is emitted for this call is:

```
{ r1 = (int) r4; }
    /* Assigning from W */
{ r5 = (int) r2; }
    /* Swap live variable to a new register */
{ r2 = (int) r3; }
    /* Assigning from X */
{ r3 = (int) r5; }
    /* Assigning from Y */
{ call(mercury__p_4_0_1,
    LABEL(mercury__q_4_0_i2)); }
        /* branch to det procedure */
mercury__q_4_0_i2:
    ;
```

Although the shuffling process has been described in terms of modifications to the mapping from variables to variable status, the implementation uses both that mapping, and the mapping from registers to register status. The two are not quite the inverse of each other, since variables can be stored in places other than registers. This complicates their interactions.

The sequence of assignments generated to position the arguments for the call shown above is fairly efficient, however it is not optimal. The optimal sequence based on the given information would be:

```
r3 = r2;
r2 = r1;
r1 = r4;
```

It would be possible for the code generator to be built to find such sequences, however it would be relatively complex, and processes such as value numbering are able to discover the redundancy that the generated code contains and remove it.

In section 4.11.5 we introduce some more involved aspects of the design of the code generator which have a strong influence on the precise code sequences produced for calls.

For deterministic and nondeterministic procedures, the input arguments are positioned in the register corresponding to their argument numbers (the first argument in register one, and so on). Semideterministic procedures return an indicator of their success or failure in register one, and the arguments are positioned so that the first input argument is positioned in register two, but as before, the arguments are positioned in sequential registers.

On return from the called procedure, only registers containing the output arguments of the called procedure and variables saved on the stack are live. The components of the code_info data structure that maintain the code generator's view of the status of each register and variable are rebuilt.

Once a deterministic or nondeterministic procedure returns, execution simply proceeds from the continuation label, and no further work is needed. When a semideterministic procedure returns, code is placed immediately after the continuation label to test the value in register one (which will be the success/failure indication), and either fail, or drop through.

To show how this works, consider the semideterministic call to tree__member/2 in tree__contains/2:

```
tree__member(T2, E)
```

We omit the code sequence to save the variables that are live across the call or to set up the input arguments (which will go in registers r2 and r3). Here is the code sequence to perform the call and test whether it succeeded:

```
    { call(mercury__tree__member_2_0,
        LABEL(mercury__tree__contains_2_0_i6)); }
        /* branch to semidet procedure */
mercury__tree__contains_2_0_i6:
    ;
        /* Continuation label */
    { if ((int) r1)
       GOTO_LABEL(mercury__tree__contains_2_0_i7); }
            /* Test for success */
    { GOTO_LABEL(mercury__tree__contains_2_0_i5); }
        /* fail */
```

In general, the code to perform the 'fail' depends on the context. This is discussed in section 4.11.6.

Calls to builtin goals are generated as inline code. For instance, the call `D is D0 + 1` in tree_depth/2 has the following code generated (r3 initially contains D0, and D is in r2 at the end of the code fragment):

```
    { r1 = 1; }
        /* Make a constant */
    { r2 = ((int) r3 + (int) r1); }
        /* Evaluate binary expression */
```

Calls to the builtin comparison operators >, <, =<, and >= result in code for conditional failure (see section 4.11.6).

4.11.4 Unifications

The compiler classifies unifications into five different classes (as discussed in section 2.2): assignments, simple tests, constructions, deconstructions and complicated unifications. The assignments and constructions are deterministic, so we discuss these first.

Assignments are trivially handled. They are deterministic and are effectively introducing a new name for a value. This view of assignment is reflected in the implementation of assignment: no code is emitted for assignments. When an assignment is encountered, the code generator creates an entry in its mapping from variable to status which indicates that the value of the newly bound variable is stored in the same location as the value of the already bound variable, and the entry in the mapping from register to status is updated to reflect the fact that the newly bound variable is found in the same places as the already bound variable.

A construction unification is a unification between a free variable and a term. Construction unifications construct new terms. This includes integers, constants, floating point numbers and strings.

The construction of integers and enumeration types is trivial: an assignment of the appropriate integer value to the variable is all that is required.

To construct strings, a macro which takes the string as a C style string constant and the length of the string is called, and it returns a Mercury string constant. (The current implementation of this macro discards the length, and returns the C style string constant: Mercury strings are currently implemented as C strings.)

The construction of floating point terms is not yet implemented.

For other zero arity terms, the code generator looks up the appropriate table in the HLDS structure and composes a word with the right primary and local secondary tags. None of these instances of construction unification require the allocation of heap memory.

To allocate the memory required for a new term of non-zero arity, the code generator checks to see if the term uses a remote secondary tag. If it does not, then the number of words need for the term is given by the arity of the term. If the term uses a remote secondary tag then an extra word is needed. This memory is allocated by taking the value of the heap pointer as the address of the allocated memory, and emitting code to increment it by the appropriate number of words.

To tag the newly allocated word, first the primary tag is set, then if it exists, the remote secondary tag is set.

The arguments of the new term may or may not be filled, depending on the exact mode of the unification. For each argument that is filled, an assignment to the appropriate field of the term is generated.

As an example, consider the code to construct a new tree/3 term:

```
T = tree(1, empty, empty)
```

The compiler assigns the empty/0 functor a primary tag of 0, and tree/3 a primary tag of 1. Neither uses a secondary tag. The heap pointer is incremented to allocate the heap space with a macro incr_hp so that if bounds checking is needed, it can be done transparently.

The code sequence that the compiler emits for this construction leaving T in r4 is:

```
{ r1 = 1; }
    /* Make a constant */
{ r2 = mkword(mktag(0), mkbody(0)); }
    /* Tag a word */
{ r3 = mkword(mktag(0), mkbody(0)); }
    /* Tag a word */
{ r4 = mkword(mktag(1), (int) hp); }
incr_hp(3);
    /* Allocate heap space & tag the pointer */
{ field(mktag(1), r4, 0) = (int) r1; }
    /* Assigning from variable number 1367647 */
{ field(mktag(1), r4, 1) = (int) r2; }
    /* Assigning from variable number 120422 */
{ field(mktag(1), r4, 2) = (int) r3; }
    /* Assigning from variable number 726493 */
```

Deconstruction unifications are the unifications of a bound variable with a term. These unifications can be semideterministic: the test that the functors match can fail. If there is only one functor for the type or the deconstruction follows the functor test in a switch, then the test can be omitted and the unification is deterministic.

As with construction unifications, the builtin types are handled specially. The deconstruction unification of integers simply tests that the value of the variable is equal to the constant term. The code emitted for such unifications performs a test and then either fails or drops through. Failure is discussed in section 4.11.6. The deconstruction unification of characters, and types composed only of constants are handled in the same way.

The deconstruction unification of strings is handled by a call to a procedure to test that the two strings are equal, and to fail if they are not.

The deconstruction of other zero arity terms is composed of a primary tag test, followed if necessary by a secondary tag test. For the deconstruction unification of a variable and a term of nonzero arity, this functor test is followed by the extraction or assignment of arguments.

The unification of each of the arguments and the corresponding part of the variable is one of the following: a "don't care" unification for which no code is emitted, an assignment from a field of the variable to the corresponding variable in the term, or an assignment from a variable in the term to the corresponding field of the variable.

For example, consider the deconstruction of the tree/3 term at the start of the second disjunct for tree__member/2 in its nondeterministic mode:

```
tree__member(H_1, H_2) :-
    (
        H_1 = tree(E, _L, _R),
        H_2 = E
    ;
        H_1 = tree(_E0, L, R),
        (
            tree__member(L, V_E_1)
        ;
            tree__member(R, V_E_1)
        ),
        H_2 = V_E_1
    ).
```

The first argument of the unification has a don't care mode, and the second and third fields are assignments from the fields of H_1.

In the following code generated for this deconstruction, H_1 is initially stored on the nondeterministic stack in slot 0.

```
/* checking that H_1 has functor 'tree'/3 */
    { if ((tag((int) framevar(0)) != mktag(1)))
        fail(); }
        /* test for failure */
    { framevar(1) = (int) field(mktag(1),
            (int) framevar(0), 2); }
        /* Assigning from R */
    { framevar(2) = (int) field(mktag(1),
            (int) framevar(0), 1); }
        /* Assigning from L */
```

The framevar macro references variables on the nondeterministic stack. Its definition is:

```
#define framevar(n)    curfr[SAVEVAL-n]
```

SAVEVAL is the offset at which the saved values begin — as discussed in section 3.3 the top few slots of each nondeterministic stack frame are used to store the fixed fields such as the redoip.

Simple test unifications take place between variables of builtin types and enumeration types.

To unify two integer, character or enumeration variables in the input-input mode, the compiler uses C's builtin integer test operator `==`. If the test fails, then code to fail appropriate to the context is executed, otherwise control drops through.

To perform a simple test on two strings, a call to an out-of-line string equality test procedure is emitted. The return value of this call is tested and code to fail if the test fails is emitted. Once again, if the test succeeds, control drops through.

The simple test unification of two floats is unimplemented.

The only other kind of unification is the unification two variables in a mode pair other than (in, out), (out, in). Complicated unification is what might be called "real" unification. Unifications of this kind are performed by specially generated code in out-of-line procedures. Thus the work of generating them falls into two parts: generating calls to out-of-line unification procedures, and generating the out-of-line procedures.

The first part of this work is simple: the unification is treated as a semideterministic call with the two variables as input arguments The code to generate these calls is located in call_gen and is no more than a specialised form of semideterministic call.

The task of generating the code for the out-of-line procedures is conceptually quite straight forward. The idea is to generate a switch on the first variable, and in each case of the switch, fully deconstruct the first variable, then unify the functor and each field with the corresponding part of the other variable.

For example, the code for the unification of two trees in the (in, in) mode pair is:

```
tree__unify(H_1, H_2) :-
    (
        H_1 = empty,
        H_2 = empty
    ;
        H_1 = tree(V_120422, V_726493, V_1425279),
        H_2 = tree(V_1153084, V_895589, V_715794),
        V_120422 = V_1153084,
        tree__unify(V_726493, V_895589),
        tree__unify(V_1425279, V_715794)
    ).
```

This formulation generalises not only to the (in,in) mode pair, but to more complicated modes in which the recursive calls may have different modes to the mode of the head of the predicate.

The information of what complicated unification procedures will be required is discovered during mode analysis, and it is during mode analysis that the compiler constructs these unification procedures. From the point of view of code generation, there is nothing magic about these unifications: they are treated as ordinary procedures.

However, the procedure above does contain a subtle problem: it is polymorphic, and requires the (in,in) complicated unification of two variables of the unknown type. The right solution is to implement higher order procedures, and pass the address of the unification routine appropriate to the type of the arguments as a hidden argument to the procedure. In general, one hidden argument is needed for each type variable. Currently, code to generate a runtime abort is produced for polymorphic complicated unifications.

4.11.5 Conjunctions

Conjunctions are the most basic compound goal. To generate code for them, the code generator simply has to generate code for the component goals, one after another, threading the code_info data structure through the code generator as it generates code for each goal. The process of generating a conjunction does not depend on its determinism.

The delta instantiation of each of the component goals is examined to see if the instantiation goes to unreachable. This can happen in the presence of goals with an annotation of 'erroneous'. If the delta instantiation of a subgoal of a conjunction is 'unreachable', then no goals in the conjunction after that goal have code generated for them. This is for two reasons: the code could never be reached, so there would be no gain in producing it, and because unexpected things can happen, such as code trying to use a variable that hasn't been produced. It would require considerable effort to make the code generator stand up to such unexpected situations, and such effort is not warranted since there is the easy solution indicated above of examining the delta instantiation.

One of the difficulties in producing efficient code for conjunctions is that when generating each goal, the code generator has no clear guide to where later goals in the conjunction will need variables that the current goal produces. This means that redundant register to register copies will often be needed to move values to where they will be needed, and to move values out of the way.

To address this difficulty, we have designed a technique which takes advantage of the declarative nature of Mercury, which enables goals to be evaluated in any order, subject to data dependencies. We are able to use this property to greatly simplify the task of register allocation — one of the more difficult aspects of high performance compiler implementation.

We call our technique "lazy code generation" (that is generating code lazily, rather than generating lazy code). Inspection of the code that it produces indicates that it yields very good code for a modest cost in the complexity of the compiler.

The primary motivation for adopting this unconventional approach to code generation was the difficulty of achieving efficient usage of the limited number of real registers available (even though our virtual machine presents arbitrarily many). Techniques such as graph colouring seemed like overkill for a preliminary implementation, but we still needed some strategy for deciding where to store computed expressions.

The approach we use is to build up a cache of expressions that produce variables (which may be used in the expressions for other variables, and so on). These expressions are derived from the builtin goals for arithmetic and assignment unifications. When at the point when the code generator needs the value of a variable, it almost always knows where the value should be stored. Thus, it can generate the values of the dependencies and the final result directly into the desired locations. This way we avoid performing redundant assignments to copy values from their initial locations to where they need to be.

When the result of an expression is needed, the code generator first generates code for any variables on which the expression depends before generating code for the current expression. Thus, internally, the code generator produces code for builtin predicates in a backwards fashion. While the code generator is evaluating the dependencies, the target register of the expression is marked as reserved so that it isn't allocated for the evaluation of some other part of the expression. Because the cache is in a transient state, the register is not given a status containing the variable to prevent the cache containing incorrect information.

When a semideterministic conjunct (builtin or otherwise) is encountered, code to evaluate it is generated immediately. For efficiency, tests should be generated as soon as possible, since we would like to detect failure as soon as possible to avoid unnecessary computation.

The decision to use lazy evaluation has a big impact on the code generator data structures. Instead of containing a simple mapping from variable to storage location, the code generator data structure must contain a mapping from variable to variable status. The variable status is a disjoint union that represents the following possible states that a variable may have:

- evaluated and where it is stored
- unevaluated and cached and optionally where it should be stored when it is flushed from the cache

When a predicate call is encountered in a conjunction, either all the cached expressions for variables that are live across the call should be flushed so that they can be saved, or the cached expressions could be retained in the cache, and only their dependencies be saved.

It is not difficult to construct examples that favour either approach. Intuitively, the former approach favours conjunctions which consume values before a call, reducing the number of live variables at the call; the latter approach favours conjunctions which produce many values that are then live across the call. The latter option would complicate the design of the call_gen module considerably so the former has been implemented.

In terms of performance or stack usage, there is little to choose between the two, though the latter form has not been implemented so we have no hard figures to confirm this.

For comparison we also implemented an "eager" version of code generation in which the expression cache is flushed after each atomic goal. In cases where the code generator doesn't have information about where to store a value, it simply chooses the lowest numbered free register. While it is true that it is possible to do better than this, to do so would incur significant extra effort, and given the good performance of lazy code generation, this was deemed unnecessary.

4.11.6 Failure

Up to now the handling of failure has been glossed over somewhat. The reason is this: the correct way to handle failure depends on the local context of the failure. The main problem is that failure has different operational effects depending on the context: if the context is the condition of an if-then-else, we want computation to continue at the else part of the construct, but if the context was the then part of an if-then-else construct, the desired behaviour would be different.

The task of handling these situations is simplified by having a stack of failure continuations in the code_info structure. Each failure continuation is one of:

- known(*label*)
- unknown
- do_fail

If the failure continuation is known, then the result is a branch to the label indicated. If the failure continuation is unknown, the result is a call to the redo macro which branches to the address pointed to by the redoip of the top nondeterministic stack frame. If the failure continuation is do_fail then the result is an invocation of the fail macro which pops the nondeterministic stack and uses the redoip of what becomes the current top stack frame as the address at which forward execution resumes.

In semideterministic predicates, we allocate a label as the failure continuation, and use that label in the procedure epilogue to set the result register to FALSE and return from the procedure. This label is pushed as known(*label*) onto the failure continuation stack, so that calls to code_info__generate_failure result in a branch to this part of the procedure epilogue. If an inner context in a semideterministic procedure wishes to capture failure (such as happens in a negation, or a disjunction), the inner context will push a new failure continuation onto the failure continuation stack, and pop it off at the end of the context.

For nondeterministic procedures, we initially push do_fail onto the failure continuation stack, since unless we specifically need to do otherwise, calling do_fail is the appropriate action.

The other variations on the use of the failure continuation stack are discussed below in the context of disjunctions, if-then-else and negation.

4.11.7 Disjunctions

There are two kinds of disjunctions: semideterministic ones that have no nonlocal output bindings, and nondeterministic ones which may bind nonlocal variables.

For a disjunction to be nondeterministic it must have output bindings and it must have at least two disjuncts. If it has no disjuncts then it cannot have any output bindings, and is semideterministic. If it has only one disjunct, then it is equivalent to just the disjunct itself — in fact it is not syntactically possible to create a disjunction with a single disjunct.

So, given that a disjunction has at least two disjuncts, and has output bindings, how do we go about generating code for it?

The first issue to consider is that each disjunct after the first must have access to all the variables that are live at the start of the disjunction, regardless of the behaviour of the disjuncts that precedes it. Therefore, the first thing that happens when a disjunction is encountered is that all live variables are saved onto the stack. Which stack they get saved onto depends on the procedure context of the disjunction.

If the disjunction occurs in a deterministic or semideterministic procedure, then it must occur within a subcontext that does not have any output variables. For example, consider the goal (given that T1, T2 and T3 are ground):

```
(
    tree__member(T1, X)
;
    tree__member(T2, X)
),
tree__member(T3, X)
```

Assuming that these are the only occurrences of X, the goal as a whole is semideterministic, even though the disjunction has an output binding (X).

In this case, the compiler will insert a commit across the disjunction. Therefore, it is fine for T1, T2, and T3 to be saved onto the deterministic stack (for the semideterministic goal), and for them to be popped off whether the procedure succeeds or fails. It should be intuitive that if a nondeterministic goal occurs inside a deterministic or semideterministic procedure it must have its output bindings restricted, and the compiler will commit to the first solution (except of course for occurrences of all-solutions predicates which are currently unimplemented). Consequently, the live variables can be saved on the stack appropriate to the procedure, and no extra work is needed

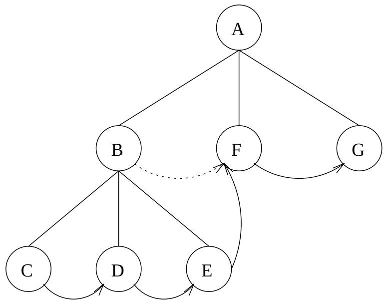

Figure 4: A piece of an SLD tree showing redo continuations

to set up a special stack frame. Of course, in the first disjunct we can use values in registers rather than having to load them in from the stack.

The next issue to consider is how to manage failure continuations.

The first thing that must happen in the generated code for each disjunct is a call to the modframe macro to set the redoip to point to where forward execution should resume on failure.

The easiest way to see how we handle failure continuations for disjunctions is to consider the SLD tree corresponding to the part of the computation around a disjunction. In figure 4 which shows such tree, the places where forwards execution is resumed after failure are represented by the arrows. The figure shows the SLD tree for some computation containing the following disjunction for the goal B in which W is initially ground, and X free:

```
% Goal B
(
    % Goal C
    W = 4,
    X = 1
;
    % Goal D
    p(Y), % det
    X = Y
;
    % Goal E
    q(X), % nondet
    X > 3
)
```

Each time we enter a nondeterministic disjunction, we need to capture the failure continuation so that each disjunct gets tried. Thus the code emitted for the goal B will start with an invocation of the modframe() macro.

For each disjunct except the last, we allocate a new label for the start of the next disjunct, generate a call to modframe() to update the redoip to point to the new label and push the label onto the failure continuation stack. We then generate the code for the disjunct followed by a branch to the end of the disjunction and the label for the start of the next disjunct. At this point we pop the label off the failure continuation stack.

If the code generator encounters a call to a nondeterministic procedure in the goal for the disjunct, the top of the failure continuation stack is replaced with "unknown" because the call may leave frames on the nondeterministic stack and on failure execution should resume at the redoip of the top nondeterministic stack frame rather than at the next disjunct. In the example, this occurs in the goal E when we call q which is nondeterministic. The call to p in the goal D does not require any modification to the failure continuation stack because p is deterministic and will not alter the nondeterministic stack.

This process is repeated until we reach the last disjunct. At this point we set the redoip to point to the start of the next disjunct from the enclosing disjunction if there is one, or if there is not, then it is set to do_fail. The appropriate address is found on the top of the failure continuation stack.

In the following code which generated for the example, W is initially in r1 and framevar(0), and X is stored in r1 at the end.

```
    { modframe(LABEL(mercury__a_2_0_i2)); }
        /* Set failure continuation */
/* checking that W has functor 4 */
    { if (((int) r1 != 4))
        GOTO_LABEL(mercury__a_2_0_i2); }
        /* test for failure */
    { r1 = 1; }
        /* Make a constant */
    { GOTO_LABEL(mercury__a_2_0_i1); }
        /* Jump to end of disj */
mercury__a_2_0_i2:
    ;
        /* Start of next disjunct */
    { modframe(LABEL(mercury__a_2_0_i3)); }
        /* Set failure continuation */
    { call(mercury__p_1_0_l,
        LABEL(mercury__a_2_0_i4)); }
        /* branch to det procedure */
mercury__a_2_0_i4:
    ;
        /* Continuation label */
    { GOTO_LABEL(mercury__a_2_0_i1); }
        /* Jump to end of disj */
mercury__a_2_0_i3:
    ;
        /* Start of next disjunct */
    { modframe(ENTRY(mercury__a_2_0_i7)); }
        /* Restore failure continuation */
    { call(mercury__q_1_0_l,
        LABEL(mercury__a_2_0_i5)); }
        /* branch to nondet procedure */
mercury__a_2_0_i5:
    ;
        /* Continuation label */
    { r2 = 3; }
        /* Make a constant */
    { if (((int) r1 <= (int) r2)) redo(); }
        /* test for failure */
mercury__a_2_0_i1:
    ;
        /* End of disj */
```

As an example of how nested disjunctions are generated, consider the nondeterministic mode of tree_member/2 in which each element stored in the tree is returned nondeterministically. Here we restate the HLDS code for this mode of tree__member:

```
tree__member(H_1, H_2) :-
    (
        H_1 = tree(E, _L, _R),
        H_2 = E
    ;
        H_1 = tree(_E0, L, R),
        (
            tree__member(L, V_E_1),
            H_2 = V_E_1
        ;
            tree__member(R, V_E_1),
            H_2 = V_E_1
        )
    ).
```

The code that is produced for the disjunctions is shown below, with the code for the goals omitted for brevity. Initially H_1 is in both framevar(0) and r1.

```
    {modframe(LABEL(mercury__tree__member_2_1_i2));}
        /* Set failure continuation */
/* checking that H_1 has functor 'tree'/3 */
    { if ((tag((int) r1) != mktag(1)))
        GOTO_LABEL(mercury__tree__member_2_1_i2); }
        /* test for failure */
    ... /* Code to deconstruct H_1 */
    { GOTO_LABEL(mercury__tree__member_2_1_i1); }
        /* Jump to end of disj */
mercury__tree__member_2_1_i2:
    ;
        /* Start of next disjunct */
    { modframe(ENTRY(do_fail)); }
        /* Restore failure continuation */
/* checking that H_1 has functor 'tree'/3 */
    { if ((tag((int) framevar(0)) != mktag(1)))
        fail(); }
        /* test for failure */
    ... /* Code to deconstruct H_1 */
    {modframe(LABEL(mercury__tree__member_2_1_i4));}
        /* Set failure continuation */
    ... /* Code to call tree__member(L, V_E_1) */
    { GOTO_LABEL(mercury__tree__member_2_1_i3); }
        /* Jump to end of disj */
mercury__tree__member_2_1_i4:
    ;
        /* Start of next disjunct */
    { modframe(ENTRY(do_fail)); }
        /* Restore failure continuation */
    ... /* Code to call tree__member(R, V_E_1) */
mercury__tree__member_2_1_i3:
    ;
        /* End of disj */
mercury__tree__member_2_1_i1:
    ;
        /* End of disj */
```

At the start of the second disjunct, we discover that if H_1 is not a tree/3 term, then the disjunction will fail, since the inner disjunction is conjoined with the deconstruction of H_1. Since the continuation at this point is do_fail, if the tag test fails then we call the fail() macro and this stack frame is popped, and forward execution resumes at the code pointed to by the redoip of the next stack frame.

At the end of each disjunct the cache of expressions maintained in the code_info structure is flushed because the disjuncts may cause different parts of the cache to be flushed, and inconsistencies would result.

When the cache is flushed, live values are positioned in the registers determined by the store map. The predicates that do this are used to generate the subgoals of all branched structures, since at the end of any branched structure all the live variables need to be in consistent places.

The process of flushing the cache and storing values in appropriate fixed locations is interleaved, so that the advantage of lazy code generation is not lost: values get generated into the right location, rather than getting generated into any free location and needing register shuffling to move them into the correct one.

In places where the cache is flushed at the end of each branch of a branched goal, some code expansion can occur. If an expression for a variable was cached before the start of the branched goal, and is live at the end of the branched goal, code for it will be generated in each branch. This code expansion is not usually significant, but pathological cases can be designed in which the code expansion is bad.

For example, consider the goal:

```
X is Y + Z,
(
    (
        W = X
    ;
        W = X
    )
;
    (
        W = X
    ;
        W = X
    )
)
```

In this goal, the code to perform the addition is generated once if eager code generation is used, and four times if lazy code generation is used. However, situations like this are very unusual, so the overall increase in code size is not significant.

The simplest semideterministic disjunction is the empty disjunction, which corresponds to the goal `fail`. When an empty disjunction is encountered, the code generator emits code to perform failure. As previously mentioned, this code depends on the local context.

Semideterministic disjunctions with two or more goals consist of code to attempt each disjunct until one succeeds, and then to branch to the end of the disjunction. If none of the disjuncts succeeds, then failure results.

The simplest approach to generating code for a semideterministic disjunction is to treat it as a nondeterministic disjunction and place a compiler generated commit after it. The way we do this is to capture the failure continuation just before the goal, and substitute a new label as the continuation. Before the disjunction, we save the redoip and nondeterministic stack pointer onto the stack and if the disjunction succeeds then we replace the failure continuation with the saved one, restore the nondeterministic frame pointer, and continue execution. If the disjunction fails, then execution will branch to the new failure continuation which restores the saved failure continuation then fails.

The alternative implementation of semideterministic disjunctions is to treat them as nested if-then-else-if constructs and use the techniques used for generating code for if-then-else. This is the approach used by the current implementation.

One of the counterintuitive aspects of the handling of backtracking and the failure continuation is that a disjunction does not require the creation of a new frame on the nondeterministic stack. From the discussion above about where values are saved onto the stack it should be apparent that if the nondeterministic disjunction occurs in a semideterministic procedure, then there will be a commit across it. The failure continuation will be restored after the first success of the disjunction, and any additional nondeterministic stack frames that the nondeterministic goal may have created are popped off the stack. Therefore in this case, we leave the nondeterministic stack as we found it, so there is no need to create a separate nondeterministic stack frame. Another possibility is that the disjunction occurs in a nondeterministic procedure, in which case the procedure prologue will have already created a new frame on the stack, so we do not need to do so. The other case that can occur is where a disjunction is inside another disjunction. In this case the code generator uses the failure continuation stack and emits modframe macros to manipulate the redoip rather than creating extra frames on the nondeterministic stack (In some sense, this is effectively moving the stack from runtime to compile-time).

In the development of the compiler, we deferred the generation of nondeterministic code because we anticipated that it would be difficult, but as should be apparent, once the machinery for handling deterministic and semideterministic code is in place, it creates few extra complications.

4.11.8 Switches

As mentioned previously in section 4.5, some disjunctions are more deterministic than others — some are comprised of disjuncts that we can determine to be mutually exclusive. These disjunctions are converted to switches.

Switches are either locally deterministic or locally semideterministic. The former means that they have a case for every functor of the type of the variable on which the switch is taking place, the latter that cases exist for only some of the functors. The local determinism is stored with the switch, and is independent of the over-all determinism of the switch and of its cases.

The simplest way to generate code for switches is to treat the switch as a nested if-then-else-if construct in which the condition of each case testing the functor of the variable on which the switch is being taken, and have the then case execute the body of the corresponding case of the switch.

To test the functor in a switch, we use the same approach as we used to test the functor in a deconstruction unification — the code generator even calls the same predicate to do so: unify_gen__generate_tag_test. This predicate generates code appropriate to the type of the variable in which the switch is being taken: integer comparisons, string comparisons, and floating point comparisons, as well as tests of primary and secondary tags. If the test succeeds, then control drops through and executes the case; if the test fails, then control branches to the code to perform the next test.

For locally deterministic switches, the functor test in the last case can be omitted because we know what the functor must be. For locally semideterministic switches, the code for the last functor test should generate failure if the match fails.

Quite often, a much better way exists to handle switches. If the program is switching on the value of a string, then it makes sense to use some sort of hashing to drastically improve the performance of the switch. Ideally, we would compute a perfect hash function as described in [3], however, the current implementation uses a simple hash function and resolves conflicts using linear probing. The entries of the hash table contain the string (so that conflicts are resolved correctly), and the address of the code for the case.

Another situation in which we can improve the performance of switches is for switches on a variable of integer or enumeration type. If all the values for which cases exist lie within a restricted range, and within that range, enough values have corresponding cases, then we can use a jump table.

For example, if we have the following switch in a program:

```
(
    X = 1,
    ...
;
    X = 3,
    ...
;
    X = 7,
    ...
)
```

we generate the following jump table:

```
static Code *jump_table[] =
    {
        LABEL(case_1),
        LABEL(no_match),
        LABEL(case_3),
        LABEL(no_match),
        LABEL(no_match),
        LABEL(no_match),
        LABEL(case_7)
```

```
    };
```

and the following code for the switch (assuming X is in r1):

```
    if (r1 < 1 || r1 > 7)
        GOTO(no_match);
    GOTO(jump_table[r1 - 1]);
case_1:
    ...
    GOTO(end_switch);
case_3:
    ...
    GOTO(end_switch);
case_7:
    ...
    GOTO(end_switch);
no_match:
    redo();
end_switch:
```

When generating jump tables, the question arises, how many cases does the jump table need to handle to be faster than the simple if-then-else implementation? The jump table requires indexing into an array, and could be quite costly: if the switch does not get executed frequently, the jump table lookup is likely to cause an extra cache-miss, whereas an if-then-else has no such penalty, even though it may have to execute several extra instructions including branches. The crossover point appears (on the current generation of RISC machines) to be about three cases. The overhead of the jump table lookup is estimated to cost about the same as testing whether a case matches twice, so for two cases the cost is about equal, and the code generator errs on the side of caution, and uses the conservative approach of an if-then-else structure.

4.11.9 If-then-else

If-then-else is the most complicated of our compound constructs. The reason is that it does not have as regular structure as the other compound structures.

A deterministic if-then-else has a semideterministic condition, and deterministic then and else cases. Since there is no failure continuation in deterministic code, to generate a deterministic if-then-else, we first push a failure continuation. Then we generate code for the condition and pop off the failure continuation. The code for the condition is immediately follow by the then case, a branch to the end of the if-then-else, the failure continuation label, and the else case.

The general form of an if-then-else is:

```
    (
        Condition
    ->
        Then
    ;
        Else
    )
```

for which we generate code of the following form:

```
    Condition, % branch to else_case on failure
    Then
    GOTO(end_if_then_else);
else_case:
    Else
end_if_then_else:
```

This structure forms a pattern on which the generation of semideterministic and nondeterministic if-then-elses is based.

For an if-then-else to be semideterministic, at least one of the then and else goals must be semideterministic and none of the condition, then or else cases nondeterministic.

Semideterministic if-then-else constructs are handled in much the same way as deterministic ones. Because we need to capture failure, we push a new failure continuation onto the failure continuation stack immediately before generating code for the condition, and pop it off again as soon as we have generated the condition (before we generate the then or else cases).

There are two kinds of nondeterministic if-then-else. In the first, the condition is semideterministic, and at least one of the then and else goals is nondeterministic. These cases are handled exactly as we handled semideterministic ones. In the second kind, the condition itself is nondeterministic. As noted in section 3.5, we allow backtracking to make computation reenter the condition. To implement this correctly, we generate a call to the modframe macro to capture failure before entering the condition, and push the corresponding label onto the failure continuation stack. This way, if the condition fails without producing any solutions, computation can be made to proceed to the else goal. If the condition produced at least one solution, we want to prune away the possibility of computation entering the else goal, so we pop the continuation stack to remove the captured failure continuation, and generate a call to modframe to restore the redoip to what it was before we reached the condition. In the else case, we want to restore the redoip as well, however in this case we know that the current stack frame is on top of the stack, because failure brought us to the else case.

For example, consider the procedure in which X is output:

```
p(X) :-
    (
        q(W)
    ->
        X = W
    ;
        r(X)
    ).
```

where we assume q/1 to output a binding of W nondeterministically.

The code for the if-then-else (not the whole procedure) is:

```
    { modframe(LABEL(mercury__p_1_0_i1)); }
        /* Set failure continuation */
    { call(mercury__q_1_0_l,
```

```
        LABEL(mercury__p_1_0_i2)); }
        /* branch to nondet procedure */
mercury__p_1_0_i2:
    ;
        /* Continuation label */
    { modframe(ENTRY(do_fail)); }
        /* Restore failure continuation */
    { GOTO_LABEL(mercury__p_1_0_i3); }
        /* Jump to the end of if-then-else */
mercury__p_1_0_i1:
    ;
        /* else case */
    { modframe(ENTRY(do_fail)); }
        /* Restore failure continuation */
    { r1 = 1; }
        /* Make a constant */
mercury__p_1_0_i3:
    ;
        /* end of if-then-else */
```

All in all, the implementation using the failure continuation stack seems to simplify this seemingly quite complex situation (enabling backtracking to reenter the condition), so that it becomes quite manageable.

4.11.10 Negation

For mode correct programs, negation is always semideterministic because even if the negated goal is nondeterministic, the negation may not have any output bindings, and so has semideterministic behaviour.

In fact, negation reduces to an easy form of an if-then-else (although we generate the code directly, rather than invoking procedures for generating if-then-else structures).

Once again, we capture the failure continuation by pushing a new failure continuation onto the stack before generating the negated goal, and then popping it off again afterwards.

For example, consider the following goal in which q/1 is semideterministic.

```
    not q(X)
```

The code generated for this goal is:

```
    { call(mercury__q_1_0_1,
        LABEL(mercury__p_1_0_i3)); }
        /* branch to semidet procedure */
mercury__p_1_0_i3:
    ;
        /* Continuation label */
    { if ((int) r1) GOTO_LABEL(mercury__p_1_0_i4); }
        /* Test for success */
    { GOTO_LABEL(mercury__p_1_0_i2); }
        /* fail */
mercury__p_1_0_i4:
    ;
    { GOTO_LABEL(mercury__p_1_0_i1); }
        /* fail */
mercury__p_1_0_i2:
    ;
```

where mercury__p_1_0_i1 is the failure continuation of the current context. (The somewhat convoluted code generated for this example is improved considerably by peephole optimisation.)

In this way, if the the call to p succeeds, execution branches to the failure continuation, and if the call fails, execution drops through to mercury__p_1_0_i2.

In a nondeterministic context, it is possible that instead of branching to a known label, either the redo or fail macros are called.

4.11.11 Quantification

The only HLDS goal type that we have not discussed so far is the existential quantifier 'some'.

Quantification is mostly a structuring tool that aids analysis. It is used to explicitly restrict the scope of variables — usually to give hints about determinism to the compiler. Since all the determinism annotations are computed by the determinism analysis pass before code generation, the information produced by quantification has already been utilised, and nothing more needs to be done. Therefore, goals of the form:

```
    some Vars Goal
```

simply reduce to

```
    Goal
```

and the code generator simply ignores the quantifier — generating code only for the subgoal.

4.11.12 Heap reclamation on failure

When a goal fails, none of the terms created during its execution can be referenced again. Therefore we can reclaim the memory used by them.

This is done by saving the heap pointer (hp) before the goal, and in the failure continuation, restoring it. We do not want to save the heap pointer before every semideterministic or nondeterministic goal because that would be inefficient. Instead, we save it only before disjunctions, negations and the conditions of if-then-elses, and then only if the construct may allocate heap.

In disjunctions, the heap pointer is saved before the disjunction and restored at the start of each disjunct. In if-then-elses it is restored only in the else case.

For example, consider the following predicate:

```
q(X, Y) :-
    (
        p(X)
    ->
        Y = yes
    ;
        Y = no
    )
```

It is possible that the semideterministic predicate p/0 may allocate heap so we save and restore the heap pointer.

The code generated for the if-then-else is:

```
{ detstackvar(1) = (int) hp; }
/* Save value on the stack */
{ r2 = (int) r1; }
/* Assigning from X */
{ localcall(mercury__p_1_0_1,
LABEL(mercury__q_2_0_i2)); }
/* branch to semidet procedure */
mercury__q_2_0_i2:
;
/* Continuation label */
{ if ((int) r1) GOTO_LABEL(mercury__q_2_0_i3); }
/* Test for success */
{ GOTO_LABEL(mercury__q_2_0_i1); }
/* fail */
mercury__q_2_0_i3:
;
{ r2 = 0; }
/* Assigning from H_2 */
{ GOTO_LABEL(mercury__q_2_0_i4); }
/* Jump to the end of if-then-else */
mercury__q_2_0_i1:
;
/* else case */
{ LVALUE_CAST(Word,hp) = (int) detstackvar(1); }
/* Restore value from stack */
{ r2 = 1; }
/* Assigning from H_2 */
mercury__q_2_0_i4:
;
/* end of if-then-else */
```

4.11.13 Procedure prologues and epilogues

To generate the prologue and epilogue, the code generator must determine the number of stack locations used and whether the succip (success instruction pointer) must be saved. The number of stack locations required by the procedure is not completely determined by the storage allocation pass — the code generator may need to save the succip across procedure calls and the heap pointer across semideterministic and nondeterministic goals that may allocate heap space. The latter may happen in nested contexts, and so simply allocating a single stack location is insufficient. For these reasons, the procedure prologue and epilogue are generated after the main body of the predicate.

The other situation where storage may need to be allocated during code generation is when the compiler discovers that it can cut away some choice points after a call to some nondeterministic code with no output variables. In this kind of situation there are three possibilities. If the nondeterministic code contains a nondeterministic call, then the nondeterministic stack pointer must be saved to cut away any frames that the call may have created. If the nondeterministic code contains a disjunction then the redoip needs to be saved so that it can be restored after the first success of the nondeterministic code. If both these conditions occur, then both must be saved. The current implementation always saves both for simplicity.

The details of the procedure prologue and epilogue are dependent on the determinism of the procedure. Deterministic and semideterministic procedures both use the deterministic stack to store values across procedure calls. As discussed in section 3.2, the deterministic stack does not need a fixed frame format, though in practice we use one for simplicity. The succip (if saved) occupies the deepest stack slot, and any stack slots allocated on the fly by the code generator are placed just above. All the slots allocated for ordinary variables by the storage allocation pass are allocated topmost on the stack.

Two reasons exist to explain this allocation: if all offsets are determined relative to the stack top, it is much more convenient to place later allocations deeper on the stack so that the offsets of other stack references do not change. The other reason is that the special purpose stack slots for saving the succip and hp will have fewer and less regular accesses. By localising the ordinary usages of the stack on the top, we can hope for marginally better locality of reference and hence better cache performance. However, without an alternate implementation, it is not possible to quantitatively evaluate the impact of the ordering of stack slots on performance.

The procedure prologue for a deterministic and semideterministic procedures consists of a labeled entry point, code to increment the stack pointer (if any stack slots are used), and possibly code to save the succip.

As an example, here is the procedure prologue that is generated for tree__member/2 in its semideterministic mode:

```
tree__member_2_0:
    ;
        /* Procedure entry point */
    incr_sp(8);
        /* Allocate stack frame */
    { detstackvar(8) = (int) succip; }
        /* save the success ip */
/* End of procedure prologue */
```

Nondeterministic code uses the nondeterministic stack rather than the deterministic stack and has stack frames of a fixed format. The nondeterministic stack requires dynamic links since the procedure deallocating a frame is not necessarily the same as the one that created it. In the procedure prologue for nondeterministic procedures the entry point is the same as it is for deterministic and semideterministic procedures. In addition, the nondeterministic procedure prologue must create a frame on the nondeterministic stack, and set the dynamic link. At the same time, the failure continuation is set to do_fail, and the predicate name is given. The predicate name is used for debugging purposes, and for the optimised versions of programs, the assignment to store it in the nondeterministic frame is eliminated, saving a marginal amount of both time and space.

Here is the procedure prologue for tree__member/2 in its nondeterministic mode:

```
mercury__tree__member_2_1:
    ;
        /* Procedure entry point */
    { mkframe("tree__member/2",4,ENTRY(do_fail)); }
        /* Nondet stackframe */
/* End of procedure prologue */
```

The procedure epilogue for deterministic procedures first positions the output arguments of the procedure, restores the succip (if it was saved), if necessary, decrements the stack pointer and then calls the proceed() macro which causes a jump to the succip.

For semideterministic procedures, two epilogues are generated — one for success and one for failure. The success epilogue positions the output arguments in the appropriate registers, restores the succip, decrements the stack pointer, and sets register one to TRUE before calling the proceed() macro. The failure epilogue is the same, except that it doesn't need to position the output arguments, and it sets register one to FALSE before calling proceed().

Here is the procedure epilogue for tree__member/2 in its semideterministic mode:

```
/* Start of procedure epilogue */
    { LVALUE_CAST(Word,succip) =
            (int) detstackvar(8); }
        /* restore the success ip */
    decr_sp(8);
        /* Deallocate stack frame */
    { r1 = TRUE; }
        /* Succeed */
    { proceed(); }
        /* Return from procedure call */
mercury__tree__member_2_0_i1:
    ;
        /* FallThrough */
    { LVALUE_CAST(Word,succip) =
            (int) detstackvar(8); }
        /* restore the success ip */
    decr_sp(8);
        /* Deallocate stack frame */
    { r1 = FALSE; }
        /* Fail */
    { proceed(); }
        /* Return from procedure call */
/* End of procedure epilogue */
```

Because a nondeterministic procedure may be backtracked into, and its frame may not even be on top of the nondeterministic stack when its epilogue is reached, its epilogue does not deallocate the frame, and simply calls succeed() after it has flushed the expression cache and positioned the output arguments. The frame will be deallocated at some future point in the computation when failure causes backtracking past the procedure.

4.12 Peephole optimisation

No matter how careful the code generator is about producing "optimal" code sequences, it is inevitable that the list of LLDS instructions will contain some redundancy. Some of this redundancy is recognized and removed by peephole optimisation. Additionally, we use the technique of value numbering described in [1] to further optimise the list of LLDS instructions.

One of the reasons that it is important that we optimise the list of LLDS instructions as much as possible is that although the C compiler can be expected to perform such tasks as instruction scheduling for pipelined architectures, it is generally not possible for it to eliminate redundant assignments. When generating ANSI C, peephole optimisations are particularly important because the flow of control has been obscured in a way that will not allow the C compiler to eliminate branches to branches.

The peephole optimiser performs traversals of the list of LLDS instructions, looking for patterns that it can optimise. When it gets a match, it replaces the instructions with those it knows to be more efficient. If no matches succeed, then the peephole optimisation pass fails. One peephole optimisation can often lead to code that will match another peephole optimisation, so the peephole optimiser is called multiple times until no matches occur. There is a compiler flag that specifies the maximum number of times that peephole optimisation should be iterated.

Currently, the peephole optimiser performs the following optimisations:

- Replacing a jump to a jump with a jump to the destination of the second jump.
- Transforming calls immediately followed by a frame deallocation and a proceed to tailcalls.
- Removing branches where the destination is a label that immediately follows the branch. This also applies to conditional branches.
- Eliminating non-exported labels that are not branched to.
- Where a mkframe macro is followed by non-branching code and a modframe macro, placing the target from the modframe in the mkframe and eliminating the modframe.
- Swapping mkframe and conditional fail/redo instructions to avoid the mkframe if the test fails.

These optimisations are not the only ones possible — we will add more as we find patterns in the generated code for which we can see a better alternative.

An important LLDS instruction that does not produce any code directly is for annotating the list of LLDS instructions with information about which registers and stack locations are live at a particular point in the program. This is used by value numbering, which is not fully implemented yet. None the less, the code generator retains the appropriate information and outputs the live-value instructions before calls and procedure returns.

4.13 Program assembly

Once the LLDS has been processed until the compiler can do no more to improve it, it is written out into a "mod" file (so called because it is named with a suffix of

.mod). The mod file is then processed with a perl script "mod2c" which turns it into a C file which can then be compiled and linked. The perl script has an option telling it whether or not to exploit GNU C features.

The resulting C code is then compiled to an object file. The object files for the modules used by the program must be linked together with the runtime system and the module initialiser to generate an executable file. The module initializer is a C function which is called on startup to build a table of exported labels to allow them to be accessed from other modules. The initializer is generated by a shell script and is compiled with the C compiler. The initializer function for the poly benchmark is

```
void init_modules(void)
{
    mercury__poly_module();
    special_labels_module();
    io_module();
    interpreter_module();
}
```

The mercury__poly_module enters the labels for the poly module. This function contains the following code to export the entry points for the exported procedures:

```
entry_mercury__var__eq___2_0 =
                &&mercury__var__eq___2_0;
entry_mercury__term__eq___2_0 =
                &&mercury__term__eq___2_0;
entry_mercury__poly__eq___2_0 =
                &&mercury__poly__eq___2_0;
entry_mercury__main3_3_0 =
                &&mercury__main3_3_0;
entry_mercury__main1_1_0 =
                &&mercury__main1_1_0;
```

The process of compiling Mercury source files to mod files, converting mod files to C, creating the initializer files, and compiling them and linking with the Mercury library is currently managed using cake [14], though eventually it will be done by the compiler itself.

5 Performance results

In this section we compare the performance of Mercury programs with the performance of programs written in other logic programming languages. We also examine the performance implications of lazy code generation.

5.1 Background

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

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

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

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

fast using both non-local gotos and global register variables

System		arith	geom	harm
SWI	none	1.00	1.00	1.00
NU	none	1.84	1.77	1.72
	decls	4.92	2.17	1.67
SICStus	compact	2.09	2.05	2.02
	compact decls	4.10	2.26	1.88
	fast	7.45	7.12	6.83
	fast decls	5.77	3.70	3.15
Aquarius	none	12.76	11.23	9.87
	decls	16.09	13.80	12.26
	analysis	24.79	21.49	18.42
	analysis decls	28.92	22.96	19.00
Mercury	ansi	13.08	11.62	10.54
	jump	20.25	18.15	16.66
	fast	40.36	35.25	32.22

Table 1: Averages of benchmark speed ratios

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

5.2 Speed tests

We have tested the speed of the Mercury implementation on a set of standard Prolog benchmarks which we have translated to Mercury. Besides Mercury, we ran the benchmarks on four other logic programming languages: SWI-Prolog 1.9.0 [22], NU-Prolog 1.6.4 [20], SICStus Prolog 2.1 [2] and Aquarius Prolog 1.0 [21]. These versions are the latest we have access to. We included Aquarius in our benchmarks because of its reputation as the fastest generally available Prolog system. We included SICStus because of its wide user base. We included NU-Prolog because of its use in the Mercury implementation and because of its pioneering implementation of coroutining. We included SWI-Prolog because its portability and free availability should enable every reader to calibrate our results. We would have liked to run the benchmarks on some other systems as well, Quintus Prolog and PARMA [19] in particular, but we do not have access to those.

For all the Mercury benchmarks we report on, the code we tested came straight out of the compiler; we did not modify any of them in any way.

SWI-Prolog and NU-Prolog are bytecode interpreters. The compact option of SICStus is also a bytecode interpreter, while its fastcode option is a native code compiler (the fastcode we tested is not the one described in [6], but based on the results reported in that paper, our results would not be much different if it were). Aquarius is also a native code compiler. Aquarius has an option asking the compiler to analyze the program; we tested Aquarius both with and without this option. With the exception of SWI-Prolog, all these Prolog systems allow programmers to supply declarations, but none require it. We therefore tested each such system both with and without declarations.

NU-Prolog's when declarations and SICStus Prolog's block declarations specify that a predicate should be called only when certain variables have been bound. 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 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.

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

Table 1 contains summaries of our timing results, using the arithmetic, geometric and harmonic means respectively. These numbers are derived from table 2, which shows the speed of each variant of each system on each benchmark. All speeds in this table are normalized to the speed of SWI-Prolog, a publically available system. Table 3 contains the same data normalized to the speed of the fastest system on each given benchmark; the speed of this system is shown as 100%. The timing data from which this is derived was 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 [11]. 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. Cqueen, crypt, queen and query are mostly nondeterministic while deriv is mostly semideterministic; the other five benchmarks are mostly deterministic.

Table 1 shows Mercury to be the fastest system overall with all three averaging methods. Using the harmonic mean, the fast option of Mercury is 70% faster than the fastest variant of Aquarius, and it outperforms SICStus fastcode by a factor of 4.7; it outperforms all the other systems we measured[1] by factors ranging from 16 to 32. Using the geometric mean, Mercury is 53% faster than Aquarius, while using the arithmetic mean, Mercury is 40% faster. For averaging rates, the harmonic mean is the most appropriate of the three averaging methods, since it is the one least influenced by a single good result. For example, table 2 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 decla-

[1] Andrew Taylor, the author of PARMA [19], has seen the C code generated by the Mercury compiler and the assembly code generated from that by gcc -O2. He says that for the examples he looked at, Mercury usually generates better code than PARMA.

System		cqueen	crypt	deriv	nrev	poly	prime	qsort	queen	query	tak
SWI	none	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
NU	none	1.5	1.4	1.5	3.4	1.8	1.6	1.6	1.6	1.9	2.2
	decls	1.6	1.2	0.9	3.4	1.6	1.7	1.2	33.8	1.9	1.9
SICStus	compact	2.1	2.1	1.6	3.0	2.2	1.7	1.8	2.1	1.8	2.5
	compact decls	1.9	1.8	1.3	1.9	1.9	1.5	1.4	25.3	1.7	2.2
	fast	6.1	6.0	6.8	12.1	7.9	5.7	6.9	6.8	4.4	11.6
	fast decls	3.5	3.1	2.4	2.2	3.2	2.7	2.2	30.8	3.9	3.8
Aquarius	none	16.2	7.3	7.2	15.7	9.8	8.1	14.7	15.9	4.5	28.0
	decls	17.5	7.4	7.9	19.4	12.6	9.0	16.7	17.4	8.6	44.3
	analysis	25.9	7.5	15.7	36.9	18.8	11.7	32.6	26.1	18.2	54.5
	analysis decls	25.9	7.6	15.8	36.8	18.9	12.9	32.8	26.1	18.2	94.2
Mercury	ansi	10.8	12.3	5.6	7.4	12.1	10.2	10.0	9.6	23.2	29.5
	jump	20.9	16.1	10.1	11.0	20.2	15.0	14.4	16.3	29.4	49.0
	fast	39.1	24.5	19.4	29.9	39.7	24.3	41.3	28.8	39.6	116.8

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

System		cqueen	crypt	deriv	nrev	poly	prime	qsort	queen	query	tak
SWI	none	2.6	4.1	5.2	2.7	2.5	4.1	2.4	3.0	2.5	0.9
NU	none	3.7	5.6	7.7	9.3	4.4	6.7	3.8	4.8	4.8	1.9
	decls	4.2	4.9	4.5	9.3	4.0	6.8	3.0	100.0	4.7	1.7
SICStus	compact	5.3	8.5	8.0	8.2	5.6	7.0	4.4	6.1	4.5	2.2
	compact decls	4.9	7.4	6.9	5.2	4.8	6.3	3.3	75.0	4.3	1.9
	fast	15.5	24.6	35.2	32.8	20.0	23.6	16.8	20.3	11.1	9.9
	fast decls	8.9	12.7	12.5	5.9	8.0	11.0	5.4	91.1	9.7	3.2
Aquarius	none	41.5	29.7	37.2	42.7	24.8	33.4	35.7	47.0	11.4	24.0
	decls	44.6	30.3	41.1	52.7	31.6	36.9	40.5	51.5	21.8	38.0
	analysis	66.2	30.7	81.3	100.0	47.4	48.2	78.8	77.3	45.8	46.7
	analysis decls	66.2	30.9	81.4	100.0	47.6	52.9	79.4	77.4	45.8	80.7
Mercury	ansi	27.6	50.1	29.0	20.1	30.5	42.1	24.1	28.5	58.5	25.3
	jump	53.4	65.7	52.4	29.9	51.0	61.7	34.9	48.3	74.1	41.9
	fast	100.0	100.0	100.0	81.2	100.0	100.0	100.0	85.2	100.0	100.0

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

rations on that one benchmark.

Although we expect the fast version of Mercury to be available on almost all machines, even the ansi version of Mercury is more than 50% faster than SICStus fastcode, which, like Aquarius, runs on only a few machines. Given that the only optimizations implemented by the Mercury compiler so far are some forms of indexing and some local peephole optimizations, these results reflect the inherent efficiency of the execution algorithm as it is currently implemented in the compiler. We are therefore quite confident that Mercury will perform just as well on large programs.

Mercury is the fastest system on eight out of the ten benchmarks, and it is within 20% the speed of the fastest system on the other two. Aquarius with analysis and declarations is the next fastest system. Even Aquarius is significantly slower than Mercury on the two largest benchmarks, poly and crypt. Aquarius beats Mercury on only one small benchmark, nrev. The reason for this is that Mercury does not yet implement any recursion optimizations. The Mercury peephole optimizer does perform last call optimization (and therefore tail recursion optimization) if the last call happens to leave the values of all the outputs of the predicate in the right registers. Such opportunities do in fact occur in prime, qsort and tak. They are surprisingly frequent in real programs as well due to the popularity of the accumulator-passing style of programming.

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

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

Lazy Code	Number of Registers				
	1	2	4	8	16
yes	29	51	77	98	100
no	27	48	74	97	99

Table 4: Percentage of register accesses captured by physical registers

problem with Mercury, since our results show that the Mercury compiler puts declarations to good use.

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

5.3 Lazy code generation

Our decision to use lazy code generation was based on the desire to make a reasonable job of register allocation. Although the virtual machine presents arbitrarily many virtual registers, it is important to localise register access to as few registers as possible, since only a small number of virtual registers can be mapped onto real machine registers.

Table 4 shows the percentage of register accesses captured by various numbers of real machine registers. That is, with lazy code generation enabled, 51% of all register accesses are to two registers, and 98% are to 8 registers.

The data summarised in this table were collected by switching on a C compiler option which causes every virtual machine register access to be counted. The register counts were normalised and averaged over the set of benchmarks.

From this table we can see that the use of lazy code generation results in a more compact access pattern. Even though the improvement is small, this can have a significant impact on performance since an access to a virtual machine register that is mapped to a global variable in memory is several times as expensive as an access to one that is mapped to a real machine register.

The compactness of the register access pattern will be particularly important when we port the Mercury compiler to Linux running on Intel x86 hardware. On such machines, there are only three registers readily available, so the compactness of register accesses is at a premium.

It is possible to improve the register access pattern further by changing the predicate call convention so that the N input arguments are in registers 1 to N, and the M output arguments are in registers 1 to M. This would require only small modifications to the code generator, and we intend to do this in the near future.

There is another benefit of lazy code generation which this data does not show. With lazy code generation switched on, term constants are constructed in a way that enables the terms to be allocated as static structures, rather than needing to be allocated dynamically and constructed at run time. The use of lazy code generation enables these static structures to be used because of the way that goals are reordered. Eager code generation does not reorder goals, and the way that term constants are decomposed in the transformation to superhomogeneous form makes it impossible for static terms to be detected without reordering.

5.4 Program size and compile time

The programs generated by the Mercury compiler for these benchmarks are quite small. The biggest benchmark is poly, whose source is about 260 lines of Mercury. Compiled to ANSI C, its executable is 22 Kb; the stripped executable is 14 Kb. Compiled to exploit both GNU C extensions, its executable is 11 Kb, with the stripped executable being 8 Kb.

To give a real example, the type-checking module of the Mercury compiler is about 2600 lines of Mercury. When given the fast option and this 95 Kb file, our Mercury to C compiler generates about 330 Kb of C code, which gcc -O2 -msupersparc then converts to about 90 Kb of object code. Given the high level nature of Mercury code, this is quite competitive with compilers for imperative languages such as C. Although we generate a separate code sequence for each mode of a predicate, this does not cause any problems with code size for two reasons. First, very few predicates are used in more than one mode; in the Mercury code we have written so far, about 40 predicates have multiple modes out of more than 2000. Second, the code for each mode can be small because it is specialized; the native code emitted by a conventional Prolog compiler has to be prepared to handle all possible dataflow patterns.

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

6 Software engineering issues

The Mercury compiler is the largest Mercury program written so far. At over 42,000 lines of source, it is quite a large program. As such, its development has provided some insights into the software engineering properties of Mercury.

One of the important properties of Mercury is its ability to support a team of programmers working on the one program. The compiler is being developed by three people, and our experience has been that one of us can make significant changes to one part of the program without adversely impacting on the work of the others.

The module system is partly responsible for this, since it enables us to hide the implementation of a part of the compiler behind an interface, so that internal representations and implementation dependencies are hidden.

This principle is exemplified by the code generator. The main data structure used in the code generator is the code_info type which is an abstract type exported by the code_info module. All the other modules in the code generator manipulate the code_info structure by calling predicates in the code_info module. This means that changes to things such as the format of the failure continuation stack (for example) requires almost no changes outside the code_info module.

The analysis passes have proven to be extremely effective at detecting programmer errors. The type system catches quite a large proportion of errors. The smaller number of errors found by mode analysis tend to be variable naming errors.

In a large piece of software that is continually evolving, the determinism system is invaluable. Most of the compiler is deterministic (just under 85% of predicate modes are deterministic), and many errors arise from the omission of a clause to handle a new alternative for a type. These cases are caught by the determinism analysis because the inferred determinism of an otherwise deterministic predicate which is missing a clause is semideterministic.

The type, mode and determinism checkers have found many errors in the compiler itself. They are so successful at locating programmer errors, that it is not unusual for us to add large pieces of functionality and have them work correctly as soon as we have removed the errors found by the compiler.

The code generator is the largest single component of the compiler. The eight modules in the code generator total over 6,000 lines of code, code_info having about 2,800. This makes it one of the largest single modules in the compiler (the modules for reading in programs, type checking and mode checking are about the same size).

This raises a software engineering issue that applies not only to Mercury, but to all languages with strong module systems: empirically, a good upper bound to place on the size of a (Mercury) module seems to be about a thousand lines. In modules longer than about a thousand lines of code, it becomes increasingly difficult to keep track of the dependencies and invariants that should be maintained. A problem arises then, when a module becomes large it is often difficult to decide when and how to split a large module into smaller ones.

One of the desirable characteristics of a module is that it have a reasonable small, clean interface to other modules. If care is not taken when dividing up a large module, this characteristic can be lost completely, and no real improvement in the management of the complexity of programs is achieved. The code_info module is quite strongly coupled — components of the code_info structure are interdependent and to try to handle them in separate modules would require making concrete parts of the data structures which should clearly be abstract.

That it is not always possible to decompose programs into modules of an easily manageable size suggests that the module system could benefit from extra features such as hierarchical modules, so that for instance, the code_info module could export its data structures to the other code generator modules, without making them visible to the rest of the compiler.

7 Conclusion

The code generator for the Mercury compiler succeeds in generating efficient code that runs significantly faster than the code produced by the compilers for other logic programming languages. It does this despite being an experimental implementation in which the emphasis has been on simplicity rather than speed.

Although some parts of the compiler such as complicated unification procedures and some aspects of the handling of polymorphism are not yet fully implemented, the compiler handles many programs, for which it generates correct code. We expect to have the compiler completed to a sufficient degree that we can bootstrap it by the end of the year.

As previously noted, the compiler is being developed as a collaborative project. For the purpose of the assessment of this thesis, the following discussion attributes the different parts of the compiler to their author.

The parsing, HLDS construction, type analysis, mode analysis and switch detection passes are almost entirely the work of Fergus Henderson. The initial implementation of the determinism pass was by me, though Fergus has since done a large amount of work to improve the error reporting.

The liveness analysis, follow-code transformation, follow-vars computation and storage allocation passes are authored by me.

The eight code generator modules were designed and implemented by me, though Fergus and I have both modified and debugged parts of it as required. The code_info module is almost solely my work — the part of the code generator which Fergus has worked on has mainly been the implementation of jump tables for switches.

The peephole optimiser has been developed by Zoltan Somogyi and Fergus, and the implementation of value numbering is authored by Zoltan.

The execution model and runtime environment were developed by Zoltan, Fergus, and me.

Many modules of the standard library, which accounts for nearly half of the compiler, were initially authored by me. However, Fergus and I have both added functionality to them as required, so accurate attribution of the modules in the standard library is impossible.

Overall, the compiler has 42,000 lines of source, of which I wrote about one third.

Thanks without bound are due to Fergus and Zoltan for their encouragement and support, and for giving me the privilege of working with them. Two clubs.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley Publishers, 1985.

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

[3] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43:257–264, 1992.

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

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

[14] Z. Somogyi. Cake: a fifth generation version of make. *Australian Unix system User Group Newsletter*, 7(6):22–31, April 1987.

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

[16] 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.

[17] Z. Somogyi, F. Henderson, and T. Conway. Mercury, an efficient purely declarative logic programming language. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, Adelaide, South Australia, February 1995.

[18] Z. Somogyi, F. Henderson, and T. Conway. The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS '94 Postconference Workshop on Implementation Techniques for Logic Programming Languages*, Syracuse, New York, November 1994.

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

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

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

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

A An example module

```
:- module tree.

:- interface.

:- type tree(T).

:- pred tree__member(tree(T), T).
:- mode tree__member(in, in) is semidet.
:- mode tree__member(in, out) is nondet.

:- pred tree__contains(tree(T), tree(T)).
:- mode tree__contains(in, in) is semidet.

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

:- implementation.

:- import_module require. % for error
:- import_module int. % for arithmetic

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

tree__member(tree(E, _L, _R), E).
tree__member(tree(_E0, L, R), E) :-
    (
        tree__member(L, E)
    ;
        tree__member(R, E)
    ).


tree__contains(T1, T2) :-
    not (
some [E] (
tree__member(T1, E),
not tree__member(T2, E)
)
    ).

% This is equivalent to the following code
% which although a simpler logical statement,
% and logically identical is not handled by the
% front end of the compiler at the current time.
%
% tree__contains(T1, T2) :-
%     all [E] (
%         tree__member(T1, E) =>
%             tree__member(T2, E)
%     ).

tree__depth(empty, 0).
tree__depth(tree(_E, L, R), D) :-
    tree__depth(L, LD),
    tree__depth(R, DR),
    int__max(LD, DR, D0),
    D is D0 + 1.
```

B The code for append/3

In section 3 we presented hand compiled code for the two principal modes of append to demonstrate the execution model. Here we present the source for append and the machine generated mod file for the two modes.

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

append([], Bs, Bs).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).
```

The following code for the two modes of append was generated by the compiler with peephole optimisation switched off to make the code clearer.

```
#include "imp.h"

/* this code automatically generated. */

BEGIN_MODULE(mercury__x_module)
    /* no module initialization in
        automatically generated code */
BEGIN_CODE


/*-----------------------------------------------*/
/* code for predicate append/3 in mode 0 */
/* Start of procedure prologue */
mercury__append_3_0_l:
    ;
        /* Procedure entry point */
/*
Stack slot assignments (if any):
A     ->    stackvar(1)
 */
    incr_sp(2);
        /* Allocate stack frame */
    { detstackvar(2) = (int) succip; }
        /* save the success ip */
/* End of procedure prologue */
/* checking that HeadVar__1 has functor '[]'/0 */
    { if (((int) r1 != mkword(mktag(0), mkbody(0))))
        GOTO_LABEL(mercury__append_3_0_i2); }
        /* test for failure */
    { r3 = (int) r2; }
        /* Assigning from HeadVar__3 */
    { GOTO_LABEL(mercury__append_3_0_i1); }
        /* skip to the end of the switch */
mercury__append_3_0_i2:
    ;
        /* next case */
    { detstackvar(1) = (int) field(mktag(1),
        (int) r1, 0); }
        /* Assigning from A */
    { r3 = (int) r1; }
        /* Swap live variable to a new register */
    { r1 = (int) field(mktag(1), (int) r3, 1); }
        /* Assigning from As */
/*
 * Live lvalues at end of block:
 *    r1
```

```
 *     r2
 *     detstackvar(1)
 *     detstackvar(2)
 *     field(mktag(1), r3, 0)
 */
    { localcall(mercury__append_3_0_l,
        LABEL(mercury__append_3_0_i3)); }
        /* branch to det procedure */
mercury__append_3_0_i3:
    ;
        /* Continuation label */
    { r1 = mkword(mktag(1), (int) hp); }
    incr_hp(2);
        /* Allocate heap space & tag the pointer */
    { field(mktag(1), r1, 0) =
        (int) detstackvar(1); }
        /* Assigning from A */
    { field(mktag(1), r1, 1) = (int) r3; }
        /* Assigning from Cs */
    { r3 = (int) r1; }
        /* Assigning from HeadVar__3 */
mercury__append_3_0_i1:
    ;
        /* End of switch */
/* Start of procedure epilogue */
    { LVALUE_CAST(Word,succip) =
        (int) detstackvar(2); }
        /* restore the success ip */
/* End of procedure epilogue */
    decr_sp(2);
        /* Deallocate stack frame */
/*
 * Live lvalues at end of block:
 *     r3
 */
    { proceed(); }
        /* Return from procedure call */

/*-------------------------------------------*/
/* code for predicate append/3 in mode 1 */
/* Start of procedure prologue */
mercury__append_3_1_l:
    ;
        /* Procedure entry point */
/*
Stack slot assignments (if any):
HeadVar__3      ->    framevar(0)
A     ->    framevar(1)
 */
    { mkframe("append/3", 3, ENTRY(do_fail)); }
        /* Nondet stackframe */
/* End of procedure prologue */
    { framevar(0) = (int) r3; }
        /* Assigning from HeadVar__3 */
    { modframe(LABEL(mercury__append_3_1_i2)); }
        /* Set failure continuation */
    { framevar(2) = (int) hp; }
        /* Save value on the stack */
    { r1 = mkword(mktag(0), mkbody(0)); }
        /* Tag a word */
    { r2 = (int) framevar(0); }
        /* Assigning from HeadVar__2 */
    { GOTO_LABEL(mercury__append_3_1_i1); }
        /* Jump to end of disj */
mercury__append_3_1_i2:
    ;
        /* Start of next disjunct */
    { LVALUE_CAST(Word,hp) = (int) framevar(2); }
        /* Restore value from stack */
    { modframe(ENTRY(do_fail)); }
        /* Restore failure continuation */
/* checking that HeadVar__3 has functor '.'/2 */
    { if ((tag((int) framevar(0)) != mktag(1)))
        fail(); }
        /* test for failure */
    { framevar(1) = (int) field(mktag(1),
        (int) framevar(0), 0); }
        /* Assigning from A */
    { r3 = (int) field(mktag(1),
        (int) framevar(0), 1); }
        /* Assigning from Cs */
/*
 * Live lvalues at end of block:
 *     framevar(1)
 *     r3
 *     field(mktag(1), framevar(0), 0)
 */
    { localcall(mercury__append_3_1_l,
        LABEL(mercury__append_3_1_i3)); }
        /* branch to nondet procedure */
mercury__append_3_1_i3:
    ;
        /* Continuation label */
    { r3 = mkword(mktag(1), (int) hp); }
    incr_hp(2);
        /* Allocate heap space & tag the pointer */
    { field(mktag(1), r3, 0) =
        (int) framevar(1); }
        /* Assigning from A */
    { field(mktag(1), r3, 1) = (int) r1; }
        /* Assigning from As */
    { r1 = (int) r3; }
        /* Assigning from HeadVar__1 */
mercury__append_3_1_i1:
    ;
        /* End of disj */
/* Start of procedure epilogue */
/*
 * Live lvalues at end of block:
 *     r1
 *     r2
 */
    { succeed(); }
        /* Succeed */
/* End of procedure epilogue */
END_MODULE
```

C The compiler sources

The compiler sources are located on the Computer Science Department's main server munta.cs.mu.oz.au. The are located in the directory c̃onway/mercury/compiler. The files with a `.nl` suffix are the source files.

The source files that make up the main body of the code generator are: call_gen.nl, code_gen.nl, code_info.nl, code_util.nl, disj_gen.nl, ite_gen.nl, switch_gen.nl, and unify_gen.nl.

The source_stats.awk script in the compiler directory produces the following metrics:

```
Number of types:                  338
Number of insts:                   17
Number of predicates:            2098
Number of modes:                 2129
        - det:                   1803 ( 84.69%)
        - semidet:                323 ( 15.17%)
        - nondet:                  32 (  1.50%)
Average modes per predicate:    1.015

Blank lines:                     8808 ( 20.64%)
Comment lines:                   4523 ( 10.60%)
Total whitespace/comment lines: 13331 ( 31.24%)

Predicate declaration lines:     2781 (  6.52%)
Mode declaration lines:          2198 (  5.15%)
Type declaration lines:          1067 (  2.50%)
Inst declaration lines:            45 (  0.11%)
Module declaration lines:         864 (  2.02%)
Total declaration lines:         6955 ( 16.30%)

Code lines:                     22381 ( 52.46%)

Total number of lines:          42667 (100.00%)
```