Handling large predicates in the Mercury compiler

Zoltan Somogyi

Department of Computer and Information Systems,
University of Melbourne, Australia, and
NICTA Victoria Laboratory
zs@unimelb.edu.au

Abstract. Modern compilers for declarative languages often contain many passes, for program analyses and transformations as well as for code generation. Many of these passes employ algorithms whose complexity is significantly higher than linear. This poses a challenge: how can the compiler handle very large programs in an acceptable time? Humans may not write very large programs, but machines certainly can, and they often do. This paper describes some of the techniques that the Mercury compiler employs to keep compilation times reasonable even when it is tasked with compiling huge Mercury programs.

1 Introduction

Computer scientists who know the history of the field know that programs designed to take input from humans are often stretched to the breaking point and maybe beyond it when given input that has been generated by a machine.

The Mercury project has long had to contend with the reality behind this piece of folk wisdom, because the C code generated by the Mercury compiler has often acted as a stress test of C compilers. When they failed this test, we usually had to change the Mercury compiler to avoid generating the kind of code they had trouble with. For example, when we found out that Microsoft's C compiler cannot handle very long string constants, we changed the code that outputs a module's string table for use by the Mercury debugger to output an array of characters instead. And when we found that the linker on some versions of MacOS X had very bad performance on object files that put each data structure needed by the Mercury debugger in a named global variable of its own, we modified the compiler to partition these variables based on their type, and to replace all variables of the same type with a single array of that type, hugely reducing the number of symbols the linker had to deal with.

However, in the last few years, we also have found ourselves on the opposite side of the fence. Several groups that use Mercury have started generating Mercury code automatically. The Mercury code their tools generate acted as a stress test of the *Mercury* compiler, and in some cases, the compiler failed that test. Some autogenerated Mercury programs it could compile only after working on them for a long time, many minutes or even hours; some it could not compile because it ran out of memory (for either the stack or the heap), and for some, it did not run out of memory, but nevertheless used so much of it that it caused the machine to *thrash* [8] itself to the point of unresponsiveness.

We have found that all such problems reported to us were caused by one or at most a few large predicates in the programs being compiled. Large programs consisting of many small predicates are typically compiled in time that is linear in the number of those predicates, and have not posed any performance problems, at least not so far. In this paper we therefore describe *some* of the most important performance problems that the Mercury compiler used to exhibit when given large predicates, and their solutions. These changes, together with many more that we do not have room to report on, have made the Mercury compiler able to handle, and in most cases handle quickly, even the largest and most complex of our stress tests.

The structure of this paper is as follows. First we present some required background in section 2. The next three sections contain the heart of the paper, descriptions of the performance problems

caused by large predicates and their solutions. We organize those three sections based on the appearance of a predicate that is formatted to put one goal per line, which is a popular way of laying out logic programs.

- In section 3, we show how we handle very *wide* predicates: predicates in which a single line is very long, because a single goal is very big. Given that in real life programs, both predicates and function symbols have limited arities, this typically happens only if the goal contains some very big terms. Given that Mercury does not yet support partially instantiated data structures, in practice these terms are invariably ground terms.
- In section 4, we show how we handle very *tall* predicates: predicates containing many goals. Since the demands of abstraction tend to limit the number of conjuncts in a conjunction to a few hundred even in machine generated code, this tends to happen only if the predicate contains a disjunction with a large number of disjuncts (or equivalently, a large number of clauses).
- In section 5, we show how we handle very *deep* predicates, predicates in which a single narrow and short piece of source code can put disproportionate demands on the compiler, which typically happens in the form of requiring the generation of much larger than usual amounts of target code. In Mercury programs, this usually happens when the predicate defines many higher order constructs and/or uses a large hierarchy of typeclasses.

Normally, one would present all the performance results at the end of the paper, but in this case, most of the material in these sections is independent of each other, so we evaluate the performance of each technique just after presenting it. Our performance results are therefore contained in sections 3, 4 and 5. *No, they aren't. It turned out to be effectively impossible to get any performance results.*

In section 6, we discuss some pragmatic issues, including how we found all the areas that needed improvement.

We do not discuss code generation for large predicates in this paper, since we have dealt with that topic in an earlier paper [3].

2 Background

Mercury is a pure logic/functional programming language intended for the creation of large, fast, reliable programs. While the syntax of Mercury is based on the syntax of Prolog, semantically the two languages are very different due to Mercury's purity and its type, mode, determinism and module systems.

The abstract syntax of the part of Mercury relevant to this paper is:

pred P :	$p(x_1, \ldots, x_n) \ \leftarrow \ G$	predicates
	$\mid f(x_1, \ldots, x_n) = r \ \leftarrow \ G$	functions
goal G :	$x = y \mid x = f(y_1, \ \ldots, \ y_n)$	unifications
	$\mid p(x_1, \ \ldots, \ x_n)$	first order calls
	$\mid x_0(x_1, \ \ldots, \ x_n)$	higher order calls
	$\mid (G_1, \ \ldots, \ G_n)$	sequential conjunctions
	$\mid (G_1 \ \& \ \ldots \ \& \ G_n)$	parallel conjunctions
	$\mid (G_1; \ldots; G_n)$	disjunctions
	$\mid \text{switch } x \ (\ldots; \ f_i : G_i; \ldots)$	switches
	$\mid (if \ G_c \ then \ G_t \ else \ G_e)$	if-then-elses
	$\mid not \ G$	negations
	$\mid some \ [x_1, \ldots, x_n] \ G$	quantifications
	$\mid promise_pure \ G \mid promise_semipure \ G$	purity promises

The atomic constructs of Mercury are unifications (which the compiler breaks down until they contain at most one function symbol each), plain first-order calls, and higher-order calls. The composite constructs include sequential and parallel conjunctions, disjunctions, if-then-elses, negations and existential quantifications. These should all be self-explanatory. A switch is a disjunction in

which each disjunct unifies the same bound variable with a different function symbol. Switches in Mercury are thus analogous to switches in languages like C.

Mercury has a strong Hindley-Milner type system very similar to Haskell's. Mercury programs are statically typed; the compiler knows the type of every argument of every predicate (from declarations or inference) and every local variable (from inference).

Mercury also has a strong mode system. The mode system classifies each argument of each predicate as either input or output; there are exceptions, but they are not relevant to this paper. If input, the caller must pass a ground term as the argument. If output, the caller must pass a distinct free variable, which the predicate or function will instantiate to a ground term. It is possible for a predicate or function to have more than one mode; we call each mode of a predicate or function a *procedure*. The Mercury compiler generates separate code for each procedure of a predicate or function. The mode checking pass of the compiler is responsible for reordering conjuncts (in both sequential and parallel conjunctions) as necessary to ensure that for each variable shared between conjuncts, the goal that generates the value of the variable (the *producer*) comes before all goals that use this value (the *consumers*). This means that for each variable in each procedure, the compiler knows exactly where that variable gets grounded.

Each procedure and goal has a determinism, which may put upper and lower bounds on the number of its possible solutions (in the absence of infinite loops and exceptions): *det* procedures succeed exactly once (upper bound is one, lower bound is one); *semidet* procedures succeed at most once (upper bound is one, no lower bound); *multi* procedures succeed at least once (lower bound is one, no upper bound); and *nondet* procedures may succeed any number of times (no bound of either kind). Goals with determinism *failure* can never succeed (upper bound is zero, no lower bound). Goals with determinism *erroneous* have an upper bound of zero and a lower bound of one, which means they can neither succeed nor fail, so the only things they can do is throw an exception or loop forever.

The Mercury compiler is written in Mercury; it is compiled with previous versions of itself.

The Mercury compiler keeps a lot of information associated with each goal. Amongst other things, this includes:

- the set of variables bound (or *produced*) by the goal, and their new instantiation states or *insts*;
- the *nonlocal set* of the goal, which means the set of variables that occur both inside the goal and outside it; and
- the determinism of the goal.

We ran all our benchmarks on a Dell Optiplex 755 desktop PC with a 2.4 GHz Intel Core 2 Quad Q6600 CPU (four cores, no hyperthreading), four Gb of main memory, running Linux 2.6.31. We ran each test ten times, discarded the highest and lowest times, and averaged the rest.

3 Handling wide predicates

The Mercury compiler's internal representation of predicate definitions uses what we call superhomogeneous form. In standard logic programming terminology, a clause is in homogeneous form if the arguments in the clause head consist of distinct variables [5]. Superhomogeneous form imposes some further restrictions:

- the arguments of calls in the clause body must consist of distinct variables,
- all the variables appearing in a unification must also be distinct, and
- a unification may contain at most one function symbol.

This means that the compiler will internally transform this clause for the `attribute` predicate

```
attribute(attribute_id("calc_ref_premium"),
    attribute(
        attribute_id("calc_ref_premium"), currency,
        multi_text(multi("VGPMPRM", "VGPMPRM (EN)"), no),
        single, no,
        [external(entity_id("chosen_rate_unit"))],
        amount(14, 2), [], no, default, default, 9999,
        no, no, no, no, no, no, no)).
```

into something like this, with the ellipses representing arguments omitted from this example fragment:

```
attribute(HeadVar1, HeadVar2) :-
    HeadVar1 = attribute_id(V1),
    V1 = "calc_ref_premium",
    HeadVar2 = attribute(V2, V3, V4, ...),
    V2 = attribute_id(V5),
    V5 = "calc_ref_premium",
    V3 = currency,
    V4 = multi_text(V6, V7),
    V6 = multi(V8, V9),
    V8 = "VGPMPRM",
    V9 = "VGPMPRM (EN)",
    V7 = no,
    ...
```

The Mercury compiler identifies variables using unique integers, not names, which makes it trivial to create new variables that cannot be confused with existing variables.

The restrictions imposed on unifications by the superhomogeneous form have two main advantages that greatly simplify many operations of the Mercury compiler. The first advantage is that in superhomogeneous form, every part of every term has a simple yet unambiguous handle: the variable that holds its value. A Prolog compiler that does not use superhomogeneous form, when it is traversing a term in an atom, can easily refer to the current subterm and its children, but can refer to other subterms only using complex referents. The second advantage is that superhomogeneous forms allows the Mercury compiler to look at just one thing happening at the program at a time, whether it is

– a procedure receiving a parameter from its caller,
– testing the value of a variable against a template,
– creating a new term and binding it to a variable, or
– passing a parameter to a call.

When a Prolog compiler looks at a clause head, the first three of those actions are all mixed together, and when it looks at a body atom, the last three of those actions are all mixed together.

When the Mercury compiler first flattens clauses into superhomogeneous form, it puts the resulting unification in a top-down order, as shown above. This is because the first compiler pass after flattening that cares about order is type analysis, and type analysis prefers this order. The reason *why* it prefers this order has to do with ambiguity, as shown by the following example.

The Mercury standard library has two types, bool and maybe, that both define the function symbol “`no`” with arity zero:

```
:- type bool ---> no ; yes.
:- type maybe(T) ---> no ; yes(T).
```

The type of the term `[no, no, yes]` is `list(bool)`, while the type of `[no, no, yes(3)]` is `list(maybe(int))`. The superhomogeneous expansion of the first term is `V1 = [V2 | V3], V2 = no, V3 = [V4 | V5], V4 = no, V5 = [V6 | V7], V6 = yes, V7 = []`, while the expansion of the second term is almost the same: it just replaces `V6 = yes` with `V6 = yes(V8), V8 = 3`.

Most terms occur as arguments in the clause head or in a call, and in most Mercury programs, virtually all predicates have type declarations. (Mercury encourages programmers to write down even declarations that the compiler could infer, since those declarations are useful documentation.) This means that when the type checker encounters a sequence of unifications like these, the declared type of the argument will usually tell it what the type of the top level variable is, regardless of whether the term appears as an argument in a clause head or as an argument of a call in the body. Let us say the term that expands to the unification sequence above is argument n in a call. In almost all cases, the compiler will know the declared type of this argument of the called predicate. Suppose it knows that `V1` has type `list(bool)`). Then when it processes `V1 = [V2 | V3]` it knows that `V2` has type `bool` and `V3` has type `list(bool)` even before it sees the following two unifications.

If the flattening process generated a bottom-up expansion of the term, such as reverse of the above or `V2 = no, V4 = no, V6 = yes, V7 = [] V5 = [V6 | V7], V3 = [V4 | V5], V1 = [V2 | V3]`, then the type checker wouldn't know whether `V2` and `V4` have type `bool` or `maybe(T)` for some `T` until it saw the unifications putting them into the same list as `V6`, which is obviously of type `bool`. The extra bookkeeping for recording and eventually resolving the ambiguity would slow down the type checker. Even for a state-of-the-art constraint-based type checker such as the one reported in [2], one would expect the slowdown to involve either a substantial increase in the constant factor in the complexity of the type checking process, the addition of a log(N) factor in its big-O complexity, or both, depending on the implementation details.

As it happens, the impact of non-top-down ordering of unifications on the actual Mercury type checker is much worse. (The Mercury type checker was designed and implemented before [2] was published.) When the Mercury type checker encounters ambiguity such as whether `V2` has type `bool` or `maybe(T)` for some `T`, it duplicates its current set of tentative assignments of variables to types, and assigns to `V2` the type `bool` in one copy and the type `maybe(T)` in the other copy. When it finds a later goal that is inconsistent with a type assignment in a type assignment set, it will drop that type assignment set, but if sees nothing like that, the type assignment sets will keep accumulating. After twenty consecutive two-way ambiguities, the number of type assignment sets will reach a million (2^{20}); after a few more, the compiler will run out of memory. This means the compiler must avoid ambiguities wherever it can. That means that it must give the typechecker large terms in top-down order; in practical terms, other orders won't work.

3.1 The problem with wide predicates: mode reordering

The compiler pass after the type checker that cares about the order of unifications is mode analysis, and it has different and variable requirements about that order: whether mode analysis prefers the top-down or bottom-up order depends on the context in which the term occurs. Here are two contexts with opposite requirements:

```
:- pred test_list(list(bool)::in) is semidet.
test_list([no, no, yes]).

:- pred make_list(list(bool)::out) is det.
make_list([no, no, yes]).
```

One task of mode analysis is to decide, for each call, which mode of the called predicate is being called, and for unifications, whether the unification is a test, an assignment, the construction of a new term, or the deconstruction of an existing term. This means that the mode analyzer decides which variables each goal will produce. Another task of the mode analyzer is to reorder conjunctions so that if a conjunct consumes the value of a variable, it comes *after* the conjunct that generates the value of that variable.

All this means is that in contexts which test the shape of an existing term, such as the occurrence of the list in `test_list`, the mode analyzer will want to put the unifications resulting from the flattening of the term into a top-down order, if they aren't in that order already. In contexts which construct a new term, such as the occurrence of the list in `make_list`, the mode analyzer will want to put the unifications resulting from the flattening of the term into a bottom-up order, if they aren't in that order already.

The code in mode analysis that reorders conjunctions as needed processes the conjuncts given to it left to right. At any point in time, it has a list of conjuncts that it has already scheduled (whose order of execution is now fixed), and a list of conjuncts that remain to be scheduled. At each step, it picks the first conjunct in the second list and tests whether it can be scheduled right after the conjuncts in the first list. Whether the test succeeds depends on the *instantiation state* of the variables in the goal. When the compiler successfully schedules a goal, it updates its record of those variables' instantiation states to simulate the execution of the goal.

If the order of the conjuncts in the original list is a viable order of execution, the schedulability test will always succeed. Consider `test_list`. The body of this predicate consists of the conjunction `V1 = [V2 | V3], V2 = no, V3 = [V4 | V5], V4 = no, V5 = [V6 | V7], V6 = yes, V7 = []`. The initial instantiation state of `V1` is ground, while all the other variables are `free`. When mode analysis looks at `V1 = [V2 | V3]`, it knows it can schedule it as a deconstruction unification that binds `V2` and `V3` to ground terms. It can then schedule `V2 = no`, then `V3 = [V4 | V5]` and so on: all the scheduling tests succeed.

The performance problem arises when the order of the conjuncts in the original list is *not* a viable order of execution. Consider the mode analysis of `make_list`. The original body of this predicate consists of the same conjunction, `V1 = [V2 | V3], V2 = no, V3 = [V4 | V5], V4 = no, V5 = [V6 | V7], V6 = yes, V7 = []`. but now, the initial instantiation state of *all* the variables is free. The attempt to schedule `V1 = [V2 | V3]` will fail, because unlike Prolog, Mercury cannot execute a unification until one side or the other is ground, and in this case, both sides contain variables. The mode analyzer will therefore delay the goal. The attempt to schedule `V2 = no` will succeed, but the attempt to schedule `V3 = [V4 | V5]` will also fail. In general, the attempt to schedule every unification involving a cons cell will fail until the nil at the end of the list has been built.

After every successful attempt to schedule a goal, the mode analyzer will attempt to schedule each of the delayed goals. In this case, all those attempts will fail. After processing n list elements, there will be n delayed goals, so overall mode analysis will make about $(n^2)/2$ such attempts before it gets to the end of the list. The performance problem continues even after it gets there. When there is more than one delayed goal, mode analysis attempts to schedule them in their order in the original conjunction, in an effort to preserve as much of the original order of the conjunction as possible. (If more than one delayed conjunct is schedulable at a given point, then scheduling the one that originally appeared first is required by the language semantics.) This means that after `V2 = no, V4 = no, V6 = yes` and `V7 = []` have been scheduled, mode analysis will attempt to schedule first `V1 = [V2 | V3],` and then `V3 = [V4 | V5],` both of which will fail, and only then attempt to schedule `V5 = [V6 | V7],` which will succeed. Since after every scheduling of a goal that binds a variable that is a part of the list skeleton (in this case, `V1`, `V3`, `V5` and `V7`), we will fail to schedule *all* delayed goals except the last, we will make approximately $(n^2)/2$ scheduling attempts after binding `V7`, as well as before. The complexity of reversing a list of unifications that result from the expansion of a ground term is therefore $O(n^2)$. We have seen the value of n reach several thousand in machine generated code. Reversing such lists with this algorithm is obviously not feasible.

One possible solution would be to change mode analysis to keep track of which variables a delayed goal depends on, and attempt to schedule it only when the instantiation state of at least one of these variables has changed. This would improve the complexity of scheduling a conjunctions like our running example to $O(n \, log \, n)$. The $log \, n$ factor accounts for the fact that the size of the set of delayed goals is itself $O(n)$, and the cost of operations that access such sets is $O(log \, n)$ using always-balanced 2-3-4 trees, which are the best data structures for general sets in the Mercury standard library. (All other data structures we know that could do the job with better complexities

get those complexities by using destructive update. The Mercury compiler needs access to previous versions of data structures reasonably frequently, so those solutions are not appropriate.)

Unfortunately, such a change is hard to make, because it requires changes to some of the most basic data structures used by mode analysis, and would therefore require updating all the other pieces of code that operate on those data structures. This would be very hard, because mode analysis is by a fair distance the most complicated part of the Mercury compiler. It is complicated because it does several jobs at the same time:

1. *Groundness:* It keeps track of whether each variable is bound or free.
2. *Uniqueness:* If a variable is bound, it keeps track of whether the value it is bound to has any references to it other than the one from this variable. (In other words, whether this variable has a unique reference to the value or not. Only unique references allow destructive update.)
3. *Higher order modes:* If a variable is bound, and the variable has a higher order type, it keeps track of the mode of the *arguments* of the predicate that the variable is bound to.
4. *Possible bindings:* If a variable is bound, it keeps track of which function symbols it could be bound to.
5. *Reachability:* It keeps track of whether the program point after the currently scheduled goals is reachable or not.

Task 1 (groundness) is obviously required by Mercury's use of a strong mode system. The others are needed by Mercury's mode system as well, though the need for them may not be as obvious. The four small code fragments in figure 1 each show why mode analysis needs to do one of tasks 2 to 5. *Question to reviewers: are the examples needed? If yes, should the examples of tasks 2 to 4 be replaced with more realistic examples, even though those would be bigger, and require significant exposition of the background? (Task 5 is a paraphrase of real code, to wit, the predicate map.lookup in the Mercury standard library.)*

Task 2 (uniqueness) is needed because without it, the compiler cannot determine that `task2a` is mode correct while `task2b` is not, and must generate a uniqueness error.

Task 3 (higher order insts) is needed because without it, the compiler cannot modecheck the two higher order calls through `Q`.

Task 4 (possible bindings) is needed because without it, the compiler cannot recognize that when execution reaches the inner disjunction, `A` can be bound only to `f` or `g`, and that therefore the inner disjunction, which is a switch on `A`, covers all the values that `A` can be bound to at that program point. It would therefore believe that `task4` is semidet, even if in fact it is det.

Task 5 (reachability) is needed because without it, the compiler would complain about the fact that the else part of that if-then-else does not bind `Value`. It does not have to, because the end of that else part will never be reached; the call to the standard library predicate `unexpected` will abort the program. Mode analysis knows this because `unexpected` has only one declared mode, and that mode is declared to have determinism erroneous.

All changes to mode analysis must maintain correctness with respect to all these tasks. For small changes, this is hard but doable. Unfortunately, changes that require updating large parts of the mode analyzer, such as a redesign of the goal delaying subsystem, are so hard as to be effectively impossible. [1]

3.2 Fixing the mode reordering problem with `from_ground_term` scopes

The key to our solution of this problem is the introduction of a new kind of scope goal. The Mercury compiler already implements several kinds of scopes that each wrap around an inner goal, including scopes that represent existential quantifications and purity promises.

[1] We have plans to replace the current mode analyzer, which is based on abstract interpretation, with one based on generating and solving sets of constraints [7]. A mode analyzer built using such an approach can handle these tasks using additional kinds of constraints (tasks uniqueness, higher order insts and possible bindings), or using the selective non-imposition of some constraints (task reachability). Unfortunately, we haven't had funding to implement a full version of this proposed mode analyzer.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Task 2: uniqueness

:- type tuple
  --->  tuple(int, int, int).

:- type field_id
  --->  first
  ;     second
  ;     third.

:- pred task2a(field_id::in,
  int::in, bool::out) is det.
task2a(FId, N, Result) :-
  T0 = tuple(1, 2, 3),
  destructive_update(FId, N T0, T1),
  T1 = tuple(_, _, C),
  ( if C = 42 then
    Result = yes
  else
    Result = no
  ).

:- pred task2b(field_id::in,
  int::in, bool::out) is det.
task2b(FId, N, Result) :-
  T0 = tuple(1, 2, 3),
  destructive_update(FId, N T0, T1),
  ( if T1 = T0 then
    Result = yes
  else
    Result = no
  ).

:- pred destructive_update(field_id::in,
  int::in, tuple::di, tuple::uo) is det.

destructive_update(FId, N, T0, T) :-
  T0 = tuple(A0, B0, C0),
  (
    FId = first,
    T = tuple(N, B0, C0)
  ;
    FId = second,
    T = tuple(A0, N, C0)
  ;
    FId = third,
    T = tuple(A0, B0, N)
  ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Task 3: higher order insts

:- pred task3(int::in, int::in, int::in,
  int::out, int::out) is det.

task3(A, B, C, Y, Z) :-
  p(C, D),  % should bind D
  Q = (pred(J::in, K::out) is det :-
    K = D * J
  ),
  Q(A, Y),  % should bind Y
  Q(B, Z).  % should bind Z

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Task 4: possible functors

:- type t
  --->  f(int)
  ;     g(int)
  ;     h(int).

:- pred task4(t::in, int::out) is det.
task4(A, Z) :-
  (
    ( A = f(N)
    ; A = g(N)
    ),
    ... long code to compute M from N ...
    (
      A = f(_),
      Z = M
    ;
      A = g(_),
      Z = 2 * M
    )
  ;
    A = h(Z)
  ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Task 5: reachability

:- pred lookup(map(K, V)::in,
  K::in, V::out) is det.
lookup(Map, Key, Val) :-
  ( if search(Map, Key, ValPrime) then
    Val = ValPrime
  else
    unexpected("lookup failed")
  ).

:- pred search(map(K, V)::in,
  K::in, V::out) is semidet.
```

Fig. 1. Examples of mode analysis tasks

These reflect specific constructs in Mercury source programs. The new scope, which we call the `from_ground_term` scope, exists only in the internal representation of Mercury programs. A `from_ground_term` scope can contain only sequences of unifications that result from the conversion of a ground term to a superhomogeneous form. Each `from_ground_term` scope has a subtype that states what invariants the conjunction in the scope observes (we will discuss these soon), and it identifies the variable that represents the whole of the original ground term. The `from_ground_term` scope goal that represents our running example is `scope(from_ground_term(initial, V1), conj([V1 = [V2 | V3], unify(V2 = no), unify(V3 = [V4 | V5]), unify(V4 = no), unify(V5 = [V6 | V7]), unify(V6 = yes), unify(V7 = [])]))`.

An initial `from_ground_term` scope always contains the unifications that result from the expansion of a ground term, and it always contains them in top down order. Before mode analysis starts analyzing the goals inside such as scope, it will test the instantiation state of the variable that represents the whole term. If it is ground, it will leave the goals as they are; if it is free, it will reverse their order. In our running example, the bodies of both `test_list` and `make_list` consist of the scope goal at the end of the previous paragraph. In `test_list`, the initial instantiation state of `V1` will be ground, so mode analysis will leave the conjunction inside the scope unchanged; in `make_list`, it will be free, so mode analysis will reverse the conjunction.[2] In `test_list`, every conjunct will be classified as a deconstruction unification, with information flowing from the left hand side of the equals sign to the right (e.g. picking up the values of `V4` and `V5` from `V3`) and the left hand side variable will always be ground. Mode analysis will therefore mark the scope as a `from_ground_term_deconstruct` scope. In `make_list`, every conjunct will be classified as a construction unification, with information flowing from the right hand side of the equals sign to the left (e.g. constructing `V3` from `V4` and `V5`) and the right hand side variables will always be ground. Mode analysis will therefore mark the scope as a `from_ground_term_construct` scope. *In both cases, mode analysis will not need to delay any goals at all; all attempts to schedule a conjunct will succeed immediately.* This makes the complexity of mode analysis even better than the possible solution discussed in subsection 3.1.

The introduction of a new kind of scope goal required changes in most parts of the Mercury compiler. Nevertheless, it was an easier change to make than changing how mode analysis handled delays. Part of the reason for that is that mode analysis is not just a large piece of code (about 12,000 lines), but it is also (a) complex, and (b) it has lots of coupling, so that changing one part requires changes in most other parts. By contrast, the only *substantial* changes required in the process of adding the `from_ground_term` scopes were

– modifying the parser to construct these scopes (see next subsection);
– modifying some other passes that transform procedure bodies to remove the `from_ground_term` scope if they modified the code inside it in ways that invalidate the scope's invariants.

Modifying mode analysis to reverse the goals inside `from_ground_term` scopes in the right conditions required little code. All compiler passes that had no way to invalidate the scope's invariants (this was most of them) could be trivially modified initially to behave as if the scope wrapper wasn't there. They could later be updated to take advantage of the invariants promised by the scope, but these changes were not coupled; they could happen one at a time, at our leisure. We will discuss some of these changes in section 3.4; discussion of the full list would take too much space.

3.3 Construction of `from_ground_term` scopes

The Mercury compiler uses an unusual three-stage parsing strategy. In the first stage, it converts Mercury code from sequences of characters into terms, using a term parser derived from a parser

[2] Since Mercury does not (yet) support partially instantiated data structures, these are the only two possibilities with respect to groundness, though as noted in section 3.1, there are many possible ground instantiation states, each recording different information about uniqueness, possible values etc.

used by Prolog systems. [3] The second stage converts each term representing a clause into a structure we call the *parse tree*. The parse tree has separate representation for each different kind of goal (conjunctions, disjunctions, if-then-elses, calls, unifications etc), but leaves the arguments of calls and unifications as terms. The third stage converts the parse tree into the format we call HLDS (high level data structure), which is the internal representation of the program that almost all of the compiler modules operate on. It is this third stage that flattens terms, during the conversion of predicate definitions into superhomogeneous form.

One way of detecting ground terms and wrapping `from_ground_term` scopes around the result of their expansion would be to test each term to see whether it is ground *before* it is flattened. This approach has trouble if the term is large, and most of its subterms are ground, but it has a few variables that are encountered late in the term's traversal. In that case, every subterm that contains a variable will be traversed many times.

For example, consider the small term `f(gt1, gt2, g(gt3, gt4, gt5, h(X, a)))`. The tests for whether the terms rooted at `f`, `g`, and `h` are ground all fail. The term rooted at `g` will be tested twice, once for itself and once as part of the test of `f`, while the term rooted at `h` will be tested three times, once for itself, and once each as a subterm of the terms rooted at `f` and `g`. In general, each nonground term will be tested as many times as the number of function symbols above it, which can be a substantial number. The worst-case complexity of this algorithm is therefore pretty bad. In practice, it actually turns out to perform surprisingly well, because large non-ground terms are very rare in Mercury programs.

We nevertheless wanted an algorithm with a linear worst case complexity. Our algorithm, when it flattens a term, returns not only the sequence of superhomogeneous unifications that result from that process, but also an indication of whether the term was ground or not. If some argument terms of a function symbol are ground and some are nonground, then our algorithm can wrap a `from_ground_term` scope around the expansions of the ground argument terms. It can also wrap such scopes around the results of expanding whole terms. Note "can", not "will'. This is because we have found that putting `from_ground_term` scopes around short sequences of unifications, such as those resulting from the flattening of small terms like `f(a)`, actually slows the compiler down. The reason is simply that the presence of the scope makes the HLDS representation of the procedure it occurs in bigger, and every compiler pass has to do some work to handle the scope, even if that handling consists merely of deciding to disregard its presence, and proceeding to process the goal within. If the number of unifications inside the scope is small enough, then the benefit the scope gets from making mode analysis more efficient will not be big enough to counterbalance that overhead. Our algorithm therefore keeps track of the number of function symbols inside each ground term, and wraps a `from_ground_term` scope around the expansion of such a term only if this number, which will also be number of unifications in that expansion, exceeds a configurable threshold.

The type analyzer can discover that what looked like the construction of a ground term from some ground subterms is actually a function call, because what looks like a function symbol is actually the name of an evaluable function. When this happens inside an `from_ground_term_construct` scope, the compiler will have to remove that scope since it won't know what the output of that function call would be, but it will put smaller `from_ground_term_construct` scopes around the unification sequences that construct the arguments of that function call, as well as around the unification sequences that construct the siblings of that function call at all ancestor levels. For example if in the term `f(gt1, gt2, g(gt3, gt4, gt5, h(gt6, gt7)))` the function symbol `h` turned out to represent a function call, we would delete the `from_ground_term_construct` scope around the expansion of the whole term but put smaller `from_ground_term_construct` scopes

[3] We did this to ensure compatibility with Prolog at a lexical level during the initial implementation of Mercury. For the first few years of the Mercury project, the Mercury compiler was written in the intersection of Mercury, NU-Prolog and SICStus Prolog; to execute the compiler with Prolog, we just stripped out all the Mercury declarations. This was the only way to execute the Mercury compiler before it was bootstrapped, and it remained the only way to *debug* the Mercury compiler until the Mercury debugger was implemented.

around the goals resulting from all of gt1, gt2, gt3, gt4, gt5, gt6 and gt7. This preserves as high a fraction as possible of the benefits of the original from_ground_term_construct scope.

3.4 Other uses of from_ground_term scopes

Besides their main purpose of greatly improving the complexity of scheduling, the compiler exploits the special properties of from_ground_term scopes in many other ways.

Some compiler passes can completely avoid traversing the goal inside an from_ground_term scope. One example is the compiler pass that looks for singleton variables (variables the occur only once, and whose names do not begin with an underscore) and reports a warning for them. By construction, every variable inside an from_ground_term scope occurs exactly twice. There is only one exception, the variable representing the whole term, whose identity is recorded in the from_ground_term scope wrapper; this pass need only test this single variable.

Some compiler passes still need to traverse the goals inside such scopes, but can exploit their special properties to improve either the big-O complexity of the code they execute, its constant factor, or both. An example is mode analysis itself. It can exploit the same special property of all variables in the scope except one appearing exactly twice. The first occurrence of a variable gives it a value; the second picks up that value; and then the variable won't be referred to again. Knowledge of this fact allows mode analysis to throw away its record of the instantiation state of every variable after it has seen a consumption of the value of that variable. When processing most ground terms, this keeps the size of the *instmap*, the compiler data structure that maps variables to their current instantiation state, at or below a very small bound. When processing a list of booleans, the instmap will never need to hold more than two variables. In the absence of this invariant, mode analysis would have to keep the instantiation state of every variable it has seen so far in a conjunction. Since every goal in from_ground_term_construct scope constructs a new variable, a scope containing n unifications would otherwise require the compiler to record the instantiation state of $n/2$ variables on average. The cost of simply looking up the current instantiation states of the variables inside the goal being scheduled would therefore add a factor of $O(log\ n)$ to the complexity of mode analysis.

One of the tasks of mode analysis is updating the data structure representing each scheduled goal to record what changes, if any, its execution will make in the instantiation states of the variables occurring in the goal. Mode analysis has to traverse the unifications inside from_ground_term_construct scopes to record this information, which is just one of the many items of information that the Mercury compiler records for each goal. Normally, these other items of information would be computed and then recorded by other compiler passes. However, for some of these items of information, we know how those slots would be filled in later, and we can save a traversal of the scope by filling them in during mode analysis. For example, we know that all the unifications in a from_ground_term_construct goal that constructs a ground term have determinism det. This can be thought of either as improving the complexity of determinism analysis's handling of the scope from linear to constant, or as simply reducing the constant factor of the combined cost of the traversals of the scope during mode and determinism analysis.

In almost all cases, we want to translate the contents of a from_ground_term_construct scope to constant data structures, as in Figure 2. However, in some cases, the ground term is the initial value of a data structure that the program wants to update in-place using compile-time garbage collection, a technique that allows the new version of a data structure to reuse the storage of the old version if the old version is provably unreachable after the update [6]. Such updates are done by predicates whose arguments representing the old and new versions of those data structures have modes destructive input and unique output (di and uo) respectively. Having mode analysis set the instantiation states of the ground terms and subterms being constructed by a from_ground_term_construct scope to a unique inst would require them to be changed to take away the uniqueness (since the code generator will *not* generate code like the code in Figure 2 for unifications that construct unique terms). This would require an extra traversal of the scope. On the other hand, recording a nonunique inst for all the variables constructed inside

```
static const MR_Box bools_scalar_common_1[3][2] = {
  {
    ((MR_Box) ((MR_Integer) 1)),
    ((MR_Box) (((MR_Word *)((char *)((((MR_Integer) 0) << 3)) + ((0))))))
  },
  {
    ((MR_Box) ((MR_Integer) 0)),
    ((MR_Box) (((MR_Word *)((char *)(&bools_scalar_common_1[0]) + ((1))))))
  },
  {
    ((MR_Box) ((MR_Integer) 0)),
    ((MR_Box) (((MR_Word *)((char *)(&bools_scalar_common_1[1]) + ((1))))))
  },
};
```

Fig. 2. The representation of `[no, no, yes]` in generated C code.

`from_ground_term_construct` scopes will generate a mode error if such variables are ever passed in an argument position that specifies mode 'di' for that argument.

Our solution of this problem is to record a nonunique inst for variables constructed inside `from_ground_term_construct` scopes *initially*, but also record that the procedure contained such a scope. If the mode analysis of a procedure fails, and the flag says that the procedure contains such a scope, then the mode error may have been caused by the lack of uniqueness. The compiler will therefore mode analyze the procedure again, but this time, it will record a unique inst for variables constructed inside `from_ground_term_construct` scopes. It will regard the results of this second analysis as definitive. This approach preserves correctness in all cases while also retaining good performance in the common case.

It nevertheless has some potential for collateral damage. If a procedure has several `from_ground_term_construct` scopes, only *some* of which construct variables that need to be unique, our approach will construct the terms in *all* of those scopes dynamically. This flaw could be fixed by having mode analysis try to figure out, when the scheduling of a goal fails due to a variable being nonunique when the goal requires it to be unique, recording this information, and using it to turn the `from_ground_term_construct` scopes of only the affected variables unique. However, this would require changes in the data structures used by mode analysis, which (as we discussed at the end of section 3.2) is quite a hard task. It is a task we do not plan to carry out until we have seen a real-life program that needs that capability.

4 Handling tall predicates

The previous section considered some of the problems of clauses with many conjuncts. In this section, we consider some of the problems of caused by predicate definition with many clauses, or equivalently, predicate definitions containing disjunctions with many disjuncts. Programmers may write the disjunction explicitly, as in the example predicate on the left of Figure 3, or they may write clauses that the compiler will turn into disjunctions, as in the example predicate on the right of that Figure.

4.1 The main problem with tall predicates: merging many insts

As we mentioned earlier, one of the tasks of mode analysis is to keep track of which function symbols each variable may be bound to. If the variable is bound inside a disjunction, then different disjuncts may unify that variable with different function symbols.

The simplest algorithm for keeping track of the possible bindings of variables in disjunctions is to keep the set of the function symbols that each such variables may be bound to, and to

```
:- pred is_alnum_or_underscore
    (char::in) is semidet.

is_alnum_or_underscore(Char) :-
    ( Char = '0'
    ; ...
    ; Char = '9'
    ; Char = 'a'
    ; ...
    ; Char = 'z'
    ; Char = 'A'
    ; ...
    ; Char = 'Z'
    ; Char = '_'
    ).
```

```
:- pred edge(int::in, int::out)
    is nondet.

edge(1, 2).
edge(2, 3).
edge(3, 4).
edge(4, 5).
edge(5, 6).
...
edge(7996, 7997).
edge(7997, 7998).
edge(7998, 7999).
edge(7999, 8000).
edge(8000, 1).
```

Fig. 3. Some example tall predicates.

```
HeadVar1 -> 1                      HeadVar2 -> 2
HeadVar1 -> 1,2                    HeadVar2 -> 2,3
HeadVar1 -> 1,2,3                  HeadVar2 -> 2,3,4
HeadVar1 -> 1,2,3,4                HeadVar2 -> 2,3,4,5
...                                ...
HeadVar1 -> 1,2,3,4,...,7999       HeadVar2 -> 2,3,4,5,...,8000
HeadVar1 -> 1,2,3,4,...,7999,8000  HeadVar2 -> 1,2,3,4,5,...,8000
```

Fig. 4. Computing bindings for the edge predicate using insertion esort.

update this set after processing each disjunct by merging this set with the possible bindings from the disjunct. The Mercury compiler originally followed this algorithm. This meant that when compiling the `edge` predicate from Figure 3, the compiler's record of the possible values of the two variables representing the arguments of `edge`, which the compiler names `HeadVar1` and `HeadVar2`, would evolve as shown in Figure 4. The nth line in that figure shows these variables' possible bindings after processing all disjuncts up to and including the nth disjunct.

It is easy to see that this algorithm is quadratic in the number of disjuncts. In this case, it has to process 8000 disjuncts, and while processing each disjunct, it has to add a new function symbol (an integer in this case) at the end of the list of the function symbols that the variable may be bound to. The reason why it has to add them at the end is that the compiler keeps the list of possible function symbols sorted. In the usual case, this means that both successful and unsuccessful searches testing whether a given function symbol in this list will have to traverse half the list on average. However, in this degenerate case, all the searches fail, and all these unsuccessful searches have to traverse entire list.

Representing the set of possible bindings not as a sorted list but as an unsorted list looks like an attractive option, since in the above case it would allow us to add the possible bindings from the just-processed disjunct to the *front* of the list of function symbols. Unfortunately, while this approach can yield a speedup in this case, it can yield significant slowdowns in more common cases. Consider the code in Figure 5. We definitely do not want to represent the set of function symbols that `Changed` can be bound to as `{yes,yes,yes,...,no}`. To avoid creating a memory leak, an insertion into an unsorted list would first need to check whether the function symbol to be inserted was already in the list or not. In the case of the `edge` predicate, it won't be, so the membership test would need to scan the entire list anyway, and the overall time complexity of the algorithm would still be quadratic. By inserting all function symbols at the start of the list, it *could* reduce the amount of memory allocated by those insertions from quadratic to linear. However, later passes of the compiler will also do searches on the list of function symbols. For example, when determinism analysis looks at a switch on a variable, it needs to know the set of

```
process(DataIn, ..., DataOut, Changed) :-
    (
        DataIn = alternative1(...),
        ...
        Changed = yes
    ;
        DataIn = alternative2(...),
        ...
        Changed = yes
    ;
        DataIn = alternative3(...),
        ...
        Changed = yes
    ;
        ...
    ;
        DataIn = alternativeN(...),
        DataOut = DataIn,
        Changed = no
    ).
```

Fig. 5. Example: many switch arms set Changed to the same value.

function symbols that the variable could be bound to at the program point just before the switch, so it could compare this to the set of function symbols covered by the switch. The complexity of the algorithms needed for this task, as well as most others that look at binding information, would be significantly better if the list is sorted.

We have therefore adopted a different solution to this problem. This solution builds up a sorted list of function symbols, but whereas the original algorithm effectively constructed this list using insertion sort, we build it using merge sort.

Our algorithm starts by recording the instantiation state of each variable in each disjunct. That instantiation state may say that e.g. variable `X` may be bound to any of the function symbols `f1`, `f2`, ..., `fn`, but in the edge predicate, it will say that `HeadVar1` is bound to a single integer (the number of a node in a graph). Our algorithm then performs a series of passes. Each pass iterates over a list of insts for each variable, merging up to four consecutive insts into one. Since each pass reduces the number of insts in the list by a factor of four, the number of passes we need is logarithmic in the number of disjuncts, and even if the original list is sorted, as for `edge`, most of the merges add new items to the end of a *short* sorted list. Figure 6 shows the evolution of the bindings of `HeadVar1` with this algorithm.

4.2 Another problem with tall, wide predicates: merging complex insts

For variables of atomic types like `HeadVar1` in `edge`, recording its possible bindings simply requires recording the list of function symbols it may be bound to; by the definition of what it means for a type to be atomic, all these function symbols will be constants. For variables of nonatomic types, recording the list of function symbols is not enough; the Mercury compiler also wants to know the possible bindings of the *the arguments* of these function symbols, if these are known, and the possible bindings of the arguments of those arguments, and so on.

This poses a performance problem. The inst of a variable bound to a ground term will actually be *bigger* than the ground term: it will contain all the function symbols in the ground term, and more. You can judge *how much* bigger from this example, which shows the internal representation

```
before pass 1       after pass 1            after pass 2
HeadVar1 -> 1       HeadVar1 -> 1,2,3,4     HeadVar1->1,...,16
HeadVar1 -> 2
HeadVar1 -> 3
HeadVar1 -> 4
HeadVar1 -> 5       HeadVar1 -> 5,6,7,8
HeadVar1 -> 6
HeadVar1 -> 7
HeadVar1 -> 8
...                                         HeadVar1->7985,...,8000
HeadVar1 -> 7997    HeadVar1 -> 7997,7998,7999,8000
HeadVar1 -> 7998
HeadVar1 -> 7999
HeadVar1 -> 8000
```

Fig. 6. Computing bindings for the edge predicate using mergesort.

of the inst of a variable that could be bound to either of the two ground terms `f(g(h))` and `f(i(j))`:

```
bound(shared, [
    bound_functor(f/1, [
        bound(shared, [
            bound_functor(g/1, [
                bound(shared, [
                    bound_functor(h/0, [])
                    ])
                ])
            ])
        ]),
        bound(shared, [
            bound_functor(i/1, [
                bound(shared, [
                    bound_functor(j/0, [])
                    ])
                ])
            ])
        ])
    ])
```

Even though merge sort requires many fewer comparisons than insertion sort, the number of comparisons it needs can still hurt if the things being compared (in this case, insts) are big. This is especially so if they need to be traversed almost to the end to find a difference, as they often would for a predicate like this:

```
p([yes]).
p([no, yes]).
p([no, no, yes]).
p([no, no, no, yes]).
p([no, no, no, no, yes]).
p([no, no, no, no, no, yes]).
```

Some of our stress tests have many clauses, with each clause containing some large ground terms. Unlike the lists of booleans above, these ground terms weren't constructed specifically to illustrate a worst-case scenario, but their much larger size still makes their comparisons slow. We

have therefore adopted an approach that can totally eliminate the requirement for such comparisons.

The insight behind this approach is the observation that predicates that build large numbers of large terms tend not to do anything else. Specifically, we have never seen them *use* the variables representing those terms in ways that require knowing their possible bindings. Therefore our system, as it gathers the bindings for each variable in each disjunct, it also figures out which variables, if any, have their values defined by `from_ground_term_construct` scopes in *all* disjuncts. Then, before it starts the mergesort, it will by default replace the insts of all those variables with just plain `ground`, throwing away all its information about the precise shapes of the terms that may be bround to the variable. Strictly speaking, this violates the Mercury language specification, so users can disable this behavior with a compiler option, but we have never seen a need for using that option. On the other hand, the default behavior is essential for acceptable compiler performance on pretty much all the tall, wide predicates we have seen.

There is a obvious extension of this proposal that also warrants consideration. The idea is to set the inst of *all* variables generated by disjunctions to ground, not just variables defined by large ground terms, *unless* their information about their bindings is needed by later code.

This extension has an obvious upside, an obvious downside, and a non-obvious downside. The obvious upside is that it totally avoids the need for even the $n \log n$ sorting algorithm demonstrated by 6, allowing most disjunctions to be processed in linear time. The obvious downside is that it requires an extra traversal of the procedure body to figure out *which* variables need precise binding information. The non-obvious downside is that this traversal needs to be able to predict what binding information *later* compiler passes may need. Since adding a new compiler pass is not an infrequent event, this would be a nontrivial maintenance burden. It is mainly because of this last consideration that we have not implemented this extension.

5 Handling deep predicates

The previous sections have dealt with performance problems caused by predicates that were visibly big. However, in some cases, the input to compiler passes can be big *without* this fact being obvious in the source code. In a sense, these predicates are *deep*: the volume of the task they present to the compiler is not visible on a two-dimensional page or screen.

5.1 One source of depth: many lambda expressions

Mercury supports higher order programming. One form of this support is allowing programs to contain higher order constants in the form of lambda expressions, like this:

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

lambda_example(A, B, List, OutputX, OutputY) :-
    FuncX = (func(FXIn::in) = (FXOut::out) is det :- FXOut = FXIn + A),
    FuncY = (func(FYIn::in) = (FYOut::out) is det :- FYOut = FYIn * B),
    OutputX = list.map(FuncX, List),
    OutputY = list.map(FuncY, List).
```

This example unifies each of the variables `FuncX` and `FuncY` with a higher order value.

During the initial passes of the Mercury compiler, the first two lines of `lambda_example` are both represented internally as unifications in which the right hand side is a lambda expression consisting of a goal, an argument list, including a mode for each argument, and some other information. This is necessary during the semantic analysis passes of the Mercury compiler. For example, the type analyzer can use the types of the variables outside a lambda expression, such as `A`, to help it infer the types of variables inside the lambda expression, such as `FXIn` and `FXOut`. (The information can also flow in the other direction.) The Mercury compiler uses a data structure

called the *vartypes map* to record the types of all the variables occurring in a predicate, regardless of whether those variables occur inside a lambda expression, outside it, or in both kinds of places.

On the other hand, each lambda expression is effectively an anonymous predicate, and once semantic analysis is complete, a compiler pass called the lambda expansion pass materializes those predicates, yielding the following code:

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

lambda_example(A, B, List, OutputX, OutputY) :-
    FuncX = closure_constructor<lambda_func_1>(A),
    FuncY = closure_constructor<lambda_func_2>(B),
    OutputX = list.map(FuncX, List),
    OutputY = list.map(FuncY, List).

:- func lambda_func_1(int::in, int::in) = (int::out) is det.
lambda_func_1(A, FXIn) = FXOut :-
    FXOut = FXin + A.

:- func lambda_func_2(int::in, int::in) = (int::out) is det.
lambda_func_2(B, FYIn) = FYOut :-
    FYOut = FYin * B.
```

Each of the newly created predicates includes a sequence number in its name for uniqueness. The arguments of these predicates include not just the listed arguments of the original lambda expression, but also the variables that the lambda expression 'captures' from the surrounding code. The former have their listed modes, the latter are all input.

Once the new procedures have been created, the lambda expansion pass replaces each original unification with a lambda expression with a construction unification that constructs a closure. The closure will contain a pointer to the code of the new predicate, a count of the number of the captured variables in the closure, and the captured variables themselves. The closure will also contain a closure descriptor, a pointer to a static data structure whose contents describe the types of the hidden variables. This is used e.g. by the debugger to print out closures.

In the case of the first line of `lambda_example`, the closure being assigned to `FuncX` will contain the code address of `lambda_func_1`, and one hidden argument value, the value of `A`. The calls to `FuncX` from within `list.map` will add this value to each element of `List`.

Since the variables in the new function (`lambda_func_1`) are a subset of the variables in the old function (`lambda_example`), the Mercury compiler has traditionally simply copied the maps containing information about variables (their types, their names) from the old predicate to the new. This is fast, since it copies only a few pointers. The downside is that the tables in each predicate are larger than they need to be, since typically both procedures' tables contain some entries for variables that do not occur in that procedure.

This is almost always okay, but in rare circumstances, this downside could be a performance problem. The problem occurs in a later compiler pass that expands out equivalence types. An equivalence type is a type that is defined to be just another name for an existing type, like this:

```
:- type prog_var == int.
```

Such type definitions can help the readability of the program. For example, this definition says that program variables are implemented as integers.

Often the equivalence is private to a module. The module defining the new type (in this case `prog_var`) exports the type and some operations on the type, but keeps the definition of the type hidden, like this:

```
:- interface.
:- type prog_var.
...
:- implementation.
:- type prog_var == int.
...
```

We call this an *abstract export* of the type. This prevents any other module of the program from using operations on the type except the operations exported with it. This in turn allows the maintainers of the module to change the definition at any time, knowing that the only code they will have to update will be the code implementing those exported operations.

When performing semantic checks on the code of module A, the Mercury compiler does not know (because it deliberately refrains from looking up) the definitions of abstract exported types defined in other modules. However, once semantic analysis is complete, the compiler could use that knowledge to generate better code. For example, it can use intermodule inlining on calls to the operations on that abstract exported type, and can then compile the Mercury code inside those operations into C code that operates on the actual types (integers in this case). This is why the Mercury compiler has a pass that expands out equivalence types.

As we mentioned above, the original version of the lambda expansion pass simply copied the vartypes map of the original procedure to become the vartypes maps of the anonymous procedures created to implement lambda expressions in the original procedure, and this pass occurred *before* the type equivalence expansion pass. This arrangement caused a huge performance problem when compiling a predicate in `zm_enums.m`, a Mercury program automatically generated by the G12 project's Zinc-to-Mercury compiler. This predicate had about 6,200 variables and about 200 lambda expressions. This gave the type equivalence expansion algorithm the task of processing about 1,240,000 entries in all those vartypes maps. Since expanding each type allocates the memory needed to represent the expanded type *and* the memory needed to put this expansion back into the vartypes map, which takes a few tens of bytes on average, this task quickly caused to the compiler to use more memory than typical machines had back then (this was in 2009) or have even now. By destroying sharing between the different copies of the table, it also permanently and massively increased the memory footprint of the compiler for the rest of the compiler's lifetime. The resulting memory exhaustion typically initiated thrashing in the virtual memory system, and (if the other programs running on the machine required enough memory) could cause the OS to kill a process to stop the trashing. On machines whose kernels allows the overcommitment of main memory, this was not always the compiler process that caused the thrashing!

There are two obvious possible fixes of this problem. The first is to simply swap the order of the lambda expansion pass and type equivalence expansion pass. The second is to restrict the vartypes maps of the predicates touched by the lambda expansion pass (both the original predicates and the newly created lambda predicates) to contain just the variables mentioned in their bodies. We chose the latter solution, even though it takes slightly more time, because the smaller vartypes maps have the added benefit of speeding up lookups in those maps during later compiler passes.

5.2 Another source of depth: complex typeclass hierarchies

The example predicate in the previous subsection had about 200 visible lambda expressions, but it did *not* have about 6,200 visible variables. It had only a few hundred visible variables; all the other variables it had were added automatically by the compiler, specifically by the compiler transformation that adds RTTI (run-time type information) to the program.

This compiler pass is named the *polymorphism transformation*, because it was originally designed to implement parametric polymorphism. Consider the `set.is_member` predicate:

```
:- pred set.is_member(map(T)::in, T::in) is semidet.
```

This is a polymorphic predicate because its type signature contains some type variables, in this case the type variable T. This means that this predicate can check set membership regardless of the type of the elements of the set. Given a call to this predicate, in code like this:

```
:- pred var_is_nonlocal(hlds_goal::in, prog_var::in) is semidet.

var_is_nonlocal(Goal, Var) :-
    get_goal_info(Goal, GoalInfo),
    goal_info_get_nonlocals(GoalInfo, NonLocals),
    set.is_member(NonLocals, Var).
```

the code of `set.is_member` needs to know how to compare two items in the set (in this case, two variables) for equality. The way we give it that information is through data structures we call *typeinfos* and *typectorinfos*.

Every type definition in a Mercury program actually defines a *type constructor*. For example, in Mercury there is no such thing as *the* list type, because `list` is a type constructor of arity one; given a type such as `bool`, it constructs another type, in this case `list(bool)`. A type constructor does not denote a type on its own unless its arity is zero. The Mercury compiler therefore generates a typectorinfo (short for 'type constructor info') for every type definition. Each typectorinfo is a static data structure that records the set of function symbols defined by the type definition, together with the representation chosen by the compiler for each of those function symbols; this information is used by the predicates in the Mercury standard library that read and write values of arbitrary types. Each typectorinfo also contains the addresses of the predicates that implement unification and comparison for types constructed by this type constructor.

If the type constructor has arity zero, then the unification and comparison predicates will be monomorphic (i.e. not polymorphic). Therefore the test for equality in the code of `set.is_member`, which the programmer probably wrote as

```
    Item = SetElement,
```

and which the polymorphism pass has transformed into

```
    unify(TypeInfo_for_T, Item, SetElement),
```

can perform its task simply by having the builtin `unify` predicate pull the address of the type-specific unification predicate for program variables out of the typectorinfo in the typeinfo given to it, and jump to that address.

The variable `TypeInfo_for_T` will get its value from the extra argument added to the call to `set.is_member` by the polymorphism pass. The typeinfo for a type `TypeCtor(ArgType1, ArgType2, ..., ArgTypeN)` is constructed by the special function symbol `type_info_constructor` applied to (a) the typector info for `TypeCtor`, and (b) the typeinfos of `ArgType1`, `ArgType2`, ... `ArgTypeN`. Following this scheme, the polymorphism pass would transform the above example into

```
var_is_nonlocal(Goal, Var) :-
    get_goal_info(Goal, GoalInfo),
    goal_info_get_nonlocals(GoalInfo, NonLocals),
    TypeCtorInfo_for_ProgVar = type_ctor_info_for_<prog_var>,
    TypeInfo_for_ProgVar = type_info_constructor(TypeCtorInfo_for_ProgVar),
    set.is_member(TypeInfo_for_ProgVar, NonLocals, Var).
```

Actually, our RTTI system has long had an optimization that allows the typectorinfo of an arity-zero type to act as a typeinfo containing just that typectorinfo [4], so the polymorphism pass actually generates the following code instead.

```
var_is_nonlocal(Goal, Var) :-
    get_goal_info(Goal, GoalInfo),
    goal_info_get_nonlocals(GoalInfo, NonLocals),
    TypeCtorInfo_for_ProgVar = type_ctor_info_for_<prog_var>,
    set.is_member(TypeCtorInfo_for_ProgVar, NonLocals, Var).
```

The creation of a typeinfo cannot be optimized away if the type constructor has a nonzero arity. For example, in a predicate in which items in `Set` have type `list(bool)`, the polymorphism pass would transform the call `set.is_member(Set, Item)` into

```
    TypeCtorInfo_for_Bool = type_ctor_info_for_<bool>,
    TypeCtorInfo_for_List = type_ctor_info_for_<list>,
    TypeInfo_for_ListBool = type_info_constructor(
        TypeCtorInfo_for_List, TypeCtorInfo_for_Bool)
    set.is_member(TypeInfo_for_ListBool, Set, Item)
```

When the unification predicate for lists needs to know how to unify two elements of the list, it gets the address of the unification predicate for list elements from the typeinfo of the list elements, which in this case will be `TypeCtorInfo_for_Bool`. Our scheme works for types containing arbitrarily many type constructors nested arbitrarily deep [4].

In Mercury, values of every type are presumed to be unifiable and comparable, which is why typectorinfos contain the addresses of unification and comparison predicates. Mercury also supports *typeclasses*. A typeclass is the specification of a set of operations (predicates and/or functions); a type is a member of the typeclass if it has an instance declaration giving the names of the predicates and/or functions that implement the operations of the typeclass on values of that type. Membership in one typeclass may require membership in another typeclass, and the membership of a type in a typeclass may also imply membership of e.g. lists of elements of that type in that typeclass.

The Mercury runtime system implements typeclasses using typeclassinfos. These are similar to typeinfos in that they contain the addresses of operations (in this case, the operations of the typeclass), as well as typeclass infos for any superclasses, and typeinfos representing the actual types. The difference is that the polymorphism pass will build big typeinfos only for types that are visibly big, but it can build big typeclass infos even when this fact is not immediately apparent in the source code.

The motivating example for the work reported in this subsection were calls that looked as innocent as this:

```
    post_constraint(Constraint, ConstraintStore0, ConstraintStore)
```

However, the predicate being called had four typeclass constraints on it, which meant that the type of `Constraint` had to be a member of four typeclasses, and that every call to this predicate had to pass a typeclassinfo for every one of these four typeclasses. The polymorphism pass therefore translated the above call into

```
    ...
    TypeClassInfo_for_Interval_Solver = typeclass_info_constructor(...)
    ...
```

[4] The first field of each typectorinfo is the arity, while the first field of each typeinfo is a (non-null) pointer to the typectorinfo of the top level type constructor. The size of both fields is one machine word. This allows the runtime system to tell typeinfos apart from the typectorinfos of arity-zero type constructors: the first word of the former *cannot* contain a zero, while the first word of the latter *will* contain zero.

```
TypeClassInfo_for_Domain_Events = typeclass_info_constructor(...)
...
TypeClassInfo_for_FD_Solver = typeclass_info_constructor(...)
...
TypeClassInfo_for_LP_Solver = typeclass_info_constructor(...)
post_constraint(TypeClassInfo_for_Interval_Solver,
    TypeClassInfo_for_Domain_Events,
    TypeClassInfo_for_FD_Solver,
    TypeClassInfo_for_LP_Solver,
    Constraint, ConstraintStore0, ConstraintStore)
```

Some of the typeclasses involved were part of a deep hierarchy of typeclasses, so in actuality, the call was preceded by 138 unifications. That is a lot! What initially looked a very small piece of code turned out to be a fairly big piece of code. And the original call was part of a long sequence of calls all needing mostly the same typeinfos and typeclassinfos. The difference between the few hundred visible variables and 6,200 actual variables in the 200-times-duplicated predicate we discussed above was a large set of variables added by the polymorphism pass to hold typeinfos and typeclass infos.

The first change we made to the compiler in an attempt to fix this problem was to record which variables hold which typeinfos and typeclass infos, and to reuse these variables if a later piece of code needed an already existing typeinfo or typeclass info. This works, and works well, if the hit rate of this cache is reasonably high. Fortunately, the hit rate is in fact high in *most* procedures that need more than handful of typeinfos and/or typeclass infos.

However, simply caching variables holding typeinfos and typeclass infos is not a good idea in *all* cases. The reason is simple: caching typeinfos and typeclass infos in variables works *only* if the procedure in which the cached value is needed is the same procedure that did the caching in the first place. If the compiler constructs some typeinfos and typeclass infos to handle a call in a lambda expression, the variables holding those typeinfos and typeclass infos cannot be reused in other lambda expressions or in the main predicate without making those variables effectively new outputs of the caching lambda expression. Returning those variables from that lambda expression and passing them to all the other lambda expressions would slow down the generated code considerably. Since that is a cost we are not willing to pay, we need some other technique to avoid having many lambda expressions each containing copies of the same unifications constructing the same typeinfos and typeclass infos.

Our second technique for solving this problem is much more radical. It involves adding a whole new concept to the internal representation of Mercury programs, the concept of a pool of constant data structures that exists *separately from* and *outside* the definition of any single procedure. [5] Each of these constant structures holds the content of a static memory cell. The arguments of such a cell may be simple constants such as integers, references to other kinds of constant data structures (such as typectorinfos), or references to earlier constant structures. References of this last kind take the form of a simple index into a conceptual array of constant structures.

The module in the Mercury compiler that manages constant data structures remembers the contents of every one of those structures. Once a structure with a given content is created, any later request for a structure with the same content will return the index allocated to the original request for the same content. Whenever the polymorphism pass needs to pass a ground typeclass info to a procedure, it will now search the constant structure database for an existing reference to that typeclass info. If it does not yet exist, it will enter it into the database, get back an index for

[5] As shown by Figure 2, the Mercury code generators targeting C have long been able to translate code that build constant data structures into the definitions of C data structures and references to those definitions, but the results of these translations were not subject to any further processing by the Mercury compiler. The constant data structures we are talking about here *are* subject to such processing. For example, the Mercury compiler's termination analysis pass traverses them to compute their size, and if some compiler pass optimizes away the last reference to a specific constant data structure, the compiler's dead code elimination pass will optimize away that constant data structure itself.

it, and unify the variable intended to hold the typeclass info with a reference to this entry in the constant structure database, like this:

```
TypeClassInfo_for_Interval_Solver = const_struct_reference(20),
TypeClassInfo_for_Domain_Events = const_struct_reference(46)
TypeClassInfo_for_FD_Solver = const_struct_reference(84)
TypeClassInfo_for_LP_Solver = const_struct_reference(138)
post_constraint(TypeClassInfo_for_Interval_Solver,
    TypeClassInfo_for_Domain_Events,
    TypeClassInfo_for_FD_Solver,
    TypeClassInfo_for_LP_Solver,
    Constraint, ConstraintStore0, ConstraintStore)
```

In every later situation in which the compiler needs the same ground typeclass info, it will find that the typeclass info is already in the database, and will be able to use the same reference. This will happen even if the two uses of that ground typeclass info (or typeinfo for that matter) are in different lambda expressions inside the same procedure, or even if they are in different procedures. Effectively we are caching the common references to the same compiler-generated data structure at the earliest possible point, the point at which those references are generated in the first place. This reduces the compiler's memory requirements, and, by avoiding the traversal of multiple copies of the same code, it speeds up the compiler as well.

The constant data structures added to the program by the polymorphism pass are "born correct", and do not need to be checked. The constant data structures built by `from_ground_term` scopes have no such guarantee; they do need to be subject to Mercury's usual semantic checks. Having the parser put ground terms in user-written Mercury code into the constant structure database would require us to duplicate all the predicates that do any part of semantic analysis; the existing copy, working on the existing representation, and a new copy, working on references to constant structures. Since the semantic analyzer consists of several tens of KLOCs, this would present an unacceptable double maintenance burden, and is thus not feasible from a software engineering point of view. However, the compiler passes after semantic analysis are a different story. Most of them already treat `from_ground_term_construct` scopes as the unification of a variable with a ground term, and only a few care what the term is. Therefore after semantic checks are complete, we enter the contents of all `from_ground_term_construct` scopes into the constant structure database, and replace the scope goal itself with a unification of the form `ScopeVar = const_struct_reference(...)`. This replacement does not need an extra pass over the module. We do it as part of the simplification pass, which is run after semantic analysis, and whose usual tasks are simplifying the program's internal representation (to make later passes faster), and generating warnings about code that is already too simple (such as if-then-else conditions that cannot succeed or cannot fail).

5.3 A third source of depth: field updates

Mercury allows programmers to give names to the fields of function symbols:

```
:- type date
    --->    date(
                date_year        :: int,
                date_month       :: int,
                date_day         :: int,
                date_hour        :: int,
                date_minute      :: int,
                date_second      :: int,
                date_microsecond :: int
            ).
```

Programs can use field names to get the value of named field, and to set it. If `D` is bound to the term `date(2012, 6, 29, 20, 10, 0, 0)`, then `D ^ date_day` will get the value of the named field, which in this case will be 29. The expression `D ^ date_day := 30` will, on the other hand, return a new term that is exactly the same as the term represented by `D` except for the value of the selected field being set to the given value; in this case, it will return `date(2012, 6, 30, 20, 10, 0, 0)`, Whenever the parser sees a field setter expression such as `D ^ date_day := 30`, it will replace it with a fresh variable (let us call it `NewD`), and it will add these two unifications into the program next to whatever goal that expression appeared in:

```
D =    date(Y, M, _D, H, M, S, U),
NewD = date(Y, M, 30, H, M, S, U)
```

This works well in all human-written Mercury programs we have seen. However, it can cause problems in machine written programs, because unlike humans, machines can easily generate types containing function symbols with large numbers of named fields.

The problem came to our attention via a program that defined a type like this:

```
:- type style
    --->    style(
                style_0000 :: int,
                style_0001 :: int,
                style_0002 :: int,
                style_0003 :: int,
                ...
                style_NNNN :: int
            ).
```

The value of N happened to be 64, but the compiler could just barely handle it, and the author of this code was asking us to make it possible to raise N to a substantially higher value.

The problem was a predicate that had four clauses for every one of those fields. Those clauses looked like these clauses for the field `style_0003`:

```
apply_prop(prop_0003(prop_type_a(N)), S) = S ^ style_0003 := N.
apply_prop(prop_0003(prop_type_b(N)), S) = S ^ style_0003 := N.
apply_prop(prop_0003(prop_type_c(N)), S) = S ^ style_0003 := N.
apply_prop(prop_0003(prop_type_d(N)), S) = S ^ style_0003 := N.
```

Besides the visible variables `N` and `S`, and the easily inferrable unnamed variables representing the first function argument (on the first line, the term `prop_003(prop_type_a(N))`), its subterm `prop_type_a(N)`, and the function result, each of those clauses also has 64 variables representing all the fields of `S`. Given that with the exception of the variables representing the function's

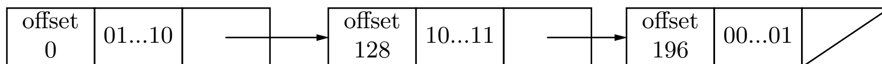

Fig. 7. A set represented using `sparse_bitset`

arguments and return value, all of the function's $4 * 64 = 256$ clauses have their own copies of all of these variables, the procedure actually has more than 17,000 variables.

This matters because the Mercury compiler has several passes that operate on sets of variables. The two main passes that do this are quantification and liveness. The quantification pass figures out the nonlocal set of each goal by computing the set of variables that occur inside the goal, the set of variables that occur outside the goal, and taking the intersection. The liveness pass figures out the points in the program where each variable is born and where it dies (the program points where that variable first and last needs storage, respectively), so that later the stack allocation pass can figure out which variables can be stored in the same stack slot.

Many of the sets the compiler operates on are relatively small. Some of these sets are sets of modules (e.g. the set of other modules imported by this module), sets of procedures (e.g. each clique in the procedure dependency graph is a set of procedures) and sets of types (e.g. the set of types that define a function symbol with a given name and arity), among others. Most of the other sets the compiler operates on are sets of variables.

Most of these sets contain only a few items, or a few tens of items. Most modules import at most two or three dozen other modules, most cliques contain only one or two procedures, most function symbols are defined by only one type, and most procedures have only a few dozen or a few hundred variables.

Since the typical set is small, the Mercury compiler has long represented sets as ordered lists without duplicates. Most operations on this representation take time that is proportional to the sizes of the set or sets involved. Since the representation is simple, the constant factors are low, which means that this representation has very good performance on very small sets (say 0-10 elements), though of course performance gets worse as sets gets bigger.

Unfortunately, some procedures like `apply_prop` have thousands or even tens of thousands of variables. When processing such procedures, many sets become similarly big, which reduces the performance of the operations on them below an acceptable level. We therefore had to look for set representations with better performance on large sets.

We tried to represent sets using balanced trees, specifically, 2-3-4 trees, but this was not a success. While this representation reduces the complexity of some operations, such as membership tests and single item insertion, from linear to logarithmic, it does not reduce the complexity of most others, such as union, intersection and difference. In fact, it can *decrease* performance on those operations, for two reasons. First, it can (and typically does) increase their constant factors. Second, some operations such as intersection are naturally best implemented using a synchronized traversal over both input operands. This can be easily implemented over sorted lists of items, but not over 2-3-4 trees.

As it happens, a typical invocation of the Mercury compiler performs more operations that balanced trees slow down than operations that they speed up. We therefore had to look for other representations.

One of the members of the Mercury team, Simon Taylor, designed a set representation that we have found to improve performance considerably. This representation is called sparse bitsets, and it exploits the fact that variables are represented by sequentially allocated positive integers. In a bitset, the presence of a variable in a set is indicated by a 1 bit in the position reserved for the variable, and its absence is indicated by a 0 bit. The sparse bitset representation represents a set using a linked list such as the one in figure 7. Each node in the list contains an initial offset, and a word sized bitmap (containing 32 bits on 32 bit machines, and 64 bits on 64 bit machines). Bit k in a node whose initial offset is o says whether the variable whose number is $o + k$ is in the set or not.

The initial offset must always be a multiple of the word size. This alignment requirement guarantees that two nodes from different sets will always cover either exactly the same set of

variables, or disjoint sets of variables. Binary set operations such as union, intersect and difference can thus traverse the two input lists together. If the current nodes in the two lists have the same initial offset, they can do their work on the node using the bitmap. (E.g. logical OR for set union, and logical AND for set intersection.) If the current nodes have different initial offsets, they can step over the node with the lower offset, after (in the case of union and the left operand of difference) remembering to include the stepped-over node in the result.

Another invariant of the sparse bitset representation is that nodes are not allowed to have bitmaps containing only zeros. If an operation such as intersection generates such a node, it must not include it in the resulting list. This can significantly reduce the amount of memory required to represent small sets. The nonexistence of a node in one input set to the set intersection and difference operations allow those operations to skip past irrelevant set elements in the other input set up to 32 or 64 times as fast as they could with an ordered set representation. Even though switching from the sorted list representation to the sparse bitset representation for sets of variables does not actually improve the big-O complexity of any of the set operations, that improvement in the constant factor can be so big that it can *feel* like it does.

6 Pragmatic issues

Most programmers' preferred method of discovering the cause of performance problems is profiling. We preferred to use this method too, in the form of Mercury deep profiler [1], whenever possible. Unfortunately, it wasn't always possible. Like other forms of profiling, Mercury's deep profiler adds instrumentation code to the program being profiled. If the program being profiled (in this case, the Mercury compiler) took more time to compile a program than the user's patience could stand, it would take even longer to execute with profiling; and if it originally took too much memory, it would take even more memory with profiling. And the profile is written out only when the program exits normally; if it is killed by an impatient user or developer, or crashes due to memory exhaustion, there will be no profile to analyze.

We therefore used another technique to discover some of the worst of the performance problems we discussed above. This technique was quite simple: run the compiler in the debugger, and turn on very verbose progress messages, which announce that the compiler is about to start executing a given compiler pass on a given procedure. When those messages stop coming for a few seconds, indicating that the compiler is spending more than that amount of time on a single pass on a single procedure, simply interrupt the compiler's execution. The Mercury debugger will stop the compiler at that point, and we can then see what the compiler is doing that is taking so long. You can then let the compiler continue for a second or two, take another look, let it continue, take another look, and repeat this a few times. Chances are that most or all of those looks will paint the same picture, and that picture will describe the actual performance problem.

The most direct way of seeing what the compiler doing at an interrupt point is simply to print the stack at that interrupt point. The stack is probably going to be huge, but the Mercury debugger had long been able to compress stack traces by detecting consecutive stack frames from the same procedure, and then printing the procedure's name and details together with a count. Unfortunately for us, some of the performance problems were caused by groups of two or more procedures that were mutually recursive but not self-recursive. If the compiler was stopped in one of these, a stack dump that went from the current execution point all the way to `main` could print one line for each of a *couple hundred thousand* stack frames, a process that could itself take a long time. You can of course interrupt the printing of the stack dump, or ask for the number lines to be printed by it to be limited to any number you choose, but the most important piece of information in such situations is knowing where that clique of mutually recursive procedures was entered in the first place. That will typically be much closer to the stack frame of `main` than to the current stack frame, which means that it will be beyond any reasonable limit on stack dump size.

The Mercury debugger originally did not have any facilities for finding out the entry point of a mutually recursive clique of procedures. We designed and implemented two such facilities specifically to support this work. One of these allows the programmer to identify the first stack

frame in the clique, allowing the user to retry that call, to print its its parameters, and/or (most important) to identify the call site it is called from. The other puts a user-defined limit on the number of lines displayed for a clique in a stack dump. This typically reduces the size of a stack dump to an acceptable level even when the program is consuming so much stack that it is about to run out.

Some programs that actually do exhaust stack space cannot be practically diagnosed using this technique, because their time interval between entering the recursion that causes stack exhaustion and the stack actually being exhausted is so short that human reaction times are too long to hit it. Fortunately, the Mercury debugger had long had a capability that makes finding such problems almost trivially easy. That capability is the `mindepth` command. A command such as `mindepth 10000` continues the execution of the program until a call reaches a stack depth of 10000. We haven't seen a single case where a procedure that *wasn't* about to exhaust the stack has reached such a depth.

Unfortunately, sometimes that stack exhaustion was not a real performance problem, but just a side effect of preparing the compiler for debugging. Some inputs to the compiler cause it to iterate over lists of many thousands or even millions of items. If the iteration is done by a tail recursive procedure, and each iteration is fast enough, this may not be a performance problem. However, in the current Mercury system, preparing a program for debugging destroys tail recursion. It is of course possible to support tail recursion even in programs that are being debugged, and we have a design for such a system, but tail recursion inherently works by reusing the caller's stack frame. In our case, we *want* the information in such stack frames; we most especially want the values of the variables in the topmost invocation in the (self or mutual) recursion.

Our solution to this problem was to modify the code of all loops in the Mercury compiler that did cheap per-item processing on lists that could contain many thousands of items. Their original structure was typically something like this:

```
process_items([], ...).
process_items([Item | Item], ...) :-
    process_item(Item, ...),
    process_items(Items, ...).
```

We changed these predicates to use a two level iteration, using code structured like this:

```
process_items(Items, ...) :-
    (
        Items = [],
        ...
    ;
        Items = [_ | _],
        process_upto_n_items(Items, 1000, LeftOverItems, ...),
        process_items(LeftOverItems, ...)
    ).

process_upto_n_items([], _, [], ...).
process_upto_n_items([Item | Item], ToDo, LeftOverItems, ...) :-
    ( if ToDo > 0 then
        process_item(Item, ...),
        process_upto_n_items(Items, ToDo - 1, LeftOverItems, ...)
    else
        LeftOverItems = [Item | Items]
    ).
```

Having the inner loop predicate return the leftover items to its caller collapses a thousand stack frames into one. If there are n items and the inner predicate returns after processing m of those, then the maximum number of stack frames we will need for the loop is approximately

n/m stack frames for the outer loop predicate and $n \ mod \ m$ for the inner loop predicate. To minimize the maximum number of the total stack frames, we picked values of m that were (very) approximately the square root of the largest value of n we considered feasible. We applied this treatment to predicates that iterated over (for example) the list of instructions generated for a predicate, the list of constant terms generated for a module, the list of characters in the module's string table, which is written out for use by the Mercury debugger if the program being compiled is being prepared for debugging, and several others.

Compiling a large program with debugging enabled posed another problem, which was that

- the Mercury declarative debugger needs access to a representation of the program being debugged, so the compiler puts such a representation into the generated code;
- to conserve space, we used 16 bit integers to identify variables in that representation;
- some large predicate had *more* than 2^{15} variables in their bodies, so those signed 16 bit integers overflowed.

The easiest fix would have been to switch to a uniform representation using 32 bit integers for variable numbers. (For the foreseeable future, computers will not have enough memory to contain the compiler's internal representation of any predicate that has more than 2^{31} variables, so 64 bits would be overkill.) However, this would waste space in the representation of almost all predicates. Therefore we switched to another system instead. We now divide procedures into four categories: those whose variable numbers fit in one byte, those whose variable numbers do not fit in one byte, but do fit in two, those whose variable numbers do not fit in two bytes, but do fit in four, and those whose variable numbers do not fit in four bytes. The compiler uses one, two and four bytes respectively to represent variable numbers for procedure in the first three categories; if it ever found a predicate in the fourth category, it would abort.

Some of the performance issues that we have identified by profiling the behavior of the compiler when given stress test inputs turned out to affect the behavior of the compiler on other inputs as well, just not enough to be easily noticeable. We found this out when we fixed the problem, and found that the performance of the compiler improved when given ordinary inputs as well.

For example, many places in the Mercury compiler test whether an item is in a set. If yes, they do something; if not, they insert the item into the set, and do something else. The performance problem is that if the membership test fails, then the code that does the insertion will redo the same traversal of the set representation that the membership test just did. Therefore if the test fails often enough, it is better to do a single traversal of the set that does both a membership test and an insertion (if the item is not in the set already). This traversal has a higher constant factor than a pure membership test does (due to the extra parameter it requires, the item to be inserted) or a pure insertion does (due to having to return an indication of the membership test's success or failure), but it is faster than two traversals.

7 Conclusion

Most papers review related work in the conclusion. In our case, however, there is almost no related work, so we discuss the reason for its absence instead :-(*Aside to reviewers: if you happen to know of any such work, please tell us about it; we would be very happy to compare our work to it.*

Traditionally, almost all Prolog predicates have been small: relatively few clauses, each containing relatively few goals. The exceptions were typically in programs that operated as deductive databases. These typically had many facts in dynamic predicates, and in some of them, the facts themselves contained large terms. However, large sizes in these two dimensions typically do not present any problems in program analysis for Prolog implementations, for the simple reason that since standard Prolog has no type, mode, determinism or uniqueness system, most of them do not do any semantic checks or other program analyses that are in any way comparable to what the Mercury compiler does. There is one Prolog compiler we know of that does do significant amounts of analysis on the programs it processes, Ciao Prolog, but we are not aware of any algorithms it employs specifically to handle large programs and/or predicates.

Some Prolog systems *have* long used some techniques that can help generate better code for large predicates. For example, Eclipse Prolog could compile ground terms to static data structures (as opposed to WAM instructions to build those data structures dynamically) since at least 1990. Unfortunately, most of these code generation techniques are too simple to be worth a paper. The one exception we know of is the paper on exocompilation in the WAM and in Mercury [3].

We don't know of any logic programming languages other than Prolog and Mercury that have a large enough set of programs written for them for compilation of very large programs to become a concern.

We have not heard of any work on compiling large functions in functional languages. We don't know whether that is because large functions don't exist in practice, exist but do not pose any particular problems for compilers, do pose problems but these have not yet been solved, have been solved but not described, or been described and we have just not found the descriptions :-(

There is certainly reason to believe that functional languages would have fewer problems with the compilation of large functions than Mercury: functional languages such as Lisp, ML, and Haskell do not need all the semantic checks that Mercury needs. Haskell and ML have type systems very similar to Mercury's, but they have no mode system that requires any analysis (arguments are always input, results are always output), they have no determinism system that requires any significant analysis (functions can never have more than one solution, and algorithms that find uncovered cases can be both simple and fast), and (with the exception of Clean) they have no uniqueness system. In addition, for Lisp and ML, the scope of useful program analyses is limited by the fact that those languages allow destructive update.

Whatever the situation with functional languages may be, large predicates *do* pose a challenge to the Mercury compiler. Before we started this work about six years ago, it often lost that challenge. However, as we implemented the techniques described in this paper one after another, such losses became rarer and rarer, and today, the Mercury compiler has good performance even when it is given programs it could previously not compile. Only a small fraction of this improvement in speed came from faster hardware; almost all of it came from improved algorithms. This paper has described some of the most important of those improvement. These improvements, and all the others, ara available in the current Mercury releases-of-the-day, which are available for download from http://mercurylang.org.

We would like to thank the people who contributed the large Mercury programs that both provided the motivation for this research and acted as benchmarks for its evaluation: Doug Auclair of Logical Types, Michael Day of YesLogic, Peter Ross and Peter Wang of Mission Critical, and everybody in the G12 project.

References

1. Thomas Conway and Zoltan Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, July 2001.
2. Bart Demoen, Maria Garcia de la Banda, and Peter J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228, Jaunary 1999.
3. Bart Demoen, Phuong-Lan Nguyen, Vitor Santos Costa, and Zoltan Somogyi. Dealing with large predicates: exo-compilation in the WAM and in Mercury. In *Proceedings of Seventh International Colloqium on Implementation of Constraint and Logic Programming Systems*, pages 117–131, Porto, Portugal, September 2007.
4. Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run time type information in Mercury. In *Proceedings of the 1999 International Conference on the Principles and Practice of Declarative Programming*, pages 224–243, Paris, France, September 1999.
5. John Lloyd. *Foundations of logic programing.* Springer, 1987.
6. Nancy Mazur. *Compile-time garbage collection for the declarative language Mercury.* PhD thesis, Catholic University of Leuven, 2004.
7. David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of Mercury. In *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pages 109–120, Pittsburgh, Pennsylvania, October 2002.

8. Wikipedia. Thrashing (computer science).