The University of Melbourne

Department of Computer Science

Optimization of Mercury programs

Simon Taylor
Supervisor: Dr. Zoltan Somogyi

November 1998

Abstract

The pure logic programming language Mercury is very well suited to the development of sophisticated program transformations due to its clean semantics and strong type, mode and determinism systems. We describe the design and implementation of some of the high-level optimization passes of the Mercury compiler, including deforestation, constraint propagation and type specialization. Mercury's mode system allows programmers to specify when a data structure can have only one reference. We present the implementation of an optimization pass to exploit this information to reuse dead cells or insert calls to place them on a free list.

Contents

1 Introduction

The aim of the Mercury project at the University of Melbourne is to produce an implementation of a logic programming language suitable for real world software development projects. The main benefit of Mercury over the imperative programming languages in common use today is stronger compile-time checking. The use of type, mode and determinism declarations allows the compiler to catch many programming errors. The declarations also aid the compiler in generating very efficient code. The other benefit is increased expressiveness – a Mercury program is closer to a declarative specification of the problem than the equivalent imperative program.

As in functional languages, Mercury encourages programmers to build their programs up by composing primitives such as *append*, *map* and *foldl*. This introduces two efficiency problems. First, there is some overhead when using polymorphic unifications or higher-order calls instead of type-specific unifications or direct calls. The Mercury compiler includes an optimization pass to remove this overhead by specializing predicates for particular higher-order arguments or argument types. Second, using a composition of simple, general predicates rather than one specialized predicate results in extra repeated passes over the same data structure and the production of data structures which are used only to store intermediate values. We present a *deforestation* algorithm which in many cases can remove these intermediate data structures, giving a result as efficient as if the programmer had specialized the composition by hand.

Our deforestation algorithm is expressed as an unfold/fold transformation [30]. Many other logic programming transformations can be expressed in this style. For example, constraint propagation transforms a program to reduce the search space by evaluating goals which could fail as early as possible. We give a simple extension of our deforestation algorithm to implement constraint propagation.

Only allowing variables to be assigned once may make the language semantics simple, but it can have a negative impact on performance. It is common in Mercury programs for a data structure to be created, used and then discarded when a slightly different version is produced. If deforestation cannot remove the construction of the old data structure completely, it may be possible to optimize construction of the new structure by reusing the memory of the old structure. If the compiler knows that all references to the old structure are dead, it can reuse the old structure for the new one by destructively updating the fields of the old cell. If a dead cell cannot be reused it may be possible to add it to a free list to be reused later. We give an account of the implementation of these optimizations as a pass within the Mercury compiler.

We assume some familiarity with the syntax and semantics of Mercury programs. For a full description see the Mercury Language Reference Manual [33].

Section 2 gives an introduction to the annotation passes of the Mercury compiler. Section 3 presents the high-level optimization passes that were present before work on this thesis began, and some extensions which were made to them. Section 4 describes the implementation of the deforestation pass of the Mercury compiler. Section 5 extends the deforestation pass to also handle constraint propagation. Section 6 describes the implementation of structure reuse and compile-time garbage collection.

2 Mercury

In this section we describe the compiler-generated annotations most useful for performing high-level transformations. Since Mercury requires the programmer to provide mode, type and determinism declarations where they can't be automatically inferred, the compiler is guaranteed to have perfect data flow information when optimizing. This is a vast improvement on the situation for Prolog, where a compiler has to work hard to infer types, and where mode inference results in unusable compiler run times.

The internal annotated source-level representation of Mercury code is called the high level data structure (HLDS). The HLDS for the well-known append predicate is given in appendix A.

2.1 Super-homogeneous form

Mercury predicates are transformed internally into what is called super-homogeneous form. The main restriction of super-homogeneous form is that arguments of predicate calls and constructors must be distinct variables, so that unifications implied by multiple occurrences of a variable in a term are made explicit. This greatly simplifies the analysis passes of the compiler.

2.2 Quantification

The quantification pass works out which variables are non-local to a goal. A variable is a non-local of a goal if it occurs both inside and outside the goal. Variables are implicitly quantified at their smallest enclosing scope. This is useful for mode analysis, for example in checking that the non-locals of a negation are not further instantiated by the negation. The quantification pass may be rerun by optimization passes which alter the structure of a goal in a way which might change the scopes of variables.

2.3 Type checking

The main job of the type-checking pass of the compiler is to compute the type of each variable in the program. Mercury has a polymorphic type system similar to that proposed by Mycroft and O'Keefe [25] extended with Haskell-style type-classes [12].

The basic types visible to the user are int (machine word size integer), float (C `float` or `double`, possibly boxed if larger than a machine word), string (equivalent to C's `char *`), character (8 bit character) and c_pointer (pointer to a C data structure, used for types such as destructively updated arrays). New types can be built discriminated union. New names for types can be added using structural equivalence.

Parametric polymorphism allows one predicate to work on data of different types. Type classes allow a predicate's declaration to specify that only variables whose type is an instance of a certain class can be passed where a polymorphic argument is expected. A type class provides an interface which allows type specific predicates (*class methods*) to be called from within a polymorphic predicate. For example, a generic sorting routine might only accept a variable whose type is an instance of a 'comparable' type class, for which the only method provided is a comparison function.

2.4 Mode analysis

Mercury's strong mode system ensures that all variables have a predictable instantiation state at every point in the program. This catches errors such as binding a variable in one branch of a disjunction but not others. Mercury's mode system is distinguished by the presence of unique modes, which allow the programmer to specify that there is exactly one reference to a data structure, preventing errors such as using the wrong version of a single-threaded data structure, ensuring destructive updates are not backtracked over and giving the compiler enough information to implement the structure reuse optimization described in section 6.

The mode analyser in the Mercury compiler is loosely based on [9]. A more up-to-date presentation of the Mercury mode system will be given in [2, 10], but we provide here an introduction to the parts of the mode system relevant to the optimization passes.

An instantiation tree (or *inst*) is an assignment of an instantiatedness (e.g free or ground) to each node of the type tree of a variable. Insts must be able to be recursive to map onto recursive type structures. A mode is a mapping from an initial to a final inst. For example, the mode *in* is equivalent to 'ground->ground'. Each predicate has one or mode declarations, which give the modes for each argument. The initial insts of the argument modes of a predicate mode declaration promise that the arguments of any call to that mode of the predicate will be at least as bound as the initial insts given.

At each call in a predicate's body, the compiler proves that the insts of the arguments are at least as bound as the initial insts of the called predicate's declaration. Mode analysis can delay a goal until its arguments are bound enough to allow evaluation of it. It is a mode error if the arguments never become bound enough.

The compiler proves that for the given initial insts for a predicate, if the predicate succeeds the final inst of each argument variable is compatible with that given in the declaration. The final insts do not have to be exactly the same. For example, a variable may have final inst 'clobbered' in a mode declaration even though no goal in the predicate destructively updates it. The structure reuse optimization in section 6 takes advantage of this by reusing or freeing memory cells for this type of variable.

In the Mercury compiler, the alternatives in the definition of the inst representation are as follows:

free(<aliasing>) Completely unbound. The aliasing field describes whether there are any aliases for the variable. At runtime, aliased variables are represented by a reference to their location. Non-aliased variables are represented by uninitialized junk. Since the Mercury mode system allows only definite aliasing the compiler must know all of the aliases.

ground(<uniqueness>) Completely ground (contains no variables). See below for a description of the uniqueness. The HLDS pretty printer outputs ground insts as just the uniqueness, with 'shared' written as 'ground'.

bound(<uniqueness>, <functor>, <args>) Bound to <functor> with argument insts <args>. The <uniqueness> describes only the top level cell. The pretty printer outputs bound insts as <uniqueness>(<cons_id>(<args>)), where 'shared' is written as 'bound'.

any(<uniqueness>) Used to describe the inst of constrained variables. *any* insts will not be discussed any further in this thesis.

not_reached Describes the inst of an output of a goal which cannot succeed.

defined_inst(inst_name) A defined inst is either a user-defined inst, or a compiler generated inst used to cache the result of abstractly unifying or merging two insts during mode analysis. Defined insts may be recursive.

alias(inst_key) An aliased term. The inst of the variable is that currently held by the given inst_key. An inst_key is a small integer which is mapped to an actual inst by an inst_key_table for the procedure and information about the current instantiation state at a point in the program. 'alias' insts allow the compiler to represent definite sharing between terms – inst nodes with the same inst_keys refer to the same run-time term.

The mode analyser in Mercury 0.8 [37] does not handle 'alias' insts. This disallows the use of partially instantiated modes and nested uniqueness. This restriction has been lifted in a development version of the compiler, and will hopefully be included in the next public release. The discussion of the implementation of structure reuse in Section 6 refers to the development version of the compiler which handles aliasing properly, since the accuracy of the mode checker is crucial to the effectiveness of structure reuse.

A *uniqueness* describes the number of references to a variable, and can be one of the following:

ground/bound The compiler doesn't know how many references there are.

unique The compiler knows all of the references, and there are no references on backtracking.

mostly_unique The compiler knows all of the references, and the cell is live on backtracking. Updates must be trailed so that they can be undone when backtracking occurs.

clobbered The data contained by the variable has been destructively updated, so there must be no references to the old structure.

mostly_clobbered The data contained by the variable has been destructively updated so there must be no references to the old structure, except on backtracking. The update must have been trailed.

Giving mode annotations for each argument of a predicate is important for documentation and for error checking. The Mercury compiler's mode analyser checks the mode annotations provided by the programmer by a process of abstract interpretation of the goal for each predicate. It is also usually able to infer modes for non-exported predicates for which declarations are not provided.

Mode analysis performs several important annotations of the code for each predicate:

- Determines which procedure applies for each predicate call. A procedure is a single mode of a predicate.
- Categorizes unifications into five categories

  assignment written X := Y in the examples. X is initially free, and takes the inst of Y after the unification.

  simple test written X == Y. X and Y must be of atomic types and must both be ground before the unification.

  construction written X := f(A, B, ...). X must be free before the unification, and takes the inst 'bound(unique, [instof(A), instof(B), ...)' afterwards, where instof(A) is the current inst of A in the instmap. Construction unifications are the only goals which cause memory to be allocated (other than hand-written C code).

  deconstruction written X ?= f(A, B, ...) if the unification is semideterministic (if X is not guaranteed to have functor 'f'), or X => f(A, B, ...) if it is deterministic. X must be bound to some functor before the unification. The argument terms of both sides of a deconstruction unification may be further instantiated.

  complicated unification written X = Y. X and Y may have any inst. A complicated unification is implemented as a call to an automatically generated out-of-line procedure for the particular type and input insts.

- Computes the determinism of unifications.
- Converts function calls into predicate calls.

A separate *unique mode analysis* pass run after determinism analysis checks that destructive updates are not backtracked over. This pass must be run after determinism analysis because it needs to know which goals are non-deterministic and could cause backtracking.

When processing a goal, the current instantiation state is stored in a structure called an *instmap*. An instmap is either *unreachable* or *reachable*. An unreachable instmap means that earlier goals in a conjunction could not succeed. If an instmap is reachable, it contains a mapping from variables to insts, and an inst_key substitution. The variable mapping contains the current instantiatedness of every variable visible in the current scope. The inst_key substitution is a mapping from inst_key to inst_key representing the current inst of cells which have had their inst changed by an operation on one of their aliases.

For example, in the HLDS for *append/3* in appendix A, the recursive call clobbers the tail of the first argument (because of the *di* or destructive input mode). The inst of all aliases of the tail of the first argument must be set to clobbered. Before the call, the inst_key representing the tail of the first argument is IK_21. A new inst_key for the cell is introduced with inst clobbered (IK_23), and the inst_key substitution records that the old inst_key maps to the new one.

Using the initial instmap for a procedure from the mode declaration for the procedure and the instmap_deltas for the sub-goals of the predicate, it is possible to compute an instmap representing the current instantiation state at any point in a goal. To interpret the data in an instmap, the compiler also needs an *inst_table*, which contains all the information about user- and compiler-defined insts. A sub-field of the inst_table, the *inst_key_table*, stores the inst corresponding to each inst_key used in the current procedure.

Mode analysis attaches mode information to every sub-goal of each procedure in an *instmap_delta* stored in the goal's *hlds_goal_info*. An instmap_delta has the same structure as an instmap and is applied to an instmap by overlaying the entries in the instmap_delta over the top of those in the instmap. An instmap_delta makes little sense on its own – the inst_key substitution from the instmap before the goal is needed to be able to completely compute the new inst of a variable. An instmap_delta on its own also does not contain enough information to work out which variables are bound by a goal, since it does not contain information about variables not occurring in the goal whose insts contain inst_keys which are updated by the goal.

2.5 Switch detection

One very common type of disjunction in logic programs is described in the Mercury compiler as a *switch* [11] due to its similarity with the C construct. A disjunction is a switch if at the start of each disjunct there is a deconstruction unification between a variable bound before the switch and a constructor of a discriminated union type. This is a generalisation of the usual Prolog indexing scheme, the main difference being that the terms being indexed on do not need to appear in the head of a clause, and multiple levels of constructors in a term may be indexed.

Switches are useful because at most one of the branches can match, so a choice point is not needed. Detecting switches is vital to the accuracy of the determinism analysis pass of the Mercury compiler. The detection of switches aids in maintenance of large programs, because code can be written so that the compiler will give errors if an alternative is added to a discriminated union type and deterministic switches on elements of that type are not updated. This has been very effective during the development of the Mercury compiler.

Switches are also very useful for optimization, since if a program analysis pass can discover the constructor to which a *switched-on* variable is bound, the other branches can be pruned away, performing the functor test at compile time. Binding information such as this is provided by mode analysis, in the form of *bound/2* insts.

The compiler can generate very efficient code for switches. There are several alternative code generation methods – lookup tables if the outputs of the switch are constants, jump tables for non-sparse switches on integers or enumerations, hash tables for switches on strings and a choice of several different algorithms for switches on discriminated union types including chains of if-then-elses, jump tables and binary search.

2.6 Determinism analysis

Using the binding information provided by mode analysis and the disjunction indexing information provided by switch detection, the determinism analysis pass of the Mercury compiler [11] is capable of proving the large majority of predicates to be deterministic if the programmer follows some relatively simple coding rules.

The determinism of a goal is computed according to the number of solutions it could produce and whether it can fail before producing a solution. These properties are derived from the determinism annotations placed on unifications by mode analysis and predicate determinism declarations. As with mode analysis and type checking, determinism analysis can infer declarations for non-exported predicates.

Experience has shown that most programs spend the the majority of their time in predicates which cannot succeed more than once, so much of the optimization effort should go towards this type of code. With the addition of the Aditi deductive database interface [15], this will be less true. Aditi queries are written as non-deterministic Mercury predicates.

2.7 Polymorphism

Where a predicate has a polymorphic argument, the compiler must be able to handle operations on data of any type including unification, comparison, copying (for the accurate garbage collector) and deconstruction (using *functor/3*). For polymorphic arguments constrained by a type class [12], there will be other operations supported on the type. The polymorphic predicate needs some way of calling the type-specific operation for a given variable. The Mercury compiler implements this by performing a source-to-source transformation on the program to introduce a *type_info* argument for each type variable and a *typeclass_info* argument for each class constraint in the predicate's type declaration.

A type_info variable contains the addresses of the unification and comparison routines for the type as well as pointers to tables describing the layout of the type, including the names and argument types of the data constructors. This information is used to implement library predicates such as *io__write/3* to print a term of any type and *functor/3* to get the name of the constructor of a term.

A typeclass_info variable contains the addresses of the methods of the class as well as pointers to the type_infos of the constrained variables and pointers to the typeclass_info variables for any constraints on the instance declaration.

3 High-level optimization passes

Prior to the addition of the optimizations described later, the Mercury compiler already performed some significant source-to-source transformations.

3.1 Simplification

The simplification pass of the Mercury compiler performs several functions:

- Common structure elimination – replacement of constructions and deconstructions of terms which have been seen earlier in the goal with assignments to the arguments. This process pays no attention to maintaining unique mode correctness, and may introduce aliases to terms which are meant to be unique, which was acceptable before structure reuse (see section 6) was implemented.

- Excess assignment elimination – removal of assignments to local variables to a conjunction by substituting away the local variable throughout the conjunction.

- Duplicate call elimination – removal of multiple calls to the same predicate with the same input arguments.

- Evaluation of builtins where all the input arguments have known values.
- Pruning of branches which cannot succeed from switches and disjunctions, replacing deterministic goals with no outputs with 'true' and replacing goals with determinism failure with 'fail' where the termination behaviour of the program is unchanged.
- Flattening of nested conjunctions.
- Tweaking the goal to enforce invariants required by the code generator. For example, a goal which cannot succeed must have an unreachable instmap_delta.

As we shall see in section 4, these optimizations can be used as the basis of more powerful program transformations.

3.2 Inlining

The Mercury compiler automatically inlines small predicates and those marked with '`:- pragma(inline...)`' declarations. The inlining pass performs a bottom-up traversal of the module's dependency graph, inlining calls to small procedures or procedures which are called from one place only. The inlining pass tries to avoid producing procedures with large numbers of variables, since that can cause slow compilation due to $O(n^2)$ algorithms in the back end of the compiler.

3.3 Higher-order and type specialization

Higher-order programming aids code reuse, by allowing one piece of code to perform many similar functions. The price for this genericity is inefficiency. Calling higher-order terms is more expensive than a normal call because an arbitrary number of curried arguments must be unpacked from the closure and placed in registers. The higher-order specialization pass of the Mercury compiler removes this overhead by creating specialized versions of predicates with higher-order arguments where the value of those arguments is known, by a process of constant propagation.

Polymorphism and type classes [12] are special cases of higher-order programming where the compiler introduces the higher-order arguments automatically as described in section 2.7. In some cases the additional arguments have significant overhead. For a simple predicate for which the type_info arguments are not actually needed, such as naive reverse, the passing of type_infos has an overhead of roughly 15%. For procedures which perform unification or comparison of polymorphic variables the overhead is much worse – polymorphically unifying a list of integers takes over six times as long as the equivalent monomorphic operation. The difference between the two versions is that in the monomorphic version the equality test on the elements of the list is an inline integer comparison, while the polymorphic version must do a higher-order predicate call to the comparison predicate.

An alternative implementation of polymorphism might create a version of each polymorphic predicate for each instantiation of type variables which could occur at runtime. This removes the overhead of passing extra arguments and calling closures, but is quite wasteful of space. The extra code size could decrease performance through increased cache misses. What is needed is a hybrid approach, producing specialized versions of heavily used predicates while using the more compact but less efficient representation for others.

The higher-order specialization pass of the Mercury compiler has been extended to propagate type_info and typeclass_info constants, and to evaluate at compile-time the builtin predicates which manipulate them, so that the runtime cost of polymorphic code can be completely removed. Type specialization is most effective for builtin types such as int and float, where the comparison and unification routines are simple enough to be inlined. Specialization with respect to types such as

Benchmark	Unoptimized (sec)	Optimized (sec)
unify_list	32	5
insert	47	42

Table 1: Benchmark results for type specialization

list(T) is likely to be less useful, since the comparison and unification procedures will still need to perform a higher-order call to call the comparison or unification predicate for type T.

The inter-module optimization capabilities described in section 3.6 do not handle top–down optimizations such as type specialization very well, resulting in duplicate copies for different specializations. The problem is less severe for higher-order specialization because it is rarer that the same specialization appears in multiple modules – most higher-order terms refer to predicates local to a module. With the aid of profiling information, the compiler could work out which type specializations are likely to be worthwhile. In the meantime, we allow the programmer to give a pragma declaration to make the compiler produce specific specializations. For example, the following declaration would request the specialization of *map__lookup/3* for integer keys.

```
:- pred map__lookup(map(K, V), K, V) is det.
:- mode map__lookup(in, in, out) is det.
:- pragma type_spec(map__lookup/3, [K - int]).
```

The compiler implements this by adding a new predicate 'TypeSpecOf__map__lookup_3' which has the type signature of map__lookup except that type 'int' is substituted for all values of the type variable K. The body of the new predicate is just a call to the unspecialized version of map__lookup. The presence of extra type information about the arguments to this call forces the higher-order and type specialization pass to create the specialized version of map__lookup.

The presence of this specialized version is communicated to importing modules through the automatically generated module interface files. A slightly different version of the pragma is used which specifies the name of the predicate introduced to force specialization.

```
:- pragma type_spec(map__lookup/3, [K - int], 'TypeSpecOf__map__lookup_3').
```

This pragma is handled exactly the same way in importing modules as in module 'map' – the new predicate is added to force creation of a specialized interface to map__lookup. The difference here is that the importing module does not have the code for map__lookup, so it just creates a specialized interface with no body. The specialization code is careful to ensure that the names of the specialized version are the same in both module 'map' and the importing module.

To give an idea of how much type specialization can improve performance, we have benchmarked the specialization on a couple of small programs. The results are shown in table 1. The benchmarks were performed on a 2 CPU 200MHz Pentium Pro with 256Mb of RAM running Linux (kernel 2.1.102). The benchmarks were compiled with optimization level -O5. 'unify_list' unifies two lists of 10,000 ones 10,000 times. 'insert' inserts 2,000 random elements into a binary tree 1,000 times. There is less speed-up for insert because it does more computation, including a lot of memory allocation.

Specifying certain commonly used specialized predicate types such as map__lookup for maps with integer keys and set__insert and set__union for operations on sets of integers gave only a 2–3% speed-up for the compiler. Given the speed-up for binary trees this low speed-up is not surprising. The code size increase is only 70Kb on top of the initial size of 4Mb, an insignificant increase.

Type specialization was implemented for this Honours thesis. The implementation required a substantial rewrite of the existing higher-order specialization pass, which is now 1800 lines of Mercury code, as well as significant changes to other parts of the compiler to implement the pragma. Type specialization is included in the Mercury 0.8 release [37]. The changes to implement the user-guided type specialization pragma were completed during the code freeze for the release, and are contained in the workspace /home/mercury1/stayl/tspec/mercury.

3.4 Unused argument elimination

In many cases the type_info arguments added by the polymorphism pass are completely unnecessary, since the called predicate does not unify or compare values of the polymorphic type. In these cases, it is not necessary to type-specialize the predicate, creating a duplicate version, since the type_info arguments can just be removed. Rather than change the polymorphism pass to analyse predicates to find out which need their type_infos, the Mercury compiler has a pass to perform general removal of unused input arguments. The pass builds up a graph to represent which variables depend on which other variables, then work out which are needed by a process of fixpoint iteration. The presence of the unused argument removal pass also simplifies other passes, such as higher-order specialization, since they do not need to be concerned with removing arguments for which the uses have been specialized away.

3.5 Last call optimization

It is common in Mercury programs for a recursive call to be followed only by construction unifications which fill in the fields of cells with unaltered outputs of the recursive call – this is true of append as shown in appendix A. Unfortunately, the program must store the head of the list on the stack before the recursive call and retrieve it after the call to put in the output. The reason why this is not tail recursive as it is in Prolog is that the third argument of the recursive call must be free, not bound to the second field of the cons cell of the result. The last call optimization solves this problem by creating a new version of the predicate which expects the address of the location in memory to place the argument, rather than returning the argument in a register and creating the cell after the recursive call. The old version is left in place to provide an interface to the different calling convention. This optimization is described in more detail in [32]. The effect on append is shown in figure 1.

The deforestation and constraint propagation algorithms described in the next two sections do not need to deal with the output of this transformation, since they are run before it. The structure reuse optimization described in section 6 is run after last call optimization because structure reuse introduces annotations which would be difficult for other passes to maintain.

3.6 Inter-module optimization

Several of the optimizations described above can be applied across module boundaries. This is done by writing the code for predicates which are judged to be candidates for optimization in importing modules to a special interface file with suffix '.opt'. A predicate is written if it is small enough to be inlineable, has a higher-order argument which can be specialized away or is a candidate for deforestation (see section 4). Any declarations which are needed for the compiler to analyse the newly exported code must also be written to the '.opt' file. Information about unused arguments for exported predicates can be written to the '.opt' file if the '–intermod-unused-args' option is given to the compiler. '.opt' files are read in when processing importing modules as if they were part of the source code for that module.

Currently information can be propagated from imported modules to importing modules only – there is no way to request transformations on code in imported modules. Also, the optimizations are not transitive – inlined code can only cross one level of importation. An exception is that transitive propagation of termination analysis information is supported.

```
append(As, Bs, Cs) :-
    (
        As ?= [],
        Cs := Bs
    ;
        As ?= [A | As1],
        Cs := [A | Cs1],
        % The address of Cs1 is passed to append_tr.
        % ‘&’ represents the address operator as in C.
        append_tr(As1, Bs, &Cs1)
    ).

    % The tail recursive version of append takes a reference
    % to the output (free(aliased)) which it fills in.
    % The programmer cannot write this because of the
    % free(aliased) inst, which is internal to the compiler.
:- pred append_tr(list(int)::di, list(int)::di,
        list(int)::(free(aliased)->unique)) is det

append_tr(As, Bs, Cs) :-
    (
        As ?= [],
        % Fill in the reference.
        % ‘*’ represents the deference operator as in C.
        *Cs := Bs
    ;
        As ?= [A | As1],
        *Cs := [A | Cs1],
        append_tr(As1, Bs, &Cs1)
    ).
```

Figure 1: append after last call optimization

4 Deforestation

4.1 Introduction

Logic programming languages have much in common with functional languages in terms of semantics, so it is natural that optimizations designed for functional languages can also be applied to logic languages. One example of this is deforestation, proposed by Wadler [38] for a lazy functional language. The aim of deforestation is to optimize programs which construct unnecessary intermediate data structures or perform multiple traversals of the same data structure.

Several researchers have produced transformation techniques capable of performing deforestation for logic programming languages [13, 30]. The main problem to be solved for these techniques is to find heuristics to guide the transformation process to perform a useful amount of the possible optimization without causing code explosion. Here we present the details of an implementation of deforestation as an optimization pass within the Mercury compiler, which uses the type, mode and determinism information provided by the compiler to detect opportunities for the optimization.

4.2 Simple partial evaluation

A very simple version of partial evaluation can by implemented in the Mercury compiler using the notion of a switch. Each predicate is analysed to determine if the top level of the goal contains a switch, and which of the arguments of the predicate are switched on. If at any call to a predicate, the top level functor of a switched-on argument is known, the call is *unfolded*. Unfolding involves replacing a call with the body of the called predicate, renaming apart the local variables of the body. Mode analysis is then run on the unfolded goal to push through the extra binding information from the call. The simplification pass described in section 3.1 can then be run to prune away the failing branches, remove unnecessary constructions and evaluate builtins where possible. This corresponds to the determinate unfolding rules suggested by many partial deduction researchers [13, 28, 31].

Some effort has to be made to ensure that the unfolding process terminates when unfolding recursive predicates. The aim of the termination check is to allow the specialization process to continue on recursive calls while the amount of information to use monotonically decreases, for example by consuming an input list.

When a call is unfolded, information about the call is added to a stack of ancestor calls. When the processing of the unfolded goal is finished, the call is popped off the stack – an unfolded call only restricts unfolding of calls within its unfolded goal. Each element of the call stack must be smaller than the element below according to some ordering. It must be impossible for there to be an infinite sequence of calls unfolded which satisfy the ordering. There are many possible orderings which could be used. Researchers have suggested well-founded orderings on called atoms [23], homeomorphic embedding of atoms [19] and methods involving the specialization behaviour of the program [20].

Since our implementation only propagates simple binding constraints, a simple termination check is sufficient. Our implementation uses an ordering based on the sum of the sizes of the insts of the arguments to a call. A call is considered smaller than another call if it is a call to a different predicate, or if the sum of the sizes of the argument insts are smaller. The size of an inst measures the number of constructors occurring within the inst after expanding any defined insts until a loop in the expansion is reached (defined insts can be recursive) or until the inst is no longer a defined inst. A 'ground' inst counts as zero size. This is a reasonable approximation to the amount of information for specialization contained in the inst since consuming the first element of a list reduces the inst's size by one.

To handle the case where some argument sizes increase, for example for accumulating parameters, we use a method suggested in [23]. The ordering does not always sum the sizes of all arguments to a called predicate. For the first occurrence of a call to a predicate recorded on the call stack, all arguments are used. If the sum of the argument inst sizes does not decrease for a recursive call being considered for unfolding, the increasing arguments are removed from the sum for this and subsequent calls to the predicate, leaving those which decreased or stayed the same size. If there were any decreasing arguments, the unfolding process can continue. Since insts must be of finite size, the algorithm must eventually run out of decreasing arguments and stop unfolding.

To avoid unnecessary code bloat, the compiler should also check that the unfolding of the call will probably result in a performance improvement. Removing a branch but in the process duplicating a page of code is not likely to be very productive. The compiler computes a rough estimate of the cost of a goal, and undoes any unfolding where the estimated performance improvement is less that some proportion of the code size increase.

On its own, this process is not very interesting, since the binding constraints it propagates are quite limited. More sophisticated partial evaluation systems propagate the collected substitution through the goal to take advantage of equality constraints, and in some cases, inequality constraints. The limited form of partial evaluation presented here is sufficient for deforestation.

4.3 Deforestation algorithm

More interesting transformations can be achieved by using the branching information of the predicates called in a conjunction to obtain a specialized version of the conjunction. In some cases there is extra information in one of the branches of one called predicate which can be used to determine which branch will be taken in a call to a later predicate in the conjunction. A good example of this is *double_append/4* (see figure 2).

The body of append is a switch on the first argument. This means that knowing the functor of the first argument to append is enough to decide which branch is taken.

The annotations at the end of each disjunct show the change in instantiatedness of the non-local variables. In the instmap at the end of the second disjunct of append there is enough information to determine which constructor the last argument is bound to. This information is lost when the information from each branch is merged to give the instmap after the switch. The aim of the deforestation process is to transform the conjunction so that the second call is in a position to use the extra information available in the branches of the first.

After choosing a conjunction to optimize where both the first and last goals are calls, the optimization process is as follows:

1. Create a new predicate for the specialized conjunction, in this case *da/4*.
2. Unfold the first call.
3. Push the second call into the branches of the first.
4. The partial evaluation process described above can now unfold the second call where there is enough information to choose one of the branches.
5. Run the simplification pass of the compiler on the goal. The intermediate data structure created by the original predicate can then be removed by common structure elimination.
6. In the recursive branch, there is now an instance of the original conjunction, which can be replaced with a call to *da/4*. This gives a recursive definition for *da/4*, avoiding the creation of the intermediate structure. This replacement is referred to as *folding*.

This algorithm performs only determinate unfolding (step 4), except when it unfolds the first call in an attempt to deforest a conjunction (step 2). Non-determinate unfolding is only applied when it is likely to lead to deforestation. This is important, since poorly controlled application of non-determinate unfolding can lead to code explosion.

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

double_append(As, Bs, Cs, Ds) :-
    append(As, Bs, Temp),
    append(Temp, Cs, Ds).

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

append(As, Bs, Cs) :-
    (
        As ?= [],
        Cs := Bs
        % Cs -> ground
    ;
        As ?= [A | As1],
        append(As1, Bs, Cs1),
        Cs := [A | Cs1]
        % Cs -> bound('.', [ground, ground])
    ).
    % Cs -> ground
```

2.1: double_append before optimization

```
double_append(As, Bs, Cs, Ds) :-
    da(As, Bs, Cs, Ds).

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

da(As, Bs, Cs, Ds) :-
    (
        As ?= [],
        append(Bs, Cs, Ds)
    ;
        As ?= [A | As1],
        append(As1, Bs, Ts1),
        Ts := [A | Ts1],
        Ts => [A1 | Ts2],
        append(Ts2, Cs, Ds1),
        Ds := [A | Ds1]
    ).
```

2.2: double_append after unfolding and branch pruning

```
da(As, Bs, Cs, Ds) :-
    (
        As ?= [],
        append(Bs, Cs, Ds)
    ;
        As ?= [A | As1],
        append(As1, Bs, Ts1),
        append(Ts1, Cs, Ds1),
        Ds := [A | Ds1]
    ).
```

2.3: double_append after common structure elimination

```
da(As, Bs, Cs, Ds) :-
    (
        As ?= [],
        append(Bs, Cs, Ds)
    ;
        As ?= [A | As1],
        da(As1, Bs, Cs, Ds1),
        Ds := [A | Ds1]
     )
```

2.4: double_append after folding

Figure 2: Optimization of double_append

The original program performed two traversals over the first input list, one for each call to *append/3*. The optimized program performs only one, and also avoids creating an unnecessary copy.

4.3.1 Argument annotation

An annotation pass run before deforestation finds predicates for which the goal is a top-level conjunction containing a single branched goal. There are two types of information about a procedure that the deforestation algorithm needs to know.

- For which arguments is there extra information at the ends of the branches. The type of extra information being looked for is the case where at the end of a branch the functor of an argument is known, but after merging the information for all branches, it is not. The algorithm also looks at the sub-branches of the top level branched goals to detect cases where it would be worthwhile to push a following call multiple levels into the branched goal.

- For which arguments is knowing the top-level functor sufficient to allow determinate unfolding of the procedure. At the moment this only checks if an argument is switched on at the top level of the predicate.

4.3.2 Optimization of conjunctions

Before processing a conjunction, each conjunct is annotated as described above. When a call is reached, later conjuncts are checked to see if they are candidates for optimization with the current goal. A later conjunct is a candidate for optimization if its switched on variable is a member of the set of variables for which there is extra information in the first goal.

If the conjunction formed by the two goals being checked and the goals in between is an instance of a conjunction previously chosen for deforestation, a fold is performed, replacing the conjunction with a call to the predicate created for the previously attempted optimization. Otherwise, a new predicate is created containing the conjunction to be optimized, and the process described for double_append above is applied to perform the optimization.

Unfold/fold transformation procedures such as our deforestation algorithm need to be careful that folding does not introduce infinite loops in the transformed program. This can happen if a new definition is added then folded with itself. Our algorithm always performs a step of unfolding before allowing folding to occur. This means that the transformed program is always guaranteed to perform some work before encountering a recursive call resulting from a fold, so an infinite loop that was not already present in the original program cannot have been introduced.

Some method is needed to measure the improvement in performance caused by the deforestation process to avoid creating useless new versions. One common case where the deforestation process cannot produce much improvement is where the second call depends on an output of the first which cannot be removed by deforestation, for example an io__state. In this case the algorithm unfolds once then the termination check fails, increasing code size for very little gain. The compiler avoids creating specialized versions in this situation by tracking which arguments cannot be deforested. An argument cannot be deforested if there is no information about which constructor it is bound to at the end of the branches of the first goal. If any of the later goals in the conjunction to be optimized depend on such an argument of the first call, the optimization is not attempted.

In many cases the traversal of the input data structure is a minor cost in the execution of the predicate. The heuristic used the Mercury compiler compares an estimate of the original cost with an estimate of the improvement in cost by performing the optimization. The cost estimate is necessarily crude, since profiling information is not available. Optimizations leading to folding are favoured, since these cause improvement on every iteration.

4.3.3 Ensuring termination

It is possible that the optimization process given above could fail to terminate. To ensure that an infinite sequence of specialized conjunctions is never created, the algorithm needs to apply a process similar to that applied while unfolding In this case, we need to define an ordering on the specialized conjunctions, and make sure that each conjunction is in some way smaller than all of its parent conjunctions.

In the case of deforestation, a simple termination check works quite well, since in most cases we are only interested in performing the optimization if recursive calls in the body of the conjunction can be folded in one or two steps. For deforestation, in most cases where a simple termination check would fail, it is because folding has not occurred, so we should stop optimizing.

The method used by the current implementation first looks at the length of the conjunction being specialized. A conjunction being deforested consists of a pair of calls, one on each end, with some other goals in the middle. The idea is that for each pair of predicates in the original program, the conjunction length must decrease. It is possible that if a folding step occurred earlier in the conjunction currently being processed, one or both of the calls at the ends of the conjunction being examined was introduced by folding with a specialized conjunction. The termination checker must map each specialized predicate to a predicate from the original program, otherwise there are not a provably finite number of pairs of predicates under consideration and the termination check is not guaranteed to detect loops. This mapping is done by looking at the conjunction which caused the specialized predicate to be created. For example if the algorithm is trying to map the call at the left end of a conjunction back to a predicate in the original program, it will take the left endpoint of the conjunction for which the called predicate was created. This process may need to be applied recursively.

Once the calls at the end of the conjunction have been mapped to a pair predicates in the original program, the parent conjunctions with that pair of predicates are examined. If the conjunction is shorter than all parent conjunctions with those two calls at the ends, a new predicate can be created. If the conjunction is longer than any parent conjunction involving those calls, the deforestation process stops. If the length is the same as the shortest parent conjunction, first the sizes of the argument insts of the previously created version are compared with the sizes of the arguments for the current goal. If the sizes are the same and the goals otherwise match, differing only in the initial insts of the argument variables, a generalisation step takes place.

4.3.4 Generalisation

Generalisation handles a common case where one of the arguments of a recursive predicate to be deforested is a counter which is incremented for each iteration. Folding will not take place because the initial inst of the counter will not be compatible at the next iteration. Figure 3.2 shows an intermediate stage of the deforestation of calls to append and length, after the call to append has been unfolded and the call to length has been pushed into the branches and unfolded where appropriate. Note that in the second disjunct, there are adjacent calls to *append/3* and *length/3* in the same pattern as the original goal.

The problem is that the inst of the second argument to length has inst 'bound(1)' in the second disjunct, whereas in the original goal it has inst 'bound(0)'. The insts do not match, so folding cannot take place. In this case however, the extra information on the call to *length/3* is not needed – an initial inst of 'ground' for that argument results in a mode correct procedure – 'ground' is a "more general" inst than 'bound'. We must rerun mode analysis on the generalised conjunction to be sure of this, since the mode correctness of the goal may have relied on the extra information being present. If mode analysis fails, the optimization stops.

Since the argument inst sizes have now decreased ('ground' is smaller than 'bound(...)') we can continute the optimization process. After performing another predicate definition and unfolding step, we can fold giving the result shown in figure 3.3. This is similar to the most specific generalisation process described in [13].

The optimized code in figure 3.3 includes an unnecessary wrapper predicate which performs the first recursive iteration. This results from only finding out that generalisation is required when the recursive call is encountered during the deforestation process. The code duplication does not seem to be a problem in practice, so we have not taken steps to avoid it. The amount of code duplicated is typically small, and it is more common for deforestation to occur without generalisation being required. The possible cost is more cache misses due to increased code size.

Deforesting a newly created predicate which required generalisation with another predicate later in the conjunction can be problematic. Consider the following conjunction after the optimization of length_app has been performed.

```
Length0 = 0, la(As, Bs, Cs, Length0, Length), append(Cs, Ds, Es).
```

The call to *la/4* and the call to *append/3* can be deforested, eliminating the construction of the intermediate list Cs. The process begins as normal to give the program state in figure 4.1. The conjunction la1...append in the second disjunct does not match the conjunction being deforested – the first predicate is different.

When performing the termination check on the conjunction, predicate *la1/5* is mapped back to the calls in the conjunction which caused its creation, which are *length/3* and *append/3*. The endpoints of this new conjunction are therefore *length/3* and *append/3*, which is the same as for the parent predicate *la_app/6*. Therefore the termination check fails when attempting to create a new version for the conjunction.

It seems intuitively obvious that we should be able to deforest this case. The problem is that we have introduced a new predicate for the generalisation which does not match. The solution is to "undo" the generalisation process which created *la1/5*, turning the call to the generalised version *la1/5* back into the specific version *la/5*. This gives a conjunction which matches the definition of *la_app/5*, except that the inst of Length0 is 'bound(0)' and the inst of Length1 in the recursive call is 'bound(1)'. The generalisation process can now be reapplied to the entire conjunction, replacing the inst of Length1 with 'ground'. The argument insts now decrease, and the process can continue, giving the expected specialized code shown in figure 4.2.

4.3.5 Reordering

Some unnecessary code duplication can be performed when the code between the calls to be deforested is pushed into the branches of the first goal. Also, the algorithm given does not work well for cases where there are two or more recursive calls in a branch. For example, when deforesting two binary tree traversals as in figure 5, the traversals of the left and right subtrees in the unfolded version of each call should be paired. Another effect of not reordering would be that if for some reason the intermediate structure in *double_append/4* could not be removed, folding would not occur.

The deforestation algorithm performs reordering of conjunctions for which deforestation is attempted to minimise the number of irrelevant goals included in the conjunction to be optimized and to ensure that the correct goals are paired for optimization. Moving goals earlier is preferred to avoid making a tail-recursive call non-tail-recursive, and also to avoid evaluating a goal which constrains the computation later than necessary (see section 5).

Two goals should not be reordered if the second uses some of the first goals outputs, since the reordered conjunction might not be mode correct, or it might be less efficient due to choosing a mode for one of the predicates which explores a larger proportion of the search space than the original. Also, the reordering must not worsen the termination behaviour of the program according to the language semantics options chosen by the programmer (see the Mercury Language Reference

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

length_app(As, Bs, Cs, ALength) :-
    ALength0 = 0,
    length(As, ALength0, ALength),
    append(As, Bs, Cs).

length(As, L0, L) :-
    (
        As = [],
        L = L0
    ;
        As = [_ | As1],
        L1 is L0 + 1,
        length(As1, L1, L)
    ).
```

3.1: length_app before optimization

```
:- pred la(list(int)::in, list(int)::in, list(int)::in,
        int::in(bound(0)), int::out) is det.

la(As, Bs, Cs, ALength0, ALength) :-
    (
        As = [],
        Cs = Bs,
        ALength = ALength0
    ;
        As = [A | As1],
        ALength1 = 1,
        length(As1, ALength1, ALength),
        append(As1, Bs, Cs1),
        Cs = [A | Cs1]
    ).
```

3.2: length_app after unfolding, merging and simplification

```
:- pred la(list(int)::in, list(int)::in, list(int)::in, int::in(bound(0)), int::out) is det.

la(As, Bs, Cs, ALength0, ALength) :-
    (
        As = [],
        Cs = Bs,
        ALength = ALength0
    ;
        As = [A | As1],
        ALength1 = 1,
        la1(As, Bs, Cs, ALength1, ALength),
        Cs = [A | Cs1]
    ).

:- pred la1(list(int)::in, list(int)::in, list(int)::in, int::in, int::out) is det.

la1(As, Bs, Cs, ALength0, ALength) :-
    (
        As = [],
        Cs = Bs,
        ALength = ALength0
    ;
        As = [A | As1],
        ALength1 is ALength0 + 1,
        la1(As, Bs, Cs, ALength1, ALength),
        Cs = [A | Cs1]
    ).
```

3.3: length_app after optimization

Figure 3: Deforestation requiring generalisation

```
la_app(As, Bs, Ds, Es,
        Length0, Length) :-
    (
        As = [],
        append(Bs, Ds, Es)
        Cs = Bs,
        Length = Length0
    ;
        As = [A | As1],
        Length1 = 1,
        la1(As1, Bs, Cs1,
            Length1, Length),
        append(Cs1, Ds, Es)
    ).
```

4.1: Intermediate step of deforestation of la_app

```
la_app(As, Bs, Ds, Es,
        Length0, Length) :-
    (
        As = [],
        append(Bs, Ds, Es)
        Cs = Bs,
        Length = Length0
    ;
        As = [A | As1],
        Length1 = 1,
        la_app1(As1, Bs, Cs1,
            Length1, Length)
    ).

la_app1(As, Bs, Ds, Es,
        Length0, Length) :-
    (
        As = [],
        append(Bs, Ds, Es)
        Cs = Bs,
        Length = Length0
    ;
        As = [A | As1],
        Length1 = 1,
        la_app1(As1, Bs, Cs1,
            Length1, Length)
    ).
```

4.2: Final result of deforestation of la_app

Figure 4: Undoing generalisation

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

:- pred add2_tree(tree(int)::in,
        tree(int)::out) is det.

add2_tree(T0, T) :-
    add1_tree(T0, T1),
    add1_tree(T1, T).

:- pred add1_tree(tree(int)::in,
        tree(int)::out) is det.

add1_tree(T0, T) :-
  (
    T0 = nil,
    T = nil
  ;
    T0 = node(TL0, N0, TR0),
    N is N0 + 1,
    add1_tree(TL0, TL),
    add1_tree(TR0, TR),
    T = node(TL, N, TR)
  ).
```

5.1: add2_tree before deforestation

```
add2_tree(T0, T) :-
  (
    T0 = nil,
    T = nil
  ;
    T0 = node(TL0, N0, TR0),
    N1 is N0 + 1,
    add1_tree(TL0, TL1),
    add1_tree(TR0, TR1),
    N is N1 + 1,
    add1_tree(TL1, TL),
    add1_tree(TR1, TR),
    T = node(TR, N, TR)
  ).
```

5.2: add2_tree after unfolding, branch merging and simplification

```
add2_tree(T0, T) :-
  (
    T0 = nil,
    T = nil
  ;
    T0 = node(TL0, N0, TR0),
    N1 is N0 + 1,
    add1_tree(TR0, TR1),
    N is N1 + 1,
    add1_tree(TL0, TL1),
    add1_tree(TL1, TL),
    add1_tree(TR1, TR),
    T = node(TR, N, TR)
  ).
```

5.3: add2_tree after first reordering

```
add2_tree(T0, T) :-
  (
    T0 = nil,
    T = nil
  ;
    T0 = node(TL0, N0, TR0),
    N1 is N0 + 1,
    N is N1 + 1,
    add2_tree(TL0, TL),
    add1_tree(TR0, TR1),
    add1_tree(TR1, TR),
    T = node(TR, N, TR)
  ).
```

5.4: add2_tree after folding and second reordering

```
add2_tree(T0, T) :-
  (
    T0 = nil,
    T = nil
  ;
    T0 = node(TL0, N0, TR0),
    N1 is N0 + 1,
    N is N1 + 1,
    add2_tree(TL0, TL),
    add2_tree(TR0, TR),
    T = node(TR, N, TR)
  ).
```

5.5: add2_tree after deforestation

Figure 5: Deforestation requiring reordering

Manual [33]). For example the strict sequential semantics specifies that the behaviour must be the same as if the program were executed using a strategy similar to Prolog with a minimal reordering of conjunctions to make them mode correct. This semantics would disallow reordering a two goals if the first goal could fail and the other goal might not terminate – reordering the goals could replace finite failure with an infinite loop. The Mercury compiler includes a state-of-the-art termination analyser [35] which can aid in this decision.

4.4 Interaction with separate compilation

One problem with applying the deforestation process given to multi-module programs is that the code for called procedures is needed to perform the optimization. Fortunately, the Mercury compiler has support for inter-module optimization (see section 3.6), which has been trivially extended to expose the code for predicates which may be candidates for deforestation to importing modules.

4.5 Benchmark results

We have evaluated our system using the DPPD (Dozens of Problems for Partial Deduction) benchmark set [17]. The only alterations made to the benchmark programs were to add type, mode and determinism declarations and to make some programs deterministic using if-then-elses instead of disjunctions. Some of the programs could not be converted to Mercury. The benchmarks liftsolve.app liftsolve.db1, liftsolve.db2, liftsolve.lkmng, depth, ex_depth, memo-solve and model_elim are not well-moded.

The benchmarks were performed on a 2 CPU 200MHz Pentium Pro with 256Mb of RAM running Linux (kernel 2.1.102). The benchmarks were compiled with optimization level -O5.

Times are reported in milliseconds for 100,000 iterations. One major criticism of this benchmark set is that many of the benchmarks are too small, resulting in inaccurate timings. Speed-up is reported as the difference in runtimes expressed as a percentage of the old run-time. Code size is reported as the percentage increase in object file size from the unoptimized version to the optimized version. Where the code size difference is zero, our algorithm had no effect, so any time difference is just timing noise, which appears to be plus or minus 10%.

The poor results for the contains, match and remove benchmarks are due to our implementation not propagating equalities and inequalities.

The excellent results for the upto_sum1 and upto_sum2 benchmarks arise from completely removing all data structures – the optimized programs do only integer manipulation.

app_last, ground_unify.complex and ground_unify.simple test a special bottom-up propagation pass [21] present in the ECCE partial deduction system [18] for which the Mercury compiler contains no equivalent.

advisor, missionaries and ssuply are non-deterministic, so the determinate unfolding rules used by our algorithm are not likely to produce much improvement.

map.reduce and map.rev are simple higher-order specialization problems. Higher-order specialization is currently handled as a separate pass within the Mercury compiler (see section 3.3), so our deforestation algorithm results in no further improvement. The large speed-ups reported by other researchers [13] for these benchmarks are mainly due to poor implementations of higher-order language features (‘call/1’ rather than ‘call/N’).

We have also applied our deforestation algorithm to the Mercury compiler (approximately 150,000 lines of Mercury) and standard library (about 33,000 lines). Unfortunately this does not result in a measurable change in runtime. The main reason for this is that much of the runtime of the compiler is taken up by predicates manipulating 2-3-4 trees and sets represented by ordered lists. These predicates are hidden under two levels of interfaces to allow easy replacement of the

Benchmark	Unoptimized (ms)	Optimized (ms)	Speed-up (%)	Code size increase (%)
advisor	590.0	300.0	49.2	26.6
applast	3907.0	4603.0	-17.8	0.0
contains_kmp	5490.0	6337.0	-15.4	0.0
contains_lam	3133.0	2647.0	15.5	0.0
doubleapp	8177.0	5297.0	35.2	12.8
flip	4053.0	2020.0	50.2	-0.2
imperative_solve_power	17080.0	16633.0	2.6	-0.4
map_reduce	1780.0	1660.0	6.7	0.0
map_rev	3597.0	3027.0	15.8	0.0
match_kmp	467.0	417.0	10.7	0.0
match	187.0	180.0	3.7	7.9
match_append	164153.0	161910.0	1.4	7.9
maxlength	683.0	510.0	25.3	14.9
missionaries	16780.0	15983.0	4.7	0.3
regexp_r1	5893.0	4467.0	24.2	3.9
regexp_r2	1927.0	1887.0	2.1	3.9
regexp_r3	2620.0	2567.0	2.0	3.9
relative	753.0	743.0	1.3	0.0
remove	13143.0	13683.0	-4.1	-4.2
remove2	14620.0	13160.0	10.0	26.2
rotateprune	4670.0	3680.0	21.2	0.0
ssuply	73.0	80.0	-9.6	3.2
transpose	5273.0	5077.0	3.7	-10.0
upto_sum1	11333.0	393.0	96.5	-10.0
upto_sum2	3280.0	470.0	85.7	

Table 2: Performance results on the DPPD benchmark set

data structures, which hinders inter-module optimization. The 2-3-4 manipulation routines would not usually be considered candidates for inter-module optimization because they are quite large, so code duplication becomes an issue.

One positive aspect of our algorithm is that it does not increase the code size significantly. The Mercury compiler is only 80Kb bigger when compiled with deforestation (compared to the size of the compiler of 5.5Mb). Where no improvement is made, the code is usually left unchanged, since the algorithm does not perform large amounts of useless unfolding looking for optimization opportunities. It also does not take too much time. Compiling one of the largest modules in the compiler, deforestation adds 3 seconds to a runtime of 90 seconds. Compiling the double_append benchmark, deforestation adds less than 0.5 seconds to a runtime of 5 seconds.

4.6 Related work

There has been much other work on transformation of functional and logic programs, some of which may be suitable for adaptation to address some weaknesses in our deforestation algorithm.

4.6.1 Functional programming deforestation

Much work has been done on deforestation in the functional programming community. Wadler [38] introduced the idea. The Glasgow Haskell compiler includes an implementation of the "shortcut" form of deforestation described in [7], which only fuses functions expressed using the 'foldr' and 'build' functions.

Supercompilation [34] is also capable of deforestation type optimizations. Supercompilation is a powerful partial evaluation mechanism which has a lot in common with conjunctive partial deduction of logic programs.

These algorithms (and ours) perform poorly on predicates which build up their outputs using accumulator arguments. As an example, our algorithm cannot derive that applying linear reverse twice to a list gives back the original list. [27, 36] suggest representing recursive predicates as *hylomorphisms* to capture the recursive structure of the predicate. A hylomorphism is a general pattern of recursion which consists of consumption of the input structure, a transformation on the input and construction of the output structure. Researchers in this field have derived transformation laws which can be applied to fuse hylomorphisms, achieving deforestation-like transformations capable of eliminating the accumulating arguments.

Similar transformations have been achieved in the attribute grammar field. Duris et. al.[6] notes a correspondence between attribute grammars and functional programs, where a production in an attribute grammar corresponds to a clause of a function. It is shown that the Descriptional Composition algorithm can be used to fuse two attribute grammars in a manner similar to the transformation laws on hylomorphisms.

4.6.2 Conjunctive partial deduction

Partial deduction is partial evaluation applied to logic programs. Conjunctive partial deduction [13] is a form of partial deduction which specializes conjunctions instead of single atoms.

The main advantage of conjunctive partial deduction is that it propagates more general information to perform optimization. The simple argument inst size check used by our deforestation algorithm is not sufficient to take into account equality information, and would result in missed opportunities for specialization. Propagating equality and inequality constraints can give a specializer enough information to convert a naive string matcher to a Knuth-Morris-Pratt algorithm. Propagating this sort of information requires the use of a more sophisticated termination check, such as a homeomorphic embedding check [19].

The main difference between our method and conjunctive partial deduction is that we do not explicitly build SLD trees, instead working directly on the Mercury compiler's internal goal representation using unfold and fold transformations. The deforestation procedure presented here performs a subset of the optimizations produced by that method, since it does not produce specialized versions of single predicate calls. Our method may be better controlled than conjunctive partial deduction because of our use of mode information to direct the transformation. Conjunctive partial deduction has a quite weak mechanism for choosing conjunctions to specialize, relying on shared variables within conjuncts to partition conjunctions into parts to specialize. This may result in much useless analysis and transformation.

Other researchers have formulated deforestation using unfold-fold transformations [29, 30]. The method described in [29] also uses a limited form of modes to help control the analysis, although their mode system does not track which constructors a variable could be bound to, so is much less effective than that of Mercury. The main weakness of the method presented in [29] is that it is only defined for definite programs extended with inequalities. Most unfold/fold procedures in the literature have lacked effective automatic control mechanisms, which makes them unsuited to use in compilers.

4.7 Conclusion

We have described an implementation of deforestation using the annotations provided by the Mercury compiler. The optimization is cheap to apply, and can give very good speedups. There are no restrictions on the application of the algorithm – arbitrary Mercury programs can be specialized.

We have shown that with sensible constraints on the choice of predicate call pairs on which to perform deforestation, the deforestation algorithm can be applied to large Mercury programs in a reasonable amount of time. This contradicts [27], where it is claimed that unfold/fold transformations are too inefficient for use in production compilers.

The deforestation process forms the basis of the constraint propagation procedure described in section 5 by providing a framework for unfolding procedure calls and creating specialized versions of predicates while ensuring termination of the process.

Our deforestation implementation is included in the Mercury 0.8 release [37]. The basic deforestation algorithm is about 4500 lines of Mercury code of which about 2500 lines were written during a summer studentship (and then substantially rewritten during the Honours period).

5 Constraint propagation

5.1 Introduction

A common style of logic program is the so-called "generate-and-test" style. An answer is computed to a predicate, then it is tested to check whether it matches some description. While this is very expressive, it can also be expensive, because a solution must be completely generated before it can be tested. What we would like to do is to apply the test while the answer is being built up, so that if the answer fails the test, the failure happens as early as possible. Where possible, goals which constrain the generated solutions should be evaluated as soon as all input arguments are bound.

This is especially important for deductive database programs, which use disk storage of tables of facts to allow computations involving large amounts of data. The storage of relations on disk may be optimized for range queries using B-tree indexes or order preserving hash functions. In this case, if the constraints on the outputs of a call to a database relation can be placed near the call, it may be possible to compute the range of tuples required from the relation and use the index to retrieve only those tuples, resulting in large savings in the disk access cost.

Some of these problems are better solved using constraint programming [22], where "generate-and-test programs" are written instead as "constrain-and-generate". For constraint logic programs the satisfaction testing for the constraints is interleaved with the generation of solution. However, these types of programs can still benefit from transformation to ensure the constraints are evaluated as early as possible, although the trade-off is slightly more complex. Moving constraints too early may increase the size of the constraint store and lead to a performance loss. Constraints implemented by addition to a constraint store should be moved just before the first goal which could cause them to fail.

Ramamohanarao and Harland [8] and Mumick, Finkelstein, Pirahesh and Ramakrishnan [24] give methods for propagating constraints as a generalisation of the magic sets transformation for deductive database compilation. Since we would like to apply constraint propagation to predicates executed using the normal top-down execution method, this transformation is not suited to our purposes.

Kemp, Ramamohanarao, Balbin and Meenakshi [14, 16] give a formulation of constraint propagation in terms of unfold/fold transformations, which we follow here.

5.2 Local constraint propagation

As with deforestation, constraint propagation consists of unfolding, some local transformation on the unfolded goal, and then folding. The local transformation for constraint propagation attempts to move all constraints as early as possible in the goal.

A choice must be made as to which goals will be treated as constraints to be propagated. Mercury's mode and determinism systems aid in making this decision by identifying which goals can fail and which goals can introduce choice points. *Nondet* goals can fail but they introduce choice points, so if anything they should be propagated later in the goal to avoid creating the choice point. *Semidet* goals can fail but they do not create points, so they can only narrow the search space.

Another consideration is stack usage. If a goal with outputs is propagated to a point earlier in a conjunction, the stack lifetime of those outputs may be extended. If the outputs were not already stack allocated or the extra variables on the stack require allocation of extra stack slots, there will be a slight efficiency loss, which will hopefully be outweighed by the gain from earlier failure. It is very difficult to evaluate the effect of this efficiency loss at compile-time, because the stack usage is difficult to obtain directly from the HLDS without actually generating code.

It may also be necessary to propagate some goals which are not constraints along with the constraints. For example, in the conjunction Z is X + Y, Z < 2 the addition must be propagated if the Z < 2 constraint is to be propagated. If the extra goal is not nondeterministic, the only efficiency cost of propagating it is the extra cost of stack manipulation to hold the outputs if they are used later in the goal. If the extra goal constructs a constant which can be shared (that is, it will never be destructively updated), the constructed term can be allocated in the data segment of the executable. The construction then does not have a time or space cost and can be freely duplicated.

As with deforestation, we must decide which goals may be reordered – the reasoning is similar to that described in section 4.3.5. If a goal is branched, constraints can be propagated into the arms of the branches and pushed in as early as possible while maintaining a mode-correct ordering. There is an issue of code explosion if large constraints are propagated into the arms of branched goals, since the constraints are duplicated in each branch. For this reason, only *atomic* goals and negations and explicit quantifications of atomic goals are propagated into branched goals. An atomic goal is a call, higher-order call or unification. Goals cannot be propagated into negated contexts such as a negation or the condition of an if-then-else since that could change the behaviour of the program – the negated context *catches* any failure occurring within it.

Reordering two goals which can both fail may be a pessimization if the one which was originally first is more likely to fail and the second is expensive. Automatically performing this kind of reordering can surprise the user with unexpectedly poor performance. Working out which call is more likely to fail and the relative costs is likely to be very difficult without information from sample runs of the program. For builtins the difference is likely to be negligible – most builtins should take about the same amount of time. For other goals, the choice we have taken is to avoid reordering two goals which can both fail, leaving the responsibility of choosing the best ordering to the programmer.

5.3 Constraint propagation heuristics

The global strategy for constraint propagation is handled as an addition to that described for deforestation in section 4. When a predicate is processed, local constraint propagation is run to push constraints as close as possible to the goals which produce their input arguments.

When a call is not deforestable with any other call later in the conjunction, the remainder of the conjunction is checked for constraints on the outputs of the call. If there are any, we attempt to propagate the constraints into the called predicate.

If the algorithm has already introduced a new predicate for a call to the called predicate constrained by that particular set of constraints, the call is replaced with a call to the introduced predicate – folding. Otherwise, a new predicate is defined consisting of the conjunction of the goal of the called predicate and the constraints on the call, appropriately renamed. Local constraint propagation is then performed to push the constraints in as far as possible and the goal is recursively processed.

The method used to ensure termination of the constraint propagation is a simple adaption of that used to ensure termination of deforestation – when creating a specialized version of a predicate, the length of the list of constraints must be smaller than for any parent version created for that predicate. As for deforestation, generalisation may be required to achieve folding if the arguments include counters incremented on each iteration.

5.4 Example

We demonstrate our constraint propagation algorithm on the ancestor ancestor program familiar from deductive databases, taken from [16]. Constraints on the output variables can be propagated into the branches of the call. Folding is not possible in this case, but the optimization is still valuable. The constraint is next to the calls to the base relation *p/2*, so it can be used to optimize retrieval of tuples if *p/2* is indexed on its first argument. Also, the constraint restricts the input to the call to the 'in, out' mode of *anc/2* in the second disjunct.

5.5 Conclusion

In this section, we have presented the implementation of constraint propagation as an extension to the deforestation algorithm described in section 4. The algorithm is effective in transforming programs so that constraints are evaluated as early as possible so that failure is detected as early as possible, reducing the search space of generate and test programs.

The implementation of the algorithm consists of 500 lines of Mercury code to perform the propagation and some minor changes to the deforestation pass.

```
:- pred query(int::out, int::out) is nondet.
query(X, Y) :-
   anc(X, Y), X < 50.

:- pred anc(int, int) is nondet.
:- pred anc(out, out) is nondet.
:- pred anc(in, out) is nondet.
anc(X, Y) :-
   (
      p(X, Y)
   ;
      p(X, Z),
      anc(Z, Y)
   ).

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

6.1: anc before constraint propagation

```
query(X, Y) :-
   anc_cp(X, Y).

anc_cp(X, Y) :-
   (
      p(X, Y)
   ;
      p(X, Z),
      anc(Z, Y)
   ),
   X < 50.
```

6.2: anc after unfolding

```
query(X, Y) :-
   anc_cp(X, Y).

anc_cp(X, Y) :-
   (
      p(X, Y),
      X < 50
   ;
      p(X, Z),
      X < 50,
      anc(Z, Y)
   ).
```

6.3: anc after local propagation

6 Structure Reuse

6.1 Introduction

Since Mercury is a pure single-assignment language, it is not possible for the programmer to destructively update a program variable. The program must be evaluated as if a new heap cell is allocated for each construction, which is inefficient. Several variables may be aliases for a single data structure, which is a possible cause of errors. Uniqueness information added to mode declarations helps towards solving these problems. The compiler can check the uniqueness annotations to ensure that a structure that is intended to be single-threaded is not copied. The compiler can also use the uniqueness information to destructively update variables and to reclaim memory without invoking the garbage collector.

Detailed unique mode declarations can be quite verbose, so it is anticipated that they will only be used where necessary for error checking or for performance. Automation of the annotation process is very desirable. Live structure analysis [4] should be able to discover enough opportunities for destructive update to allow the structure reuse optimization described here to give large speed-ups for many programs, even for large programs like the Mercury compiler. Live structure analysis for Mercury is currently being worked on by a research group at Katholieke Universiteit Leuven, Belgium.

There are two ways to use the structure liveness information to optimize the memory allocation behaviour of a program.

The first, and best, option is to reuse dead cells for new constructions. It is very common in logic programs for a new cell to contain many of the fields of recently dead garbage cell within the same predicate. Reusing the old structure can eliminate the overhead of cell allocation and of filling in the identical fields. For example, for append (see appendix A) the first field of the new cell created in the recursive disjunct is the same as the first field of the old cell.

The other alternative is to explicitly free the memory. The Boehm conservative garbage collector [1] provides the function *GC_free()* for this purpose. This is useful when there is no new cell created locally within the predicate which last uses the data which could reuse the data. The disadvantage is that there is some overhead in maintaining free lists.

For both of these methods there is the disadvantage that a pointer to the cell must be kept until it is reused, increasing stack usage and possibly making the garbage collector believe that dead data pointed to by the garbage cell is live longer than necessary.

Apart from the decision on how to implement the actual optimization, there is also a decision to be made on where to place the optimization in the list of compiler passes. Since structure reuse complicates the semantics by destructively overwriting the contents of memory cells, it should be placed after all passes which depend on the pure declarative semantics of Mercury to do their work. This means placing structure reuse and compile-time garbage collection after all other optimization passes, coming before only the code generation passes.

6.2 Data representation

Like most logic and functional programming languages, Mercury represents data as tagged heap cells. All Mercury data is word-aligned, so on byte addressed architectures there are spare bits in pointers which can be used to store tags – 2 bits for architectures with a 32 bit word size and 3 for 64 bit architectures. These bits are used to store the *primary* tag for the cell. The primary tag must be removed from the pointer before attempting to access the data. Many types have more alternatives than can be differentiated using the primary tag bits. Constants (constructors with arity zero) can share a primary tag and do not require heap allocation – the rest of the word is used to distinguish between the different constants. Several non-constants may share a primary tag. An extra word is allocated at the start of the data cell to store a *secondary* tag to distinguish between them.

For lists the nil constructor has primary tag 0 and the cons constructor has primary tag 1. No secondary tag is required.

6.3 Structure reuse

The common structure elimination process described in 3.1 optimizes cells where the arguments are exactly the same as a previous cell. If it is known that a cell will never be referenced again, it can be used to optimize a construction even if the arguments are not the same, since the differing arguments can be destructively overwritten. This is done by converting the construction into a *reconstruction* by adding a description of a cell to reuse. The description includes which variable to reuse, which constructor that variable is bound to and which arguments need to be overwritten. The constructor of the reused variable is needed to decide whether the primary and secondary tags of the reused cell must be updated. The current implementation simplifies things by only reusing cells for which the tags are the same – it is expected that this restriction will be lifted later.

For the code for *append/3* in appendix A, the first and second arguments have mode *di*, which is equivalent to 'unique→clobbered'. These arguments must have exactly one reference on entry to the procedure, and there must be no references later by the caller. Parts of these input arguments which are not already clobbered by sub-goals of append are candidates for reuse. For example, the tail of the first input argument is clobbered by the recursive call, so it cannot be reused locally – it may be able to be reused within the recursive invocation of append. The first cell of the first argument list is not clobbered by the goal, so it is reusable. The expected outcome of the structure reuse pass is that the first cell of the first argument is reused for the first cell of the third (output) argument.

The structure reuse pass is split into two phases. The mode liveness pass attaches to each goal the set of variables which become reusable before and after the goal. A second pass uses this information to keep track of which data cells are reusable and determine which reusable variables should be reused by which constructions.

6.3.1 When is a cell reusable?

A variable is reusable if its inst is 'unique' and neither it nor any of its aliases are referenced later in the program. Note that the current implementation does not reuse cells which are needed again on backtracking (these would have inst 'mostly_unique'). The main reason for this is that the assignments to the fields would have to be trailed so they could be undone on backtracking, which would probably be at least as expensive as the overhead of allocating and garbage collecting a new cell, especially when using the native garbage collector where an allocation is just a heap pointer increment.

Like most compilers, the Mercury compiler includes a pass to work out the syntactic liveness of variables. A variable is born on its first occurrence and becomes dead at its last occurrence. This syntactic liveness is distinct from the form of liveness used by unique mode analysis and structure reuse, where a variable or term is live if a syntactically live variable or a variable live on backtracking could contain a reference to it. That form of liveness is referred to as *mode liveness*. To make the analysis more tractable, the compiler makes the simplifying assumption that if a variable is referenced later, all parts of it are mode live.

The mode liveness pass must be able to detect when all references to a variable are dead. As was discussed in section 2.4 an instmap and an inst_key_table are used to hold the current instantiation state for all variables while processing a goal. These structures contain enough information to work out which variables reference which other variables. The problem is that they are not indexed in a manner which allows efficient lookups of the information required by mode liveness. Instmaps are designed for efficient lookups of the inst of a variable – mode liveness needs to look up the variables which could contain a given inst_key.

The mode liveness pass uses a structure called a *bwd_instmap*. As the name suggests, it contains a reverse indexing of the information in the inst_key_table and the current instmap. For each inst_key, we store the following items:

- The set of inst_keys in the inst_key table which reference it. (Recall that the inst_key_table maps inst_keys to insts.) This does not change over the goal.
- The set of variables whose current insts reference it.
- The set of inst_keys which the current inst_key substitution in the instmap maps to it. This is useful for undoing parts of the inst_key substitution to examine previous instantiation states of a variable.

Given a variable it is possible to use the bwd_instmap to work out which variables could contain references to it using the following process:

1. Lookup the variable's inst in the the current instmap.

   If the inst is not an *alias/1* inst, the top-level cell is unique and the variable is syntactically dead, no other variables could contain references so the variable is *mode dead*.

2. Use the reverse inst_key substitution in the bwd_instmap to work out which past instantiation states are represented by the inst_key of the variable of interest.

   This step is necessary because the inst_key substitution is not applied to the entire bwd_instmap at each update as the goal is processed since that would be very inefficient. Instead, when looking up the referencing variables we need to collect the references introduced at each past instantiation state of the variable of interest.

3. Use the reverse indexing of the inst_key table to find out which inst_keys reference any of the inst_keys found in step 2.

4. Repeat steps 2 – 4 on each inst_key found at step 3. This is necessary because it may be possible that a cell is referenced through a chain of inst_keys.

The 'inst_key to set of variables' mapping in the bwd_instmap can then be used to find out which variables reference any of the inst_keys found at steps 2 and 3. This is the set of variables which contain references to the variable of interest. For each dead variable, the compiler keeps track of which live variables contained references when the variable became dead. For each live variable, the set of referenced dead variables is maintained. When a variable is found to be dead according to the mode liveness pass, it is removed from the referencing variable sets of all variables it referenced. Any variables for which the referencing variable set becomes empty are rechecked for any new referencing variables using the algorithm above. This is necessary because some new references may have been added through aliases after the variable itself became liveness dead. It would be too inefficient to maintain this information as we go through the goal, so it is recomputed only when necessary. If there are no new references, that variable is now mode dead, and variables it references should be checked.

Dead variables are registered as available for reuse by including them in the *pre_mode_death* or *post_mode_death* set for the goal, depending on whether it died before or during the goal. A variable can only become dead before a goal if the goal is a branch of a branched goal (if-then-else, switch or disjunction) and the variable is referenced in other branches but not this one.

The other part of mode liveness is checking whether any goal on an execution path including the current goal makes a variable unsuitable for reuse by changing its uniqueness to 'shared', 'mostly_unique', 'clobbered' or 'mostly_clobbered'. 'Shared' variable have references that the mode system is not strong enough to track, so their storage can never be destructively updated.

For a conjunction, this requires looking ahead to the end of the conjunction to check the final instantiatedness of any local variables. Branched goals are slightly more tricky. If a variable is clobbered or shared in some branches but not in others, and is not referenced after the branched goal, it may be reusable within the other branches.

This information is also used to decide whether a cell could be allocated in the shared data segment of the executable or whether a heap cell must be allocated a runtime. If a data cell could never be destructively updated or explicitly freed (see section 6.4) it can be allocated in shared data. The compiler currently only knows a variable could not be updated if its inst is made 'shared' on every execution path. If the compiler cannot guarantee this, a heap cell must be allocated.

6.3.2 Choosing which cell to reuse

When the structure reuse pass finds a construction unification, it must choose which dead cell, if any, should be used for the construction. Debray [5] gives an algorithm for choosing a cell for each construction given a bipartite graph describing which constructions could use which cells, and shows that an optimal solution to the problem is NP-complete. A polynomial time heuristic algorithm is given. In many cases, even this is overkill. For most data structures which can be destructively updated there is one obvious candidate to reuse the old cell, so the complexity of the algorithm given in [5] is unnecessary.

We have chosen not to implement this since it would increase the cost and complexity of this pass of the compiler with possibly very little gain over a simpler algorithm. In most cases where structure reuse is crucial for performance there will be very few cells to reuse and the choice of which cell to reuse will be fairly obvious, We would also like to avoid constructing the full choice graph as given in [5]. It would be possible to use a hybrid approach, only doing the optimal choice for straight line sections of code.

The choice of cell to reuse for a given construction is made using two criteria. First the number of fields overwritten should be minimised. Second, the stack lifetime of variables should be minimised. The algorithm prefers to reuse a cell which has recently been created, in the hope that older cells will not need to be reused and can be explicitly freed as described below.

For the case where all fields of the cells match, structure reuse is exactly equivalent to the common structure elimination described in 3.1, except that it maintains unique mode correctness. It is better to avoid constructing a cell at all than create it then reuse it for a later construction. For this reason, we first run a pass of structure reuse which only reuses cells where all the arguments match before running full structure reuse.

6.3.3 Avoiding unnecessary field assignments

The C code output by the compiler after structure reuse optimization of append is given in figure 6[1] Note that the C code for the tail recursive version of append is not quite optimal. At each recursive step the tail of the old list is assigned to the tail of the new list. This is not necessary because the memory cells are the same – the tail of the new list is already in the right place because it reuses the cell of the old list. The reason the structure reuse algorithm does not detect this is because it depends on information about the contents of the fields of the list cell which is not available until

[1]The C code has been slightly altered because the value numbering pass was not designed to handle code to which the last call optimization has been applied. Value numbering is a low level optimization of straight line code segments. It has been applied by hand to this C code, eventually the value numbering pass will be fixed. The C code has also been altered to be a little less cryptic than normal by expanding some macros.

after the structure reuse pass is complete, and which cannot be obtained locally – the structure reuse pass must have information about all calls to the predicate to know that this condition holds.

This problem could be fixed the problem by adding a post-processing pass to detect and optimize this case using a process of abstract interpretation [3] to find out which fields of reconstructions don't actually need to be filled in. The post-processing pass has not been implemented, since for the naive reverse benchmark, the overhead of the extra assignment has been measured to be of the order of 3-4%. This would be close to the maximal overhead, since naive reverse does no processing on the elements of the input lists. For most programs the difference will be negligible, so the extra pass is not worth the extra compilation time.

6.4 Explicit freeing

If the previous passes decide that a dead cell could not or should not be reused, it may be worthwhile to tell the garbage collector that the memory is no longer in use. The C headers for the Mercury runtime include a macro *MR_free_heap()*[2] for this purpose. The definition of this macro depends on the method of garbage collector in use.

When the conservative Boehm collector [1] is being used this macro expands to a call to the function *GC_free()*, which places the memory on a free list for future allocations and avoids traversing the cell when collecting. The documentation for the Boehm collector states that for small cell sizes (less than 8 bytes) adding calls to *GC_free()* is likely to be a pessimization. Our benchmarks show that for small cells, it doesn't actually make a lot of difference.

The compiler must be careful to ensure that the input to *GC_free()* actually contains allocated memory. This precludes compile-time garbage collection of abstract data types such as destructively updated arrays, which are implemented using the C interface. Currently, the compiler only explicitly frees memory when it knows the top-level constructor of the variable.

When there is no garbage collection, *MR_free_heap()* expands to nothing. Introduction of calls to *MR_free_heap()* is disabled when garbage collection is not being used, since the only effect is increased stack usage due to the need to save the data passed to *MR_free_heap()*.

The current prototype of the native Mercury garbage collector does not have a free list, so compile-time garbage collection is not possible. When accurate garbage collection is being used, memory is allocated by incrementing the heap pointer, so searching in a free list would add unacceptable overhead on forward execution.

6.5 Possible problems

Performance improvement is not guaranteed with this implementation of structure reuse. The overhead of storing pointers to cells on the stack unnecessarily could outweigh the gains made. There is also an issue of safety. Since the liveness of cells to be reused is extended, the program may use more memory than before. If pointers to a large structure are kept unnecessarily across a call, a program which terminated successfully without structure reuse may actually run out of memory and abort.

It is unlikely that these problems will be significant in practice. The problems will be most significant when a large data structure is modified but a pointer to the original is kept unnecessarily. There are two factors that reduce the impact of this. Firstly, the subcomponents of the modified data structure may also be available for reuse. Also, it is not likely that the entire data structure will be changed, so a large portion of the original structure will still be live without the unnecessary pointer. Cases where success is turned into an abort due to lack of memory will most likely be

[2] MR stands for *M*ercury *R*untime, and is prepended to symbols in the runtime to avoid name clashes with user programs

```
/* code for predicate 'app'/3 in mode 0 */
/* input arguments r1 -> As, r2 -> Bs */
mercury__app__app_3_0:
    incr_sp_push_msg(2, "app:app/3"); /* Make a stack frame. */
    MR_stackvar(2) = succip; /* Save the return address. */
    if (r1 != NIL) /* Work out which arm of the switch to take. */
        goto mercury__app__app_3_0_i3;
    r1 = r2;   /* Cs := Bs */
    succip = MR_stackvar(2); /* Restore the return address. */
    decr_sp_pop_msg(2); /* Destroy the stack frame. */
    proceed(); /* Return. */
mercury__app__app_3_0_i3:
    r3 = &field(strip_tag(r1), 1); /* Take the address of the tail of the list. */
    MR_stackvar(1) = r1; /* Save the first cell of the list across the call. */
    r1 = field(strip_tag(r1), 1); /* Get the tail of the first argument. */

    /* Call app_3_1, returning to the label mercury__app__app_3_0_i4. */
    call_localret(mercury__app__app_3_1, mercury__app__app_3_0_i4);
mercury__app__app_3_0_i4:
    r1 = MR_stackvar(1); /* Put the output in r1. */
    succip = MR_stackvar(2); /* Restore the return address. */
    decr_sp_pop_msg(2); /* Destroy the stack frame. */
    proceed(); /* Return. */

/* code for predicate 'app'/3 in mode 1 */
/* input arguments r1 -> As, r2 -> Bs, r3 -> &Cs */
mercury__app__app_3_1:
    if (r1 == NIL)
        goto mercury__app__app_3_1_i2;
mercury__app__app_3_1_i4:
    while (1) { /* Follow the first list down to the end. */
    *r3 = r1; /* This assignment is not actually necessary, since *r3 == r1. */
    r3 = &field(strip_tag(r1), 1); /* Take the address of the tail of the first argument. */
    r1 = field(strip_tag(r1), 1); /* Extract the tail of the first argument. */
    if (r1 != NIL)
        continue; /* Reached the end of the first list. */
    break; } /* end while */
mercury__app__app_3_1_i2:
    *r3 = r2; /* *Cs := Bs */
    proceed(); /* return */
```

Figure 6: C code for append after structure reuse

Benchmark	Unoptimized (sec)	Compile time GC (sec)	Structure reuse (sec)
nrev	25.4	25.3	5.2
insert	10.9	9.7	4.6

Table 3: Benchmark results

very rare, however it would be nice to provide some guarantees about the behaviour of programs after the optimization is applied.

One way to solve the problem is to zero out parts of cells which will be overwritten by the reusing construction. Of course this has overhead which may reduce the usefulness of the optimization. This process also has an interaction with reusing cells within a branched goal. If a cell is created then reused in a branched goal later in the conjunction, some of the branches may overwrite different sets of fields. For data to be kept no longer than necessary no matter which branch is taken, only those fields which are not updated by any branch should be left non-zeroed. This may result in a field being zeroed and then having its old value restored in some branches.

For this zeroing "optimization" to be effective, the last use of the cell should not be a call, since the cell may become dead early in the execution of the called predicate. The zeroing of the arguments would only take place after the variable had been dead for a while. This is not a significant restriction, since if there is a cell for which a called predicate has unique access and the cell is not live after a call, it would be better if the cell were reused within the called predicate (that is, the argument be given mode *di*).

When debugging, structure reuse optimization is turned off, since the overwriting of variable values would be confusing to the programmer. It also makes it impossible to restore an earlier program state to retry a call to a buggy predicate. Common structure optimization can still be applied, since it does not destructively change the values of variables.

6.6 Benchmark results

The effectiveness of structure reuse is shown by the benchmarks in 3. 'nrev' is 100000 iterations of the naive reverse benchmark which reverses a list of 30 integers using the quadratic reverse algorithm.. 'insert' is insertion of 1000 pseudo-random integers into a binary tree 1000 times. The benchmarks were performed on a 8 CPU AlphaServer 8400/5/300 with 8Gb of memory with optimization level -O5.

For these benchmarks, structure reuse is capable of reusing all the storage, so running with both structure reuse and compile time garbage collection would not make any difference.

As expected, structure reuse is far more effective in reducing the runtime. Compile-time garbage collection has a negligible impact on performance for nrev. With the slightly larger cell sizes for the insert benchmark, compile-time garbage collection has a slightly greater effect.

6.7 Comparison and conclusion

The algorithm for structure reuse described by Debray [5] suggests memory reuse methods far more complex than those implemented here. It allows splitting one large cell for use in multiple constructions. We do not allow this because the conservative garbage collector [1] currently used by Mercury does not allow such splitting. Debray also allows reuse of a cell for constructions of a smaller size. This could lead to memory leaks when using the Boehm collector so we do not allow it. The accurate garbage collector has knowledge of the shape (and size) of the data, which would allow this type of reuse. It is doubtful whether the extra implementation complexity is worthwhile. Finally the polynomial time method presented in [5] can perform a more intelligent choice compared to the "greedy" method used in the Mercury compiler. Debray gives benchmark results showing significant speed-ups compared to a memory reuse scheme which makes no attempt to avoid overwriting fields of cells which already have the right value. Our method should not be far behind, and on most benchmarks will give exactly the same results.

The only other language of which we are aware that performs a structure reuse analysis similar to ours is Concurrent Clean [26]. Clean has a unique type system which performs a similar function to unique modes in Mercury. Not much is known about the Clean implementation of structure reuse – nothing has been published yet, and Clean is a closed source commercial system so we cannot examine the implementation ourselves.

The mode declarations of predicates in the Mercury compiler have not yet been annotated with uniqueness declarations to take advantage of the structure reuse optimization. The main reason is that the development branch of the compiler which contains the alias tracking mode analyser will need to be merged back onto the main development branch at some time, and this merge would be made much more difficult by the sort of changes to every module in the compiler that would be required to add unique modes.

The implementation of structure reuse consists of about 2500 lines of new code and changes to the code generator and low-level optimization passes of the compiler to handle reconstructions and calls to *GC_free()*. The workspace containing the implementation is /home/mercury1/stayl/mercury3/mercury.

7 Conclusion and further work

We have described our implementation of some new optimization passes in the Mercury compiler, specifically the type specialization, deforestation, constraint propagation and structure reuse passes.

Type specialization reduces the overhead of polymorphic code by replacing calls to polymorphic unification or comparison routines with a direct call to the type specific routine. This can give large speed-ups for predicates which use polymorphic unification heavily.

The deforestation pass of the Mercury compiler is able to remove the construction of unnecessary intermediate data structures. It is also able to perform loop unrolling where the top-level functor of an induction variable is known. Deforestation can produce large speed-ups for small examples where a large part of the execution time is taken constructing intermediate structures. Unfortunately, deforestation does not provide much speed-up for the Mercury compiler itself because the 2-3-4 tree lookups which take a large part of the time profile of the compiler's execution are not suited to this form of optimization.

The aim of the constraint propagation extension to the deforestation pass is to transform the code to evaluate possibly failing goals as early as possible, reducing the search space. This is especially important for deductive database programs which can operate on large amounts of data, and for which constraints on the outputs of a call can allow the use of relation indexing to restrict the number of tuples retrieved from disk.

The implementation of structure reuse and compile-time garbage collection takes the Mercury compiler closer to the aim of producing code similar to that a C programmer would write. Currently, these optimizations produce minimal speed-up for the compiler. Structure reuse should produce significant speed-up when uniqueness annotations are added for heavily updated compiler data structures, or when an automatic uniqueness analyser and specializer is implemented.

There is much more which could be done to improve the high-level optimization capabilities of the Mercury compiler. The deforestation process has a specific aim to fuse predicates with similar recursive structure. The optimization of single calls is quite crude - if the top-level functor of an induction variable is known, the call is unfolded and specialized. It would be useful in some cases to generalise this to make specialization decisions based on other criteria, for example whether two arguments are equal. Other researchers have shown that for some specific cases this results in interesting transformations such as the conversion of brute force string matching to the Knuth–Morris–Pratt algorithm [34].

In the current compiler it is difficult to make decisions about which specializations are worthwhile and which are not. The main reason is that the compiler has no idea how much of the run time or memory usage of the program is taken up by each predicate. This makes the decision as to which type specializations to produce difficult. The solution is to incorporate profiling feedback into the compiler. A member of the Mercury group is currently working on this.

There is much scope for improvement of the inter-module optimization performed by the compiler. Currently inlining is only performed across one module boundary. Higher-order specialization can result in multiple copies of the same predicate being produced. Worthwhile speed-ups could be achieved by implementing transitive inter-module inlining, and request-driven higher-order and type specialization. This would also require investigation of alternatives to *make* as a build tool, since the compilation dependencies introduced by these forms of inter-module optimization are beyond its capabilities.

References

[1] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[2] Andrew Bromage. *A strong aliasing system*. Master's thesis, Department of Computer Science, University of Melbourne, Australia, to be submitted.

[3] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10:91–124, 1991.

[4] Maurice Bruynooghe, Gerda Janssens, and Andreas Kågedal. Live-structure analysis for logic programming languages with declarations. Technical Report CW231, Department of Computer Science, K.U. Leuven, May 1996.

[5] Saumya Debray. On copy avoidance in single assignment languages. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 393–407, 1993.

[6] Etienne Duris, Didier Parigot, Giles Roussel, and Martin Jourdan. Structure directed genericity in functional programming and attribute grammars. Technical Report 3105, INRIA, February 1997.

[7] A. Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Sixth Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[8] James Harland and Kotagiri Ramamohanarao. Constraint propagation for linear recursive rules. Technical Report 92/8, Department of Computer Science, University of Melbourne, June 1992.

[9] Fergus Henderson. *Strong modes can change the world!* Honours thesis, Department of Computer Science, University of Melbourne, Australia, 1993.

[10] Fergus Henderson. *The design and implementation of a declarative programming language.* PhD thesis, Department of Computer Science, University of Melbourne, Australia, to be submitted.

[11] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.

[12] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. Technical Report 98/13, Department of Computer Science, University of Melbourne, September 1998.

[13] Jesper Jørgensen, Michael Leuschel, and Bern Martens. Conjunctive partial deduction in practice. Technical Report CW242, Department of Computer Science, K.U. Leuven, Leuven, Belgium, 1996.

[14] David Kemp. *On the foundations of query evaluation in deductive databases.* PhD thesis, Department of Computer Science, University of Melbourne, Australia, 1992.

[15] David B. Kemp, Thomas C. Conway, Evan P. Harris, Fergus J. Henderson, Kotagiri Ramamohanarao, and Zoltan Somogyi. Database transactions in purely declarative logic programming language. Technical Report 96/45, Department of Computer Science, University of Melbourne, Melbourne, Australia, December 1996.

[16] David B. Kemp, Kotagiri Ramamohanarao, Isaac Balbin, and Krishnamurthy Meenakshi. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming*, pages 981–998, Cleveland, Ohio, October 1989.

[17] Michael Leuschel. DPPD (Dozens of Problems for Partial Deduction). http://www.cs.kuleuven.ac.be/ dtai/prototypes/dppd.html.

[18] Michael Leuschel. Ecce partial deduction system. http://www.cs.kuleuven.ac.be/ dtai/prototypes/ecce.html.

[19] Michael Leuschel. On the power of homeomorphic embedding for online termination. In *Proceedings of SAS '98*, Lecture Notes in Computer Science 1503, 1998.

[20] Michael Leuschel and Bern Martens. Global control for partial deduction through characteristic atoms and global trees. Technical Report CW220, Department of Computer Science, K.U. Leuven, Leuven, Belgium, 1995.

[21] Michael Leuschel and Danny De Schreye. Logic program specialisation: how to be more specific. In *Proceedings of PLILP '96*, Lecture Notes in Computer Science 1140, 1996.

[22] Kim Marriott and Peter J. Stuckey. *Programming with constraints: an introduction.* MIT Press, 1998.

[23] Bern Martens. *On the semantics of meta-programming and the control of partial deduction in logic programming.* PhD thesis, K.U. Leuven, February 1994.

[24] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. *ACM Transactions on Database Systems*, 21(1):107–155, March 1996.

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

[26] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, pages 202–219, Eindhoven, The Netherlands, June 1991.

[27] Y. Onoue, Z. Hu, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, February 1997.

[28] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Enhancing partial deduction via unfold/fold rules. In *Logic Program Synthesis and Transformation, Proceeding of the 6th International Workshop, LOPSTR '96*, Lecture Notes in Computer Science 1207, pages 146–168, Stockholm, Sweden, 1997.

[29] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Reducing nondeterminism while specializing logic programs. In *24th ACM Symposium on Principles of Programming Languages (POPL '97)*, La Sorbonne, Paris, France, January 1997.

[30] Maurizio Proietti and Alberto Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[31] Dan Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12:7–51, 1993.

[32] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[33] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. The Mercury language reference manual. http://www.cs.mu.oz.au/mercury.

[34] Morten Heine Sørenson. *Turchin's Supercompiler Revisited.* Master's thesis, Department of Computer Science, University of Copenhagen, 1994.

[35] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. Technical Report 97/9, Department of Computer Science, University of Melbourne, July 1997.

[36] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Seventh Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.

[37] The Mercury Team. Mercury 0.8. http://www.cs.mu.oz.au/mercury, November 1998.

[38] Philip Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A HLDS code for append

This is the intermediate representation of the code for append produced by the Mercury compiler, slightly cleaned up for publication.

```
:- mode app((builtin:di), (builtin:di), (builtin:uo)) is det.
app:app(HeadVar__1_1, HeadVar__2_2, HeadVar__3_3) :-
  % nonlocals: HeadVar__1_1, HeadVar__2_2, HeadVar__3_3
  % determinism: det
  ( % cannot_fail switch on ‘HeadVar__1_1’
    % HeadVar__1_1 has functor list:./2
    % new insts: HeadVar__1_1 -> unique(list:’.’(unique, unique))

    % nonlocals: HeadVar__1_1, HeadVar__2_2, HeadVar__3_3
    % determinism: det
    ( % conjunction
      % nonlocals: HeadVar__1_1, X_5, L1_6
      % determinism: det
      HeadVar__1_1 = list:[X_5 | L1_6]
      % HeadVar__1_1 => list:./2 (X_5, L1_6)
      % new insts:
      % HeadVar__1_1 ->
      %   unique(list:’.’(alias(IK_20, unique), alias(IK_21, unique)))
      % X_5 -> alias(IK_18, unique)
      % L1_6 -> alias(IK_19, unique)
      % Inst key substitutions:
      %  IK_18 -> IK_20
      %  IK_19 -> IK_21
      ,
      % nonlocals: HeadVar__2_2, L1_6, L3_8
      % determinism: det
      app(L1_6, HeadVar__2_2, L3_8)
      % pred id: 0, proc id: 0
      % new insts: HeadVar__2_2 -> alias(IK_24, clobbered)
      %            L1_6 -> alias(IK_23, clobbered)
      %            L3_8 -> unique
      % Inst key substitutions:
      %  IK_21 -> IK_23
      %  IK_22 -> IK_24
      ,
      % nonlocals: HeadVar__3_3, X_5, L3_8
      % determinism: det
      HeadVar__3_3 = list:[X_5 | L3_8]
      % HeadVar__3_3 := list:./2 (X_5, L3_8)
      % new insts:
      % HeadVar__3_3 ->
      %   unique(list:’.’(alias(IK_20, unique), alias(IK_25, unique)))
      % L3_8 ->      alias(IK_25, unique)
    )
```

```
  % new insts:
  % HeadVar__1_1 ->
  %   unique(list:'.'(alias(IK_20, unique), alias(IK_21, clobbered)))
  % HeadVar__2_2 -> alias(IK_22, clobbered)
  % HeadVar__3_3 ->
  %   unique(list:'.'(alias(IK_20, unique), alias(IK_25, unique)))
  % X_5 -> alias(IK_18, unique)
  % L1_6 -> alias(IK_23, clobbered)
  % V_7 -> alias(IK_24, clobbered)
  % L3_8 -> alias(IK_25, unique)
  % Inst key substitutions:
  %  IK_18 -> IK_20
  %  IK_19 -> IK_21
  %  IK_21 -> IK_23
  %  IK_22 -> IK_24

;
  % HeadVar__1_1 has functor list:[]/0
  % new insts: HeadVar__1_1 ->     unique((list:[]))

  % nonlocals: HeadVar__2_2, HeadVar__3_3
  % determinism: det
  HeadVar__3_3 = HeadVar__2_2
  % HeadVar__3_3 := HeadVar__2_2
  % new insts: HeadVar__3_3 ->     alias(IK_26, unique)
  %
  % Inst key substitutions:
  % IK_27 -> IK_26
).
% new insts:
% HeadVar__1_1 -> unique(list:'.'(unique,clobbered) ; (list:[]))
% HeadVar__2_2 -> clobbered
% HeadVar__3_3 -> unique
```