Making Mercury Programs Tail Recursive

Peter Ross, David Overton, and Zoltan Somogyi

{petdr,dmo,zs}@cs.mu.oz.au
Department of Computer Science and Software Engineering,
University of Melbourne, Victoria 3010, Australia

Abstract. We present two optimizations for making Mercury programs tail recursive. Both operate by taking computations that occur after a recursive call and moving them before the recursive call, modifying them as necessary. The first optimization moves calls to associative predicates; it is a pure source to source transformation. The second optimization moves construction unifications; it required extensions to the mode system (to record aliases) and to the parameter passing convention (to allow arguments to be returned in memory). The two optimizations are designed to work together, and can make a large class of programs tail recursive.

1 Introduction

Recursive calls are often very close to being the last action in a clause. Frequently, however, the recursive call computes a value which must be put into a memory cell or used in a simple computation. Since the Mercury execution algorithm [13] requires both of these kinds of actions to be performed after the call, such clauses are not tail recursive. The code generated for them will therefore execute stack frame setup and teardown code in every iteration, and will require stack space proportional to the depth of recursion. This paper presents two optimizations that in many common cases can transform such code into a tail recursive form by moving code from after the recursive call to before the recursive call. These optimizations have been implemented in the Melbourne Mercury compiler [16].

The first optimization is a pure source to source transformation. It looks for recursive calls followed by associative computations which depend on the outputs of the recursive call. The optimization then creates a specialized version of the predicate in which the associative computation is done before the recursive call. This specialized predicate will have extra arguments that act as *accumulators*: they hold the cumulative results of the associative computations so far.

The second optimization looks for recursive calls followed by construction unifications that put some of the results of that call into memory cells, and moves those construction unifications before the call. This required extending the Mercury mode system to allow a restricted form of partially instantiated data structures and to record aliases, since in the transformed program the cell is not ground when it is created, but will become ground when the recursive call returns. To allow this to happen, we also had to extend the parameter passing conventions of the Mercury abstract machine to allow predicates to return selected output arguments in memory rather than in registers.

Some predicates contain both associative computations and construction unifications after the recursive call. To allow us to make such predicates tail recursive, the accumulator introduction optimization will perform its transformation even when the recursive call is followed by construction unifications as well as associative computations, knowing that the other optimization will move the construction unifications before the call.

The rest of this paper is organized as follows. The rest of this section presents some background. Section 2 describes the accumulator introduction transformation. Section 3 describes the changes needed to allow output arguments to be returned in memory. Section 4 presents benchmarking results. Section 5 discusses how our two optimizations are related to previous work. Section 6 concludes the paper.

1.1 Background

The Mercury language is described in full in the reference manual [11], and the main aspects of the implementation in [13]. Here we concentrate on what is relevant for this paper.

Mercury is a strongly typed language, so the compiler knows the type of every variable. It is also strongly moded. A predicate may have more than one mode, but the compiler treats each mode separately; we call each mode of a predicate (or function) a *procedure*. The compiler reorders conjunctions in the bodies of procedures to ensure that all the consumers of a variable come after its producer; the compiler knows the state of instantiation of every variable at every point in the program.

The Mercury compiler's internal representation of the source program is called the high level data structure, or HLDS for short. The HLDS stores programs in what we call *superhomogeneous form*. In this form,

- the definition of every procedure must be a single clause, whose body may contain arbitrarily nested if-then-elses, negations, conjunctions, disjunctions and *switch*es (a switch is a disjunction in which every disjunct unifies the same input variable against distinct function symbols), and
- the atoms (including the clause head) must have one of the forms `p(X1,...,Xn)`, `Y = X`, or `Y = f(X1,...,Xn)`.

The HLDS annotates every goal with mode information. For unifications, we show this mode information by writing `<=` for construction unifications, `=>` for deconstruction unifications, `==` for equality tests, and `:=` for assignments.

2 Accumulator Introduction

This section introduces the source to source transformation that looks for self recursive predicates where the recursive call is followed by an associative computation. This associative computation can then be moved in front of the recursive call. This problem has previously been studied as a transformation based on unfold/fold rules [3, 8, 9] or as a schemata matching transformation [5–7]. Our transformation is loosely based on Debray's [8] transformation.

2.1 Transformation

Consider a procedure whose body consists of a switch with two arms, where one arm of the switch contains the recursive call. We would like to determine whether or not the computation after the recursive call is associative. In the Mercury implementation, the transformation does this by looking for a *promise* declaration of the form

```
:- promise all [I,A,B,C,ABC]
    (
        (some [AB] p(I, A, B, AB), p(I, AB, C, ABC))
    <=>
        (some [BC] p(I, B, C, BC), p(I, A, BC, ABC))
    ).
```

Promise declarations in general are formulae that the programmer asserts to be true and on whose truth the language implementation is therefore allowed to rely; the implementation is allowed but not required to verify the promise.

This particular promise declaration follows the pattern of an associative computation. An associative predicate is one whose arguments can be divided into a (possibly empty) set of invariant inputs and three nonempty sets, two of inputs and one of outputs. The promise declaration simply states that processing `A`, `B` and `C` from right to left produces the same value for `ABC` as processing them from left to right.

We illustrate the motivation for our accumulator introduction algorithm by tracing the transformation of the predicate, `mirror`, in the mode `(in, out)`. The call to `mirror([1,2,3], M)` would bind `M` to the list `[1,2,3,3,2,1]`. Note that `append` is associative.

The superhomogeneous form [13] of the code, as generated by the Mercury compiler except with meaningful variable names, is as follows, with the code a human programmer would write on the right.

```
mirror(L, M) :-
    ( % switch on L
        L => [],
        M <= []
    ;
        L => [H|T],
        mirror(T, M0),
        Empty <= [],
        ListH <= [H | Empty],
        append(M0, ListH, M1),
        M <= [H | M1]
    ).
```

```
mirror([], []).
mirror([H|T], [H|M1]) :-
    mirror(T, M0),
    append(M0, [H], M1).
```

Construction unifications are not associative, so a simple associativity analysis might determine that the tail computation is non-associative. However, all of the construction unifications except the last one can safely be moved before the recursive call without breaking the mode constraints of the procedure. This rearrangement of goals is trivial in the Mercury compiler, as at each point in

a procedure it knows the state of instantiation of every variable, and two goals can have their order swapped provided that neither goal modifies the instantiation state of any variable referred to by the other goal. (Some other conditions must also be imposed to avoid the introduction of nontermination or unwanted exceptions, but these conditions are not an issue in our examples and we will not discuss them further in this paper.)

The recursive case now ends in `mirror(T, M0), append(M0, ListH, M1), M <= [H | M1]`. Since we will be rearranging the execution order inside this computation, we will introduce a new predicate, `mirror_acc`, with this body, whose arguments are the variables which occur both in this code fragment and in the rest of the code of `mirror`:

```
mirror_acc(T, M, ListH, H) :-
    mirror(T, M0),
    append(M0, ListH, M1),
    M <= [H | M1].
```

We will fold the end of the recursive case of `mirror` with this new predicate. The compiler will be able to use last call optimisation on this call and, as we shall see, we can make `mirror_acc` tail recursive as well. This requires first unfolding the call to `mirror` in `mirror_acc` with the original definition of `mirror`. This of course requires a renaming apart step; we have put an `A_` prefix on the variables that came from `mirror_acc`.

```
mirror_acc(A_T, A_M, A_ListH, A_H) :-
    ( % switch on A_T
        A_T => [],
        A_M0 <= [],
        append(A_M0, A_ListH, A_M1),
        A_M <= [A_H | A_M1]
    ;
        A_T => [H|T],
        Empty <= [],
        ListH <= [H | Empty],
        mirror(T, M0),
        append(M0, ListH, M1),
        A_M0 <= [H | M1],
        append(A_M0, A_ListH, A_M1),
        A_M <= [A_H | A_M1]
    ).
```

The second call to `append` depends on the preceding construction unification, so their order cannot be swapped. However we can inform the compiler that a call to append can be equivalent to a construction unification:

```
:- promise all [L,H,T] ( L = [H|T] <=> append([H],T,L) ).
```

Using this construction equivalence law, we can replace `A_M0 <= [H | M1], append(A_M0, A_ListH, A_M1)` with `append([H], M1, A_M0), append(A_M0, A_ListH, A_M1)`. Applying the associativity law, we transform again to `append(`

`M1, A_ListH, New), append([H], New, A_M1)` where `New` is a new variable of the same type as `M1`, `A_ListH` and `A_M1`. (The associative law requires the types of those three variables to be the same.) Another application of the construction equivalence law, this time in the other direction, leaves us with `append(M1, A_ListH, New), A_M1 <= [H | New])`.

Now the two calls to append are next to each other, so the associative law allows the two calls to `append` to be rearranged to become `append(ListH, A_ListH, Tmp), append(M0, Tmp, New)`, where `Tmp` is a new variable of the same type as `ListH`, `A_ListH`, `M0` and `New`. The first call to `append` no longer depends on the recursive call, so it can be moved before the recursive call. This results in the following code:

```
mirror_acc(A_T, A_M, A_ListH) :-
    ( % switch on A_T
        A_T => [],
        A_M0 <= [],
        append(A_M0, A_ListH, A_M1),
        A_M <= [A_H | A_M1]
    ;
        A_T => [H|T],
        Empty <= [],
        ListH <= [H | Empty],
        append(ListH, A_ListH, Tmp),
        mirror(T, M0),
        append(M0, Tmp, New),
        A_M1 <=  [H | New]
        A_M <= [A_H | A_M1]
    ).
```

We now use the original definition of `mirror_acc` to do folding in both `mirror` and the previous version of `mirror_acc` to obtain the following completely transformed code:

```
mirror(L, M) :-
    ( % switch on L
        L => [],
        M <= []
    ;
        L => [H|T],
        Empty <= [],
        ListH <= [H | Empty],
        mirror_acc(T, M, ListH, H)
    ).

mirror_acc(A_T, A_M, A_ListH, A_H) :-
    ( % switch on A_T
        A_T => [],
        A_M0 <= [],
        append(A_M0, A_ListH, A_M1),
```

```
        A_M <= [A_H | A_M1]
    ;
        A_T => [H|T],
        Empty <= [],
        ListH <= [H | Empty],
        append(ListH, A_ListH, Tmp),
        mirror_acc(T, A_M1, Tmp, H),
        A_M <= [A_H | A_M1]
    ).
```

The call to `append` in the base case is redundant. However, it is only redundant because, for this particular predicate, the base case initializes the answer to be the identity element for `append`. Partial deduction [12] can replace this call with `A_M1 = A_ListH`.

In the recursive case, application of the associativity law has rearranged the order of the arguments to `append`. The variable, `ListH`, is now placed at the start of the list, rather than the end. Thus the complexity of the call to `append` now becomes $O(1)$ rather than $O(N)$, improving the big-O complexity of the algorithm as a whole. Unfortunately the complexity can also change in the reverse direction. To prevent this from happening we introduce a heuristic. The heuristic is based on the idea that the accumulator data structure is usually 'bigger' than the individual data items being accumulated. This heuristic requires knowing, for each associative predicate, which argument position(s) control the runtime of calls to that predicate; for `append` this is the first argument. (This information may be available in programmer declarations, built into the compiler, or derived by the compiler via a program analysis such as the termination analysis algorithm of [14].) The heuristic says that after the rearrangement of the code, the accumulator variable should not be located in any of these argument positions.

This optimisation is sufficient to introduce tail recursion in predicates in which the recursive call is followed by one or more independent associative computations. The algorithm can also transform predicates where the recursive call is followed by one or more chains of independent associative calls, which have the additional property of commutativity. A commutative call is one which obeys the following law:

```
:- promise all [A,B,AB] ( p(A, B, AB) <=> p(B, A, AB) ).
```

The properties of commutativity and associativity allow us to transform a chain of calls into an equivalent chain of calls of which only one is required by the mode constraints of the procedure to be after the recursive call. For example the following two definitions of `sum` are equivalent and the one on the right is in a form suitable for our algorithm.

```
sum([], 0).
sum([H|T], S) :-
    sum(T, S0),
    plus(H, S0, S1),
    plus(H, S1, S).
```

```
sum([], 0).
sum([H|T], S) :-
    plus(H, H, TwoH),
    sum(T, S0),
    plus(TwoH, S0, S).
```

For predicates such as `mirror` in which the code after the recursive call includes construction unification as well, achieving tail recursion also requires the optimisation described in section 3.

2.2 Algorithm

The following definitions are needed to allow the presentation of the algorithm to be as concise as possible. The operator, $\triangleright$, has the following meaning: $C_b \triangleright C_a$ is true iff the goal C_b depends on the goal C_a, which means that C_a changes the state of instantiation of some variable that occurs in C_b. This means that C_a and C_b must appear in the program in their original order; they cannot have their order swapped. The operator, $\triangleright^+$, is the transitive closure of $\triangleright$. The function $nonlocals(G)$ returns the set of variables of the goal, G, that are also used elsewhere in the predicate.

The initial stage of the algorithm, shown in Figure 1, partitions the goals following the final recursive call in the recursive case into five sets: the set of goals which can be safely moved before the recursive call (*before*), and four sets of goals which cannot. The first of these sets (*assoc*) contains only calls to associative predicates; the second and third (*construct* and *construct_assoc*) contain only construction unifications, with *construct_assoc* containing the constructions that depend on members of the *assoc* set and *construct* containing the constructions that do not; and the fourth set (*reject*) contains everything else. The algorithm goes on to the second stage only if the *reject* set is empty.

The second stage of the algorithm, shown in Figure 2, is responsible for ensuring that the transformation does not worsen the big-O computational complexity of the predicate. It checks the complexity doesn't worsen using the heuristic from section 2.1, if the heuristic reports it does worsen then the algorithm terminates. It also swaps the order of the arguments to all the associative calls and determines the substitutions that will be used to mimic the unfold/fold process that was used as the justification of the algorithm.

Finally, the third stage, shown in Figure 3, creates the accumulator version of the predicate by applying the substitutions determined from stage 2. It also redefines the original predicate to call the accumulator version.

2.3 Example

As an aid to understanding the algorithm, we will walk through the algorithm for the `mirror` predicate from section 2.1. First, we present the original code of the predicate annotated to show how the algorithm refers to its various parts:

```
mirror(L, M) :-  % Inv={} In={L} Out={M}
    ( % switch on L
        L => [H | T],              % C 1,1
        mirror(T, M0),             % C 1,2     In'={T} Out'={M0}
        Empty <= [],               % C 1,3
        ListH <= [H | Empty],      % C 1,4
        append(M0, ListH, M1),     % C 1,5
        M <= [H | M1]              % C 1,6
```

Let $p(Inv, In, Out)$:-
switch(
$\langle C_{1,1} \ldots C_{1,k-1}, p(Inv, In', Out'), C_{1,k+1} \ldots C_{1,m} \rangle$,
$\langle C_{2,1} \ldots C_{2,n} \rangle$
).
where $1 \leq k < m \wedge \neg(\exists l, l > k \wedge C_{1,l} = p(_))$

$reject = construct_assoc = construct = assoc = before = \emptyset$
for $i = k+1$ to m:
if $(C_{1,i} \triangleright C_{1,j} \Rightarrow (1,j) \in (\{(1,1) \ldots (1,k-1)\} \cup before))$
$before = before \cup (1,i)$
else if ($C_{1,i}$ is an associative call such that
(its invariant input arguments depend only on Inv
$\wedge$ exactly one of its two associative input arguments occurs in Out')
$\wedge C_{1,i} \triangleright C_{1,j} \Rightarrow (1,j) \in (\{(1,1) \ldots (1,k)\} \cup before))$
$assoc = assoc \cup (1,i)$
else if ($C_{1,i}$ is a construction unification
$\wedge C_{1,i} \triangleright C_{1,j} \Rightarrow (1,j) \in (\{(1,1) \ldots (1,k)\} \cup before \cup construct))$
$construct = construct \cup (1,i)$
else if ($C_{1,i}$ is a construction unification
$\wedge C_{1,i} \triangleright C_{1,j} \Rightarrow (1,j) \in (\{(1,1) \ldots (1,k)\} \cup before \cup assoc \cup construct_assoc)$
$\wedge C_{1,i} \triangleright^{+} C_{1,j} \Rightarrow (1,j) \in (\{(1,1) \ldots (1,k)\} \cup before \cup \{C_{1,l}\} \cup construct_assoc$
where ($C_{1,l} \in assoc \wedge$
$C_{1,i}$ can be expressed as a call to the same predicate as $C_{1,l}$))
$construct_assoc = construct_assoc \cup (1,i)$
else
$reject = reject \cup (1,i)$
if ($reject \neq \emptyset$)
transformation is not applicable, exit

Fig. 1. The first stage

```
    ;
        L => [],                    % C 2,1
        M <= []                     % C 2,2
    ).
```

Note that the order of the arms of a switch does not matter, since exactly one of the arms will be taken. Note also that although in this example, all the conjuncts are calls or unifications, we do not require this in general; our algorithm would work just fine if e.g. $C_{1,6}$ were replaced with a compound goal (e.g. a disjunction).

We now present the values of the variables computed by our algorithm at the end of each stage of the transformation:

Stage 1:

$before$	$= \{(1,3),(1,4)\}$	$assoc$	$= \{(1,5)\}$
$construct$	$= \{\}$	$construct_assoc$	$= \{(1,6)\}$
$reject$	$= \{\}$		

Stage 2:

$before$	$= \{(1,1),(1,3),(1,4)\}$	$after$	$= \{(1,5),(1,6)\}$
$init_accs$	$= \{ListH, H\}$		
acc_var_subst	$= \{ListH \rightarrow A_ListH, H \rightarrow A_H\}$		
$during_assoc_input$	$= M0$	inv_input	$= \{\}$
$before_assoc_input$	$= ListH$	$assoc_output$	$= M1$
acc_var	$= A_ListH$	new_acc	$= New$
$CS_{1,5}$	$= append(ListH, M0, M1)$		
$assoc_call_subst$	$= \{M0 \rightarrow A_ListH, M1 \rightarrow NewListH\}$		
rec_call_subst	$= \{M0 \rightarrow M1, ListH \rightarrow NewListH\}$		

$before = before \cup \{(1,1) \ldots (1,k-1)\}$
$after = assoc \cup construct \cup construct_assoc$
$init_accs = \bigcup_{i \in before} nonlocals(C_i) \cap \bigcup_{i \in after} nonlocals(C_i)$

$acc_var_subst = \emptyset$
for each $var \in init_accs$
    create a new variable, A_var
    $acc_var_subst = \{(var \rightarrow A_var)\} \cup acc_var_subst$

$increase_complexity = false$
$rec_call_subst = \emptyset$
$assoc_call_subst = \emptyset$
for each $a \in assoc$
    let inv_input
        be the set of invariant input arguments in C_a
    let $during_assoc_input$
        be the associative input argument in C_a that is in Out'
    let $before_assoc_input$
        be the associative input argument in C_a that is not in Out'
    let $assoc_output$
        be the associative output argument in C_a

    let acc_var be the unique variable such that
        $(before_assoc_input \rightarrow acc_var) \in acc_var_subst$

    let CS_a be C_a with $during_assoc_input$ and $before_assoc_input$
        having their positions swapped
    if ($during_assoc_input$ is in an inefficient position in CS_a)
        $increase_complexity = true$

    create a new variable, new_acc
    $assoc_call_subst = \{during_assoc_input \rightarrow acc_var,$
        $assoc_output \rightarrow new_acc\} \cup assoc_call_subst$
    $rec_call_subst = \{during_assoc_input \rightarrow assoc_output,$
        $before_assoc_input \rightarrow new_acc\} \cup rec_call_subst$

if ($increase_complexity$)
    transformation would increase big-O complexity, exit

Fig. 2. The second stage

Stage 3:

$call_to_head = \{M0 \rightarrow M\}$ $\quad head_to_call = \{M \rightarrow M0\}$

$p_acc = mirror_acc$

$call = mirror_acc(T, M0, ListH, H)$

$base_call = mirror_acc(T, M, ListH, H)$

$rec_call = mirror_acc(T, M1, NewListH, H)$

$R_{1,5} = append(ListH, A_ListH, NewListH)$

$R_{1,6} = M <= [A_H|M1]$

$B_{2,1} = L => []$ $\quad B_{2,2} = M0 <= []$

$B_{1,5} = append(M0, A_ListH, M1)$

$B_{1,6} = M <= [A_H|M1]$ $\quad Acc = \{A_ListH, A_H\}$

$call_to_head = \emptyset$
$head_to_call = \emptyset$
for each $CV \in Out'$
  let HV be the variable from Out corresponding to CV in Out'
  $call_to_head = \{CV \rightarrow HV\} \cup call_to_head$
  $head_to_call = \{HV \rightarrow CV\} \cup head_to_call$

create new predicate, p_acc
$call = p_acc(Inv, In', Out', init_accs)$
$base_call = rename_goal(call, call_to_head)$
$rec_call = rename_goal(call, rec_call_subst)$
let C_{before} be the sequence $C_{1,j}$ where $(1, j) \in before$

redefine p as $p(Inv, In, Out)$:-
  $switch($
    $\langle C_{before}, base_call\rangle,$
    $\langle C_{2,1} \ldots C_{2,n}\rangle$
).

for each $(1, j) \in assoc$
  $R_{1,j} = rename(CS_{1,j}, assoc_call_subst)$
for each $(1, j) \in construct \cup construct_assoc$
  $R_{1,j} = rename(C_{1,j}, acc_var_subst)$
for $i = 1$ to n
  $B_{2,i} = rename_goal(C_{2,i}, head_to_call)$
for each $(1, j) \in assoc \cup construct \cup construct_assoc$
  $B_{1,j} = rename_goal(C_{1,j}, acc_var_subst)$

$Acc = rename_vars(init_accs, acc_var_subst)$
let R_{assoc} be the sequence $R_{1,j}$ where $(1, j) \in assoc$
let $R_{construct}$ be the sequence $R_{1,j}$ where $(1, j) \in construct \cup construct_assoc$
let B_{after} be the sequence $B_{1,j}$ where
  $(1, j) \in assoc \cup construct \cup construct_assoc$

define p_acc as
  $p_acc(Inv, In, Out, Acc)$:-
    $switch($
      $\langle C_{before}, R_{assoc}, rec_call, R_{construct}\rangle,$
      $\langle B_{2,1} \ldots B_{2,n}, B_{after}\rangle$
).

Fig. 3. The third stage

The final result of the algorithm is these two definitions:

```
mirror(L, M) :-
    ( % switch on L
        L => [H | T],
        Empty <= [],
        ListH <= [H | Empty],
        mirror_acc(T, M, ListH, H)
    ;
        L => [],
        M <= []
    ).

mirror_acc(L, M, A_ListH, A_H) :-
    ( % switch on L
        L => [H | T],
        Empty <= [],
        ListH <= [H | Empty],
        append(ListH, A_ListH, NewListH),
        mirror_acc(T, M1, NewListH, H),
        M <= [A_H | M1]
    ;
        L => [],
        M0 <= [],
        append(M0, A_ListH, M1),
        M <= [A_H | M1]
    ).
```

3 Last Call Modulo Constructors Optimization

If a predicate that would be tail recursive in Prolog is not tail recursive in Mercury, the reason is usually that the current Mercury implementation returns all output arguments of a call in registers. If an output argument needs to be placed somewhere in memory, this must be done after returning from the call.

For example, consider the recursive clause of `append(in, in, out)`:

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

In Prolog, *last call optimization* [1] would ensure that this clause is tail-recursive. In Mercury, however, the recursive call to `append` would return `NT` in a register which then needs to be copied into the correct memory location (i.e. the second argument of the cons cell). Actually, with the current release of Mercury, the situation is even worse. The mode checker does not allow constructions of partially-instantiated data structures. This means that the mode checker will actually delay the entire construction unification `C = [H | NT]` until after the recursive call.

Somogyi, Henderson and Conway [13] briefly outline a solution to these problems: reorder the code so that the construction unifications come before the call, then create a new version of the called predicate which returns the relevant arguments in memory and pass it the addresses in which to place the results.

We have taken this outline and used it as the basis for a new optimization pass in the Mercury compiler which we call the *last call modulo constructors* optimization (LCMC).[1] Before describing the optimization itself we describe changes that were required to the mode system, code generator and abstract machine to support it.

3.1 Mode System, Abstract Machine and Code Generation

We start with a brief overview of the mode system as implemented in the Melbourne Mercury Compiler, release 0.9 [16]. The instantiation state (*inst*) of a variable is one of the following:

`free`
: A variable having instantiation state `free` is an unbound variable with no aliases (i.e. the variable is not constrained to be part of another data structure).

`ground(uniqueness)`
: A variable having instantiation state `ground(uniqueness)` is bound to a ground term whose structure is unknown. The `uniqueness` field describes the number of references to the variable.

`bound(uniqueness, functors)`
: A variable having instantiation state `bound(uniqueness, functors)` is bound to a term whose top-level functor is in the set `functors`. The `functors` field contains, for each possible functor that the variable may be bound to, the name of the functor and the instantiation states of its arguments.

`any(uniqueness)`
: A variable having instantiation state `any(uniqueness)` is a constrained variable.

`not_reached`
: A variable having instantiation state `not_reached` is an output from a goal that cannot succeed.

`defined_inst(inst_name)`
: A variable having instantiation state `defined_inst(inst_name)` has a named instantiation state which may be looked up in the *inst table* for the procedure. This is used for describing recursive insts.

The *mode* of a predicate argument is a mapping from its initial inst to its final inst. The two most common argument modes, `ground -> ground` and `free -> ground`, are abbreviated `in` and `out`, respectively.

The *instmap* data structure maps variables to insts. Mode analysis uses abstract interpretation to determine the instmap at each point in a goal and stores the changes to the instmap across a goal in an *instmap delta*. Mode analysis

[1] This term comes from a similar idea described by Wadler [17] which he calls *tail recursion modulo cons*.

also schedules the goals within a conjunction, re-ordering them if necessary, and selects which mode to use for each predicate call.

Mercury's mode system requires that precise information about the insts of all variables be known at all points in the execution of a goal. Use of partially instantiated data structures, which is required for LCMC, creates aliasing between variables which we need to keep track of if we are to determine how further instantiation of one variable affects the variables to which it is aliased.

Andrew Bromage has recently extended the mode system to allow the tracking of definite aliases. He introduces a new kind of inst, `alias(inst_key)`, where two variables are definitely aliased iff their insts are both `alias` specifying the same `inst_key`. The actual inst of the variables is determined by looking up the current instantiation of the `inst_key` in an *inst key table*.

At the abstract machine level, a variable whose instantiation after expansion of aliases is `ground` or `bound` is represented by a word containing its value, while a variable whose inst is `free` is represented by an uninitialized word. This is not sufficient for our needs. We have therefore added a new basic inst, `free_alias`, which is similar to `free` except that a variable whose inst is `free_alias` is represented by a word that contains the address where its value will be placed when the variable is instantiated.

During mode analysis, uninstantiated parts of a partially instantiated data structure have the inst `free_alias` rather than `free`. In most places during mode analysis, `free` and `free_alias` insts are treated the same. The exception to this is in the unification of two free variables. While it is still a mode error to attempt to unify two `free` variables, as in previous versions of Mercury, two variables whose insts are `free` and `free_alias` may now be unified; after the unification, both will have the inst `free_alias`. We do not allow the unification of two variables whose insts are both `free_alias`. This is because this would require a `free_alias` variable to record the address of more than one memory location to be filled in when the variable is instantiated. While this could be implemented, using either terminated pointer chains as in the WAM [1], or circular pointer chains as in PARMA [15], the overhead required is very likely to outweigh the benefits gained by LCMC. (Note that the PARMA scheme has actually been implemented for Mercury by the HAL project [10].)

An argument of a procedure whose mode is `free_alias -> ground` will be passed by reference. The code generator will reserve an input register for this argument, and it will place the address of the memory location to be filled in in this register before calling the procedure. When a variable whose inst is `free_alias` is bound, the code generator emits code to place its value in the memory location that the representation of the `free_alias` variable points to.

3.2 The LCMC Transformation

The LCMC optimization itself is a very simple transformation. The LCMC pass occurs after other passes that may increase the chances of being able to apply LCMC. These include simplification, inlining and accumulator introduction. At this stage, mode analysis has selected an initial scheduling order for each procedure which avoids construction of partially instantiated terms.

LCMC traverses the goal for each procedure in turn. A procedure is determined to be a candidate for LCMC if any path through the goal contains a final

conjunction where the last call in the conjunction is followed only by construction unifications. The algorithm used is described more formally in Figure 4.

Procedure: LCMC()
Input: G_{in}, the goal of the procedure to be transformed
Output: G_{out}, the transformed goal

if
  G_{in} is a disjunction of sub-goals, $disj(G_1, \ldots, G_n)$
then
  $G_{out} = disj(\text{LCMC}(G_1), \ldots, \text{LCMC}(G_n))$
else if
  G_{in} is a switch of sub-goals, $switch(G_1, \ldots, G_n)$
then
  $G_{out} = switch(\text{LCMC}(G_1), \ldots, \text{LCMC}(G_n))$
else if
  G_{in} is an if-then-else, $ite(G_C, G_T, G_E)$, with condition G_C,
  then goal G_T, and else goal G_E
then
  $G_{out} = ite(G_C, \text{LCMC}(G_T), \text{LCMC}(G_E))$
else if
  G_{in} is a conjunction of sub-goals, $conj(G_1, \ldots, G_n)$
then
  if
    $\exists k.\ 1 < k < n \wedge G_k$ is a call
    $\wedge\ \forall i.\ k < i \leq n \Rightarrow G_i$ is a construction unification
  then
    let $G_{PBR} = \text{PBR}(G_k, \langle G_{k+1}, \ldots, G_n \rangle)$
    $G_{out} = conj(G_1, \ldots, G_{k-1}, G_{k+1}, \ldots, G_n, G_{PBR})$
  else
    $G_{out} = conj(G_1, \ldots, G_{n-1}, \text{LCMC}(G_n))$
else
  $G_{out} = G_{in}$

Fig. 4. The LCMC transformation

For all conjunctions where LCMC can be applied, the conjunction is reordered so that the constructions appear before the call. Now the required mode for the called predicate will be different from the previously-determined mode. The PBR() procedure in Figure 5 determines which output arguments of the call become pass-by-reference. These arguments will have an initial instantiation of `free_alias` rather than `free`. LCMC tries to find an existing procedure of the called predicate that matches the mode required. If found, the call is replaced by a call to this procedure. If it is not found, it will need to be created. This is done by taking a copy of the existing procedure and changing the modes of its arguments. The new procedure then has LCMC run on it. If the called predicate is the same as the calling predicate, this will result in a tail-recursive procedure.

Procedure: PBR()
Input: G_{in}, a call goal, $call(P, V, M)$
where P is the predicate being called,
$V = \langle V_1, \ldots, V_k \rangle$ the argument variables of the call,
$M = \langle M_1, \ldots, M_k \rangle$ the modes of the arguments,
and k the arity of P.
$\langle G_1, \ldots, G_n \rangle$, a sequence of construction unification goals
Output: G_{out}, a goal which calls a new pass-by-reference procedure

for $i = 1$ to k
do
if
$\exists I.\ M_i =$ `free -> ` I
$\wedge$ ($I =$ `ground` $\vee\ I =$ `bound(_, _)`)
$\wedge\ V_i$ appears on the right hand side of one of the
construction unifications $G_1, \ldots, G_n$
then
$M'_i =$ `free_alias -> ` I
else
$M'_i = M_i$
done
$M' = \langle M'_1, \ldots, M'_k \rangle$
$G_{out} = call(P, V, M')$

Fig. 5. The PBR procedure

After running LCMC on a procedure, it is mode checked to update the instmap deltas and to ensure that it is mode correct. This includes checking that there are no unifications involving two `free_alias` insts.

Let us look at how the transformation is applied to the procedure `append(in, in, out)` which we will refer to as mode 0 of the predicate `append/3`. The HLDS representation for this procedure, before LCMC, is shown below. The comments show the change in instantiation state of the variables after each sub-goal (the `uniqueness` field is left out since it is not relevant to LCMC).

```
% mode 0 of append/3
:- mode append(in, in, out).

append(Xs, Ys, Zs) :-
    % Initial insts: Xs -> ground, Ys -> ground, Zs -> free
    (
        Xs => [],
        Zs := Ys                % Zs -> ground
    ;
        Xs => [X | Xs1],        % X -> ground, Xs1 -> ground
        append(Xs1, Ys, Zs0),   % call mode 0 of append/3
                                % Zs0 -> ground
        Zs <= [X | Zs0]         % Zs -> bound([ground | ground])
    ).                          % Zs -> ground
```

LCMC discovers that in the second arm of the switch, construction of `Zs` can be moved before the recursive call. `Zs0` now has an inst of `free_alias` before the call so LCMC tries to find a mode of `append/3` that matches this inst. The required mode is not found so it is created (mode 1) by copying mode 0 and changing the initial inst of `Zs`. At this stage, the HLDS for mode 0 is:

```
% mode 0 of append/3
:- mode append(in, in, out).

append(Xs, Ys, Zs) :-
    % Initial insts: Xs -> ground, Ys -> ground, Zs -> free
    (
        Xs => [],
        Zs := Ys                % Zs -> ground
    ;
        Xs => [X | Xs1]         % X -> ground,  Xs1 -> ground
        Zs <= [X | Zs0]         % Zs -> bound([ground | alias(IK0)])
                                % Zs0 -> alias(IK0)
                                % IK0 -> free_alias
        append(Xs1, Ys, Zs0)    % call mode 1 of append/3
                                % Zs -> bound([ground | alias(IK1)])
                                % Zs0 -> alias(IK1)
                                % IK1 -> ground
    ).                          % Zs -> ground
```

LCMC is now run on the new procedure, mode 1. When it re-orders the call and construction it finds that an appropriate mode of `append/3` already exists (i.e. mode 1). The HLDS for mode 1 looks the same as mode 0 except that the mode of the third argument is `free_alias -> ground` rather than `out` (`free -> ground`).

Finally, we show the C code generated by the compiler for the tail recursive mode 1 of `append`. (We have made the code more readable by e.g. shortening label names and removing type casts.) The arguments `Xs`, `Ys` and `Zs` are passed in registers `r1`, `r2` and `r3`, respectively.

```
/* code for predicate 'append'/3 in mode 1 */
Define_static(append_3_1);
    if (r1 == mkword(mktag(0), mkbody(0)))
        GOTO_LABEL(append_3_1_i2);
Define_label(append_3_1_i4);
    tag_incr_hp(r4, mktag(1), 2);
    field(mktag(1), r4, 0) = field(mktag(1), r1, 0);
    *r3 = r4;
    r3 = &field(mktag(1), r4, 1);
    r1 = field(mktag(1), r1, 1);
    if (r1 != mkword(mktag(0), mkbody(0)))
        GOTO_LABEL(append_3_1_i4);
Define_label(append_3_1_i2);
    *r3 = r2;
    proceed();
```

The code first checks whether `Xs` (in `r1`) is the empty list, and branches to label `append_3_1_i2` if it is. If it is not, it falls through to the code for the recursive case. This allocates a new two-word cell on the heap, and places a pointer to this cell tagged with the tag value for cons (1), which represents the value of `Zs`, in `r4`. It places the value of the `Zs` where the caller expects it, which is the memory location whose address the caller passed in `r3`. It fills in the first word of the new cell with `X`, and puts the address of the second into `r3`, to tell the recursive call where to put the value of `Zs0`. Extracting `Xs1` from the second field of the cons cell of `Xs` and putting it into `r1` completes the setup for the recursive call, since `Ys` is already in `r2`. The recursive call is an unconditional jump to the start of the procedure, which low-level optimizations in the Mercury compiler turn into the conditional jump to the start of the recursive case, as shown above. The code of the base case implements the assignment `Zs := Ys` by copying `Ys` to where the caller expects `Zs`.

The LCMC optimization does not guarantee a tail call in every procedure it transforms. The transformed call will be a tail call if and only if it returns exactly the same sequence of outputs in exactly the same locations as its containing procedure. If it does not, the call must be followed by code that places the output arguments of the containing procedure in their proper places. In such cases, the optimization may still be useful as it may allow a tail call somewhere in the new pass-by-reference procedure by reducing the number of output registers of the procedure. This is seen in the `append` example where the call from mode 0 to mode 1 is not a tail call because mode 0 needs to save the value of `Zs` on the stack across the call and then place it in a register before it returns. However, the recursive call from mode 1 to itself is tail recursive because it can place `Zs` in the required location in memory as soon as it is constructed, so there is nothing that needs to be done after the recursive call.

Since the compiler runs LCMC recursively on each new procedure at the point where it is created, we can check whether the optimization is actually going to give us a tail call somewhere in the new procedure and, if not, abandon the optimization for this procedure.

The overall effect is that although our optimization does not depend on the last call being a recursive call, the last call will be a recursive call in most of the situations in which our optimization can be usefully applied.

The `mirror_acc/4` predicate that was created by the transformation in section 2 can also be transformed into tail recursive form by the LCMC transformation. Thus, the original `mirror/2` predicate, which could not be made tail recursive by either optimization by itself, can be made tail recursive by combining the two optimizations.

4 Results

To measure the effectiveness of our optimizations, we benchmarked a set of small programs on a PC with 450 MHz Intel PIII CPU, 512 KB of L2 cache, 256 MB of memory, running Linux 2.2.12. The benchmark programs all process lists, and they were tested with their input being a list of 30 integers. Their executions were repeated several thousand times to eliminate the uncertainties involved in

measuring small time intervals. The results are presented in the following table in the form of normalized runtimes; the raw data is available from our web site.

The table shows two sets of results: without and with garbage collection. Within each set, the columns show performance without either of the optimizations we presented in this paper, with just LCMC, with just accumulator introduction, and with both. In all four cases, all the other optimizations of the Mercury compiler were turned on with the exception of unused argument elimination, because unused argument elimination makes relating a call back to a promise using that call difficult. (We intend to move unused argument elimination after the tail recursion optimizations, but this requires first modifying its implementation to accept the modes produced by LCMC.)

Table 1. Normalised Benchmark Results

	without gc				with gc			
	None	LCMC	Acc	Both	None	LCMC	Acc	Both
append	100.0	90.4	100.0	90.7	100.0	97.6	99.4	97.7
sumlist	100.0	99.9	80.3	80.2	100.0	99.9	79.2	79.2
nrev	100.0	74.1	13.1	13.9	100.0	96.2	12.5	12.5
sumdbl	100.0	99.9	99.9	90.0	100.0	99.2	99.2	98.6
mirror	100.0	72.9	100.0	10.3	100.0	95.7	99.2	9.5

As is to be expected, `append` and `sumlist` only achieve a speed increase when the correct optimization is enabled. Naive reverse (`nrev`) benefits from both optimizations. LCMC makes the call to `append` tail recursive, which improves the performance of its caller `nrev`. Accumulator introduction allows a speed increase by reducing the big-O complexity of the algorithm (effectively turning it into non-naive reverse) and by making the `nrev` predicate tail recursive. However turning on LCMC after turning on accumulator introduction on slows the code down slightly because in the accumulator version of nrev, the first argument of append is always a list of length one, and the fixed overhead of the LCMC transformation thus cannot be compensated for by LCMC making `append` tail recursive. The predicate `sumdbl`, which in one traversal both sums a list of integers and returns another list which has each element doubled is a case where you don't get any significant speedups unless both optimizations are turned on. (Such predicates can be expected to be produced by deforestation.) The predicate `mirror` from section 2.1 also requires both optimizations to turned on for it to become tail recursive. However, mirror can also benefit from LCMC alone because of its use of `append`.

At the moment, Mercury uses the Boehm conservative garbage collector for C [4]. Using this collector requires a function call on every memory allocation, whereas in the no-gc case, Mercury allocates memory by simply incrementing the heap pointer. (The Boehm collector also lacks provision for efficiently recovering memory without garbage collection, e.g. between different repetitions of the same test.) Since memory allocation accounts for a bigger fraction of the runtime of the program with gc than without gc, we get smaller relative speedups with

gc than with without gc for optimizations that do not eliminate memory allocations themselves. LCMC cannot eliminate memory allocations. Accumulator introduction can eliminate memory allocations, and will do so whenever it reduces the big-O complexity of a predicate that allocates memory every iteration. The `nrev` and `mirror` predicates both benefit from this effect.

5 Related Work

As stated earlier the accumulator introduction transformation was based on Debray's work [8]. However Debray only showed how the transformation works with predicates which have the additional property of commutativity. Thus there is no need for the associative call to have its arguments rearranged, nor the introduction of the heuristic to ensure the big O complexity never increases. Brough and Hogger [5] allude to transformation not necessarily improving the efficiency, but only state that applicability of the transform should be related to the knowledge of the calling modes of each predicate.

The other significant facet of the accumulator introduction transformation is the possibility of leaving a predicate in a state which doesn't make the predicate tail recursive but allows the LCMC optimization to be applied.

Optimizations similar to LCMC have often been suggested in the literature of both functional [17] and logic [13] programming. Most of these suggestions have not been implemented. The major exceptions are Prolog systems in which the issues involved in implementing LCMC are completely different.

One paper we know of which addresses the issue in the context of a moded logic programming language is Bigot and Debray [2]. They describe an algorithm for deciding when to return output values in registers and when to return them in memory. However, their technique requires profiling information (the number of times each procedure is called from each one of its call sites) which the Mercury profiler cannot yet supply, and in any case, their algorithm would require significant extensions to handle separate compilation. Our implementation of LCMC probably captures most of the benefit with much less implementation complexity.

6 Conclusion

We have presented two optimizations that can improve the operational efficiency of a program. The optimizations are applicable when only independent associative goals and/or construction unifications occur after the recursive call. Both optimizations can be automatically applied and have been implemented in the Melbourne Mercury compiler. We have also shown that both optimizations can be combined to optimize a program which neither optimization would optimize by itself.

The accumulator introduction optimization is included with the current stable release, Mercury 0.9. The LCMC optimization is implemented on a separate branch of the compiler source tree which contains the alias tracking analysis which is still in an experimental stage. The source code is available upon request.

We thank the anonymous referees for their valuable comments, the attendees at the LOPSTR conference for their stimulating discussions, and the Australian Research Council for their support.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[2] Peter A. Bigot and Saumya Debray. Return value placement and tail call optimization in high level languages. *Journal of Logic Programming*, 38(1):1–29, January 1999.

[3] Charlene Bloch. Source-to-source transformation of logic programs. Master's thesis, Weizmann Institue of Science, November 1984.

[4] Hans-Juergen Boehm. Dynamic memory allocation and garbage collection. *Computers in Physics*, 9(3):297–303, May/June 1995.

[5] D. R. Brough and C. J. Hogger. Compiling associativity into logic programs. *Journal of Logic Programming*, 4(4):345–359, December 1987.

[6] D. R. Brough and C. J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9:115–134, 1991.

[7] Halime Büyükyildiz and Pierre Flener. Generalised logic program transformation schemas. In N.E. Fuchs, editor, *Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation*, pages 45–65, 1997.

[8] Saumya K. Debray. Optimizing almost-tail-recursive Prolog programs. In *Proceedings of the International Symposium on Functional Programming Languages and Computer Architecture*, pages 204–219, 1985.

[9] Saumya K. Debray. Unfold/fold transformations and loop optimization of logic programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 297–307, 1988.

[10] Bart Demoen, Maria Garcia de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. Herbrand constraint solving in HAL. In *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 260–274, 1999.

[11] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Australia, 1996.

[12] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[13] 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.

[14] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. In *Proceedings of the Fourth International Static Analysis Symposium*, pages 157–171, 1997.

[15] Andrew Taylor. *High Performance Prolog Implementation*. PhD thesis, University of Sydney, 1991.

[16] Mercury Team. Mercury 0.9. Available from http://www.cs.mu.oz.au/mercury/, December 1999.

[17] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1984.