State update transformation *

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

ABSTRACT
Recursive predicates frequently generate some state which is updated after the recursive call. We present a source to source transformation which can move the state update before to the recursive call, thus helping to make the predicate tail recursive, and report on its implementation in the Mercury compiler.
Keywords: state update, accumulator introduction, program transformation, schema, logic programming, Mercury

1. INTRODUCTION
Recursive calls are often very close to being the last action in a clause. Frequently, however, the recursive call computes and returns a value which must be put into a memory cell or used as input to another goal. Since the Mercury execution algorithm [10] 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.

Previous work has shown that a simple goal (e.g. a call) can be moved prior to the recursive call by the introduction of accumulators, providing the simple goal has the property of associativity [9, 6, 3, 5]. The need to place a value into a memory cell after a recursive call can also be avoided by using call-by-reference [9]. (This second optimization is specific to Mercury and other non-WAM-based systems, because the WAM representation of variables effectively requires variables to be passed by reference).

A predicate which inserts a list of elements into a set of elements can use either accumulator recursion (tail-recursive) or naive recursion (non-tail-recursive). Thus there must be some transformation from the naive form to the accumulator form. However the set insertion predicate doesn't have the property of associativity, so we must use a different transformation. The property that the set insertion predicate has that allows it to be written in both the accumulator and naive forms is that its state (the set) is updated in an order-independent way. In other words, it doesn't matter what order we insert elements into the set, as long as we insert each element into the set at some point.

This paper presents an optimization that looks for simple goals having the order-independent state update property after the recursive call. The optimization then creates a specialized version of the predicate which has the code to update the state before the recursive call instead. This specialized predicate will have extra arguments that act as *accumulators*: they hold the state of the computation so far. The optimization has been implemented in the Melbourne Mercury compiler.

The rest of this paper is organized as follows. The rest of this section presents some background. Section 2 describes the transformation. Section 3 presents benchmarking results. Section 4 discusses how our optimization is related to previous work. Section 5 concludes the paper.

1.1 Background
The Mercury language is described in full in the reference manual [8], and the main aspects of the implementation in [10]. 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

*This is not the final version of this paper. Please check `<http://www.cs.mu.oz.au/mercury/>` for the latest version of this paper.

(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. UPDATE TRANSFORMATION

This section introduces the source to source transformation that is the subject of this paper. This transformation looks for self-recursive predicates where the recursive call is followed by a computation which updates a state variable in a way that is order-independent, and then attempts to move this computation before the recursive call. Provided that all other computations after the recursive call can also be moved before the recursive call, either by this transformation or by some other (e.g. [9]), this will make the predicate tail recursive.

2.1 Order-independent state update

The transformation obviously needs to know what it means for a predicate to update state in an order-independent manner. In the Mercury implementation, the transformation does this by looking for a *promise* declaration of the form

```
:- promise all [I,A,B,S0,S]
    (
        (some [SA] p(I, A, S0, SA), p(I, B, SA, S))
    <=>
        (some [SB] p(I, B, S0, SB), p(I, A, SB, S))
    ).
```

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 declarations follows the pattern associated with order-invariant updates. It talks about a predicate `p` whose arguments can be divided into

- a possibly empty set of invariant arguments, (in this case, the first argument),
- a nonempty set of arguments whose role is to specify the update to be performed (in this case, the second argument),
- exactly one argument that represents the initial state of the data structure (in this case, the third argument), and
- exactly one argument that represents the final state of the data structure (in this case, the fourth argument).

The pattern requires the formula to be an equivalence, each arm of which is a conjunction of two calls to the same predicate, where

- the invariant arguments to have the same values in all four calls;
- the set of parameter arguments that occurs in the first call in the first arm of the equivalence must occur in the second call in the second arm of the equivalence, and vice versa;
- the two arms of the equivalence must have equal values in the initial state argument of the first call and in the final state argument of the second call; and
- the two arms of the equivalence have different variables, local to each arm of the equivalence, in the final state argument of the first call and in the initial state argument of the second call.

Any promise declaration that follows this pattern promises that it does not matter in what order the predicate concerned updates its state.

Many predicates which update the state of some data structure can have such a promise made for them: the two canonical examples are inserting items into sets and inserting key-value pairs into dictionaries. However, programmers must be aware of the distinction between syntactic equality (which is often called structural equality) and semantic equality; in some cases the two are the same, but in other cases they are not. As far as the programmer is concerned, the result of inserting two elements into a set should be independent of the order of insertion, i.e. the two resulting sets are guaranteed to be semantically equal. If sets are represented by sorted lists, the resulting sets will be structurally equal as well. However, if sets are implemented as unsorted lists, as binary search trees, or as B-trees, different insertion orders yield lists or trees that logically contain the same elements but are (or at least can be) structurally different.

For a promise declaration asserting order-independent update, the values in the final state arguments of the two arms of the equivalence must be equal *with respect to the equality axioms associated with the type of those arguments.* Mercury uses structural equality by default, but programmers can override this: when they define a new type, they can specify the predicate that implements equality testing on values of that type. With respect to values of that type, the equality axioms are given by the definition of that predicate rather than the usual rules implementing structural equality. For example, when sets are represented as sorted trees (of any variety), the ideal equality predicate would perform a preorder traversal on both arguments simultaneously, failing when and if it finds that the two traversals produce different lists.

2.2 Transformation

We will illustrate our transformation with the predicate, `map_list_to_set`, which takes a list of value of some type `T`, applies a function, `F`, to map each `T` into some `U` and then inserts the `U` into a set of `U`. The code a human programmer would write in Mercury is:

```
:- pred map_list_to_set(func(T) = U,
        list(T), set(U)).
:- mode map_list_to_set(func(in) = out is det,
        in, out) is det.

map_list_to_set(F, [], set__init).
map_list_to_set(F, [H | T], S) :-
    map_list_to_set(F, T, S0),
    set__insert(S0, F(H), S).
```

which is represented internally as follows:

```
map_list_to_set(F, L, S) :-
    ( % switch on L
        L => [],
        S := set__init
    ;
        L => [H | T],
        map_list_to_set(F, T, S0),
        FH := apply(F, H),
        set__insert(S0, FH, S).
    ).
```

A simple analysis would determine that the higher-order function application doesn't obey the order-independent state update formula, and hence the predicate cannot be transformed. However, the higher-order function application 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.)

After moving the higher-order function application, the only thing following the recursive call is a call which does obey an order-independent state update law, so the transformation can be applied. The goal of the transformation is to move the state update computation before the recursive call. This requires the recursive call to have an extra input argument, the accumulator, that contains the set of items that have been inserted before the recursive call. Since we want to keep the interface of `map_list_to_set` the same, this requires the creation of a new predicate, `map_list_to_set_acc`, whose arguments include all the arguments of `map_list_to_set` as well as the accumulator. (The accumulator has the same type as the state arguments of the state update call.)

Since all recursion will take place in the new predicate, the only task of the old one is to call the new one after initializing the accumulator. The initialization code is the part of the original base case that initialises the answer. The new definition of `map_list_to_set` is thus

```
map_list_to_set(F, L, S) :-
    Acc0 := set__init,
    map_list_to_set_acc(F, L, S, Acc0).
```

The new predicate `map_list_to_set_acc` has the same general structure as the original predicate, but the details of course differ. The recursive case updates the accumulator before the recursive call, instead of updating the state generated by the recursive call after the call. When computation reaches the base case, the accumulator will hold the result of the computation so far, so the base case assigns the value in the accumulator to the output variable. The resulting code is the following:

```
map_list_to_set_acc(F, L, S, Acc0) :-
    ( % switch on L
        L => [],
        S := Acc0
    ;
        L => [H | T],
        FH := apply(F, H),
        set__insert(Acc0, FH, Acc).
        map_list_to_set_acc(F, T, S, Acc)
    ).
```

2.3 Correctness

The informal presentation of the transformation in section 2.2 is an instance of the following schema.

```
p(In, Out) :-
    minimal(In),
    initialise(Out).
p(In, Out) :-
    decompose(In, Current, Rest),
    p(Rest, Out0),
    update(Current, Out0, Out).
```

Whenever a procedure matches the above schema, with the `minimal` and `decompose` goals being mutually exclusive, and `update/3` obeying the order-independent state update formula, the transformation generates the corresponding instance of the following schema:

```
p(In, Out) :-
    initialise(Acc),
    p_acc(In, Out, Acc).

p_acc(In, Out, Acc) :-
    minimal(In),
    Out = Acc.
p_acc(In, Out, Acc0) :-
    decompose(In, Current, Rest),
    update(Current, Acc0, Acc),
    p_acc(Rest, Out, Acc).
```

We wish to show that the two schemas compute the same answer. If the predicate `p` recurses N times, then unfolding its recursive clause yields the following schema:

```
p(In, Out) :-
    decompose(In, Current_{1}, Rest_{1}),
    ...
    decompose(Rest_{N-1}, Current_{N}, Rest_{N}),
    minimal(Rest_{N}),
    initialise(Out_{0}),
    update(Current_{N}, Out_{0}, Out_{1}),
    ...
    update(Current_{1}, Out_{N-1}, Out).
```

A similar unfolding performed on the transformed predicate's only clause yields:

```
p(In, Out) :-
    initialise(Acc_{0}),
    decompose(In, Current_{1}, Rest_{1}),
    update(Current_{1}, Acc_{0}, Acc_{1}),
    ...
    decompose(Rest_{N-1}, Current_{N}, Rest_{N}),
    update(Current_{N}, Acc_{N-1}, Acc_{N}),
    minimal(Rest_{N}),
    Out = Acc_{N}.
```

Since the decompose goals have their first argument as their only input, this can be reordered to the following schema without violating any mode constraints:

```
p(In, Out) :-
    decompose(In, Current_{1}, Rest_{1}),
    ...
    decompose(Rest_{N-1}, Current_{N}, Rest_{N}),
    minimal(Rest_{N}),
    initialise(Acc_{0}),
    update(Current_{0}, Acc_{0}, Acc_{1}),
    ...
    update(Current_{N}, Acc_{N-1}, Acc_{N}),
    Out = Acc_{N}.
```

The only difference between the two unfolded predicate definitions (modulo renaming and the final unification) is the order the state is updated. Given that the programmer has promised that the order in which the state is updated does not matter, the two schemas will compute semantically equal answers.

2.4 Integration with other transformations

To increase the applicability of the transformation, we wish to make sure that it can applied together with other transformations that can also remove goals after recursive calls. In particular, we want to integrate our new transformation with our previous transformation in [9], which relied on the goal being moved either having the property of associativity, or being a construction unification.

Suppose that the predicate in question matches the following schema, where compose/3 has the property of associativity:

```
p(In, OutUpdate, OutAssoc) :-
    minimal(In),
    initialise(OutUpdate),
    base(OutAssoc).
p(In, OutUpdate, OutAssoc) :-
    decompose(In, Current, Rest),
    p(Rest, OutUpdate0, OutAssoc0),
    update(Current, OutUpdate0, OutUpdate),
    compose(Current, OutAssoc0, OutAssoc).
```

Providing compose/3 is independent of update/3, that is neither goal depends on the result of the other goal, we can move update/3 before the recursive call. The only difficulty with this is that we must divide the base case into two classes of goals, those which are used to initialise the accumulators and those which are required to stay in the base case to initialise the answer for the compose/3 goals. This leads to the following schema for the t:

```
p(In, OutUpdate, OutAssoc) :-
    initialise(AccUpdate),
    p_acc(In, OutUpdate, OutAssoc, AccUpdate).

p_acc(In, OutUpdate, OutAssoc, AccUpdate) :-
    minimal(In),
    OutUpdate = AccUpdate,
    base(OutAssoc).
p_acc(In, OutUpdate, OutAssoc, AccUpdate0) :-
    decompose(In, Current, Rest),
    update(Current, AccUpdate0, AccUpdate),
    p_acc(Rest, OutUpdate, OutAssoc0, AccUpdate),
    compose(Current, OutAssoc0, OutAssoc).
```

The predicate p_acc is now in a form that our previous algorithm [9] can make tail recursive. This algorithm introduces a new recursive predicate and redefines p_acc to be an interface predicate like p. To minimize the overhead of procedure calls, the new p_acc should then be inlined inside p; the definition of p resulting from that inlining can itself be inlined into predicates that call p.

2.5 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 function $reorderable(G)$ returns a boolean value indicating whether or not the goal, G, can be safely left after the recursive call, as some other transformation will move it prior to the recursive call (e.g. [9]). Given a set of variables Vs and a list of ids of the goals Gs that make up the body of a clause that binds all the variables in Vs, the function $ids\_which\_instantiate(Vs, Gs)$ returns a subset of Gs consisting of the ids of the goals that contribute to the values of the variables in Vs. The id of a goal is therefore in $ids\_which\_instantiate(Vs, Gs)$ if the goal instantiates a variable in Vs, if the goal instantiates a variable that is required as input by some other goal in $ids\_which\_instantiate(Vs, Gs)$.

The initial stage of the algorithm, shown in Figure 1, partitions the goals following the final recursive call in the recursive case into four sets: the set of goals which can be safely moved before the recursive call (*before*), and three sets of goals which cannot. The first of these sets (*update*) contains only calls to predicates which update some state; the second set (*reorderable*) contains goals that can be moved before the recursive call by some other transformation; and the third set (*reject*) contains everything else. The algorithm goes on to the second stage only if the *reject* set is empty.

```
Let p(Inv, In, Out) :-
    switch(
        ⟨C_{1,1} ... C_{1,k-1}, p(Inv, In', Out'), C_{1,k+1} ... C_{1,m}⟩,
        ⟨C_{2,1} ... C_{2,n}⟩
    ).
    where 1 ≤ k < m ∧ ¬(∃l, l > k ∧ C_{1,l} = p(_))

reject = reorderable = update = before = ∅
for i = k + 1 to m:
    if (C_{1,i} ▷ C_{1,j} ⇒ (1, j) ∈ ({(1,1) ... (1,k-1)} ∪ before))
        before = before ∪ (1, i)
    else if (C_{1,i} is an update call
            ∧C_{1,i} ▷ C_{1,j} ⇒ (1, j) ∈ ({(1,1) ... (1,k)} ∪ before))
        update = update ∪ (1, i)
    else if (C_{1,i} ▷ C_{1,j} ⇒ (1, j) ∈ ({(1,1) ... (1,k)} ∪ before)
            ∧reorderable(C_{1,i}))
        reorderable = reorderable ∪ (1, i)
    else
        reject = reject ∪ (1, i)
if (reject ≠ ∅)
    transformation is not applicable, exit
```

Figure 1: The first stage

The second stage of the algorithm, shown in Figure 2, can be divided into two distinct sections. The first section is responsible for determining all the substitutions that will be applied to each update call.

Each update call will be changed to use accumulators. This requires that for each such call we define two new variables, the initial and final accumulators, and determine the corresponding input and output state variables.

We build two sequences of variables from these newly defined variables. (Conceptually, we build sets, but the variables in these sets will be put into atoms, and must therefore have an order imposed on them.) The first sequence, *accs*, is the sequence of final accumulator variables (one per update call), and is used to construct calls and clause heads. The second, *update_out_vars*, records the state output variables of all the update calls; it is used to determine which goals in the base case are needed to initialise the accumulators.

The relationship between each accumulator and state variable is stored in *update_subst*; this substitution is used to transform the update call to use accumulators. The relationship between each pair of accumulators is stored in *acc_var_subst* and the relationship between each pair of state variables is stored in *state_subst*; these substitutions are used to create the goals in the base case that transfer the values in the accumulators to the output variables.

The second section of stage 2 is responsible for determining which goals are needed to initialise the accumulators in the interface predicate, and which need to be placed in the base case of the accumulator version of the procedure. The goals which initialise the accumulators consist of all the goals in the base case of the original predicate that together determine the values of the state output variables. It is a bit more difficult to determine which goals need to be placed in the new base case. Usually it is all the goals in the original base case except the goals that have been moved to the interface predicate. However, a goal in the original base case may play a dual role; it may contribute to the value of both a state output variable and a non-state output variable. Such goals must be duplicated; they must appear both in the interface predicate and in the base case of the recursive predicate. This duplication does not affect the semantics of the program, but of course may affect its efficiency. However, such duplication is necessary only rarely, and even then it is likely to involve a cheap goal.

```
before = before ∪ {(1,1) ... (1,k-1)}

state_subst = acc_var_subst = update_subst = ∅
for each u ∈ update
    let state_input
        be the input argument in C_u that is in Out'
    let state_output
        be the output argument in C_u

    create a new variable, acc_zero
    create a new variable, acc

    accs= {acc} ∪ accs
    update_out_vars= {state_output} ∪ update_out_vars

    update_subst = {state_input → acc_zero,
        state_output → acc} ∪ update_subst
    acc_var_subst= {acc → acc_zero} ∪ acc_var_subst
    state_subst= {state_input → state_output} ∪ state_subst

update_base = ids_which_instantiate(update_out_vars,
    C_{{(2,1)...(2,n)}})
other_base = ({(2,1) ... (2,n)} − update_base)∪
    ids_which_instantiate(Out − update_out_vars,
        C_{{(2,1)...(2,n)}})
```

Figure 2: The second stage

The third stage, shown in Figure 3, redefines the original predicate to call the accumulator version, while the fourth stage, shown in Figure 4, creates the accumulator version of the predicate by applying the substitutions determined by stage 2.

2.6 Example

As an aid to understanding the algorithm, we will walk through the algorithm for the `map_list_to_set` predicate from section 2.2. First, we present the original code of the predicate annotated to show how the algorithm refers to its various parts:

$call\_to\_head = \emptyset$
$head\_to\_call = \emptyset$
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

for each $(2, j) \in update\_base$
    $I_{2,j} = rename(C_{2,j}, update\_subst)$
let I_{update_base} be the sequence $I_{2,j}$
    where $(2, j) \in update\_base$

redefine p as
    $p(Inv, In, Out)$:-
        I_{update_base},
        $p(Inv, In, Out, accs)$.

Figure 3: The third stage

$Acc0 = rename\_vars(accs, acc\_var\_subst)$
$InternalOut = rename\_vars(Out', state\_subst)$

for each $(1, j) \in update$
    $R_{1,j} = rename(C_{1,j}, update\_subst)$

$o = 1$
for each $O \in update\_out\_vars$
    let A be the unique variable such that
        $(O \rightarrow A) \in update\_subst$
    $B_{3,o} = rename(O := A, acc\_var\_subst)$
    $o = o + 1$

let C_{before} be the sequence $C_{1,j}$
    where $(1, j) \in before$
let R_{update} be the sequence $R_{1,j}$
    where $(1, j) \in update$
let $C_{reorderable}$ be the sequence $C_{1,j}$
    where $(1, j) \in reorderable$
let B_{other_base} be the sequence $C_{2,j}$
    where $(2, j) \in other\_base$

define p_acc as
    $p\_acc(Inv, In, Out, Acc0)$:-
        $switch($
            $\langle C_{before}, R_{update},$
                $p\_acc(Inv, In', InternalOut, accs), C_{reorderable}\rangle,$
            $\langle B_{other\_base}, B_{(3,1)} \ldots B_{(3,o)}\rangle$
).

Figure 4: The fourth stage

```
map_list_to_set(P, L, S) :-
    ( % switch on L
        L => [],                        % C 2,1
        S := set__init                  % C 2,2
    ;
        L => [H | T],                   % C 1,1
        map_list_to_set(P, T, S0),      % C 1,2
        PH := apply(P, H),              % C 1,3
        set__insert(S0, PH, S).         % C 1,4
    ).
```

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. $C_{1,3}$ 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:
$before = \{(1,3)\}$
$update = \{(1,4)\}$
$reorderable = \{\}$
$reject = \{\}$

Stage 2:
$before = \{(1,1), (1,3)\}$
$update\_subst = \{S0 \rightarrow A0, S \rightarrow A\}$
$acc\_var\_subst = \{A \rightarrow A0\}$
$state\_subst = \{S0 \rightarrow S\}$
$accs = \{A\}$
$update\_out\_vars = \{S\}$
$update\_base = \{(2,2)\}$
$other\_base = \{(2,1)\}$

Stage 3:
$call\_to\_head = \{S0 \rightarrow S\}$
$head\_to\_call = \{S \rightarrow S0\}$
$I_{2,2} = A := set\_\_init$

Stage 4:
$Acc0 = \{A0\}$
$InternalOut = \{S\}$
$R_{1,4} = set\_\_insert(A0, PH, A)$
$B_{3,1} = S := A0$

The final result of the algorithm is these two definitions:

```
map_list_to_set(P, L, S) :-
    A := set__init,
    map_list_to_set_acc(L, S, A).
```

```
map_list_to_set_acc(P, L, S, A0) :-
    ( % switch on L
        L => [],
        S := A0
    ;
        L => [H | T],
        PH := apply(P, H),
        set__insert(A0, PH, A),
        map_list_to_set_acc(P, T, S, A)
    ).
```

3. PERFORMANCE 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 each benchmark was tested with the list of integers between 1 and 30 in ascending order, descending order and three random permutations. Their executions were repeated several thousand times to eliminate the uncertainties involved in measuring small time intervals. The results are presented in table 1 in the form of normalized runtimes; the raw data is available from our web site.

Table 1 shows two sets of results: without and with garbage collection. The times are presented in a normalized form, as a percentage of the running time of the predicate without the update state optimization enabled. In all the cases, all the other optimizations of the Mercury compiler were turned on. The following algorithms have been implemented for benchmarking: insertion sort (`isort`), converting a list to a leftist heap (`list_to_leftist_heap`), converting a list to a red-black tree (`list_to_rbtree`), converting a list to a set (`list_to_set`) where a set is represented by an unordered list, converting a list to a binary search tree (`list_to_bst`), finding the maximum of a list (`maxlist`), and finding the sum of a list (`sumlist`).

The optimized predicates process their input lists left-to-right rather than right-to-left. This causes the state to be updated in a different order, resulting in different intermediate states being generated. If the running time of the state update predicate is dependent on the contents of the state, then the running time of the update state predicate may possibly increase, offsetting the decrease in running time due to the tail recursion. For example, inserting an element into a list of N elements, where the element is less then all the elements in the list takes 1 step, while inserting an element which is larger then all the elements in the list takes N steps. The predicate, `isort`, when called with an ascending list of elements, always inserts an element less then all the other elements into the list without the update transformation and while inserting an element larger then all the other elements with the transformation, hence the catastrophic slowdown. However when the input list is a descending list of elements we get the inverse behaviour, hence the fantastic speed up. The other slowdowns in the table have similar reasons.

At the moment, Mercury uses the Boehm conservative garbage collector for C [2]. 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.) The state update predicate for `list_to_set` simply appends the element to be added to the set to the front of the list, hence the running time of the state update predicate is independent of the state. As `list_to_set` is the only predicate in this benchmark which does memory allocations, and since the state update predicate's behavior is independent of the state, memory allocation accounts for a bigger fraction of the runtime of the program with gc than without gc, hence we get smaller relative speedups with gc than with without gc.

	Order	without gc	with gc
isort	asc	1189.2	1454.4
isort	desc	7.9	6.8
isort	randa	85.5	84.5
isort	randb	123.5	129.0
isort	randc	86.3	86.6
list_to_leftist_heap	asc	500.3	250.8
list_to_leftist_heap	desc	26.1	41.7
list_to_leftist_heap	randa	80.3	80.8
list_to_leftist_heap	randb	75.8	77.4
list_to_leftist_heap	randc	96.0	92.7
list_to_rbtree	asc	97.0	100.4
list_to_rbtree	desc	100.5	100.5
list_to_rbtree	randa	95.6	97.4
list_to_rbtree	randb	97.8	100.8
list_to_rbtree	randc	94.2	97.0
list_to_set	asc	55.9	93.6
list_to_set	desc	56.4	93.3
list_to_set	randa	55.4	93.7
list_to_set	randb	56.3	93.9
list_to_set	randc	55.5	93.7
list_to_bst	asc	102.8	101.2
list_to_bst	desc	96.1	99.8
list_to_bst	randa	116.5	120.1
list_to_bst	randb	117.5	120.7
list_to_bst	randc	95.1	95.4
maxlist	asc	80.3	80.1
maxlist	desc	84.3	83.9
maxlist	randa	76.1	76.3
maxlist	randb	82.6	82.1
maxlist	randc	86.2	85.3
sumlist	asc	81.5	82.0
sumlist	desc	81.2	81.6
sumlist	randa	81.6	82.0
sumlist	randb	81.6	81.9
sumlist	randc	81.5	81.9

Table 1: Normalised benchmark results

Integer addition (the state update predicate for `sumlist`) has the additional properties of associativity and commutativity, which means that `sumlist` could be made tail recursive by the transformation we presented in [9]. The two transformations yield quite similar speedups. The update transformation reduces the runtime of `sumlist` given the first 30 integers in ascending order to 81.5% and 82.0% of the original times (for the no gc and gc cases respectively), whereas the transformation presented in [9] reduces it to 80.3% and 79.2% respectively. (The benchmarks where run on the same machine.) It is trivial to prove that the commutativity and associativity promises imply the order-independent state update promise. More benchmarking needs to be done to determine which transformation should be applied when both are applicable.

4. RELATED WORK

Transformations designed to make predicates tail recursive have been studied for the cases where the computation after the recursive call consists of commutative predicates [1], associative predicates [6, 3, 7, 5] for Prolog, and associative predicates or construction unifications [9] for Mercury.

To the best of our knowledge, the only published work which is able to transform predicates which update some state is the forward simulation transformation [4]. Here is the result of the forward simulation transformation applied to the schema from section 2.3, annotated with the modes.

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

p(A, B) :-
    minimal(In'),
        % mode (out)
    init(Out'),
        % mode (out)
    p'(In', Out', A, B).
        % mode (in, in, in, out)

:- mode p'(in, in, in, out) is nondet.

p'(A, B, A, B).
p'(Rest, Out0, A, B) :-
    decompose(In, Current, Rest),
        % mode (out, out, in)
    update(Out0, Current, Out),
        % mode (in, in, out)
    p'(In, Out, A, B).
        % mode (in, in, in, out)
```

There are two key points which make this transformation problematic in Mercury. Point one is that `p'/4` is now nondeterministic as the two clauses are no longer mutually exclusive. This nondeterminism infects all `p`'s ancestors, stopping only at a call site that is either in an all-solutions predicate or in a context where only the existence of a solution matters, and the bindings produced by the solution(s) are irrelevant. This problem could be avoided by using committed-choice nondeterminism [8], possibly coupled with the use of the builtin predicate `promise_only_solution` to hide the committed-choice nondeterminism. Point two is that `decompose/3` and `minimal/1` now have different modes, and there is no guarantee that the new modes are allowed. For example, if `decompose/3` is defined as `In = [Current | Rest]`, then the current Mercury implementation is unable to schedule the unification as both sides of the unification contain a free variable, which means that the recursive clause has no valid mode.

5. CONCLUSION

We have presented a new transformation that can help to make predicates tail recursive in cases where the recursive call is followed by a call to a predicate that update some state. Since most programs contain such code, the transformation is widely applicable. The transformation's correctness depends on the state update predicate having the order-independent state update property. Since some predicates have this property without having properties such as associativity and commutativity, our new transformation is applicable in circumstances where previous transformations based on those other properties are not. Our transformation also avoids the practical problems of the one previous transformation that is applicable in the same circumstances.

The transformation we have presented has been implemented in the Melbourne Mercury compiler, which is available as the release of the day from `<http://www.cs.mu.oz.au/mercury/download.html>`. This implementation has two main limitations. The first is that it requires the programmer to supply promise declarations. This limitation could be lifted by integrating a (not necessarily very powerful) theorem prover into the compiler. The second is that while the transformation usually produces quite respectable speedups, the algorithmic changes it yields can also produce slowdowns. Detecting situations where this may occur and in which therefore the transformation should not be applied is a topic for future work.

We thank the Australian Research Council for their support.

6. REFERENCES

[1] C. Bloch. Source-to-source transformation of logic programs. Master's thesis, Weizmann Institue of Science, Nov. 1984.

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

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

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

[5] H. Büyükyildiz and P. Flener. Generalised logic program transformation schemas. In N. Fuchs, editor, *Proceedings of LOPSTR '97*, pages 45–65, 1998. LNCS 1463.

[6] S. 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.

[7] S. 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.

[8] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Australia, 1996.

[9] P. Ross, D. Overton, and Z. Somogyi. Making Mercury programs tail recursive. In *Pre-Proceedings of the Ninth International Workshop on Logic-based Program Synthesis and Transformation*, pages 107–118, Sept. 1999.

[10] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.