

The Prolog to Mercury transition guide

Version 0.13.1

Thomas Conway
Zoltan Somogyi
Fergus Henderson

Copyright © 1995–2004 The University of Melbourne.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

1 Introduction

This document is intended to help the reader translate existing Prolog programs to Mercury. We assume that the reader is familiar with Prolog. This guide should be used in conjunction with the Mercury User’s Guide and Reference Manuals.

If the Prolog code is quite declarative and does not make use of Prolog’s non-logical constructions, the job of converting it to Mercury will usually be quite straight forward. However, if the Prolog program makes extensive use of non-logical constructions, conversion may be very difficult, and a direct transliteration may be impossible. Mercury code typically has a very different style to most Prolog code.

2 Syntax and declarations

Prolog and Mercury have very similar syntax. Although there are a few differences, by and large if a program is accepted by a Prolog system, it will be accepted by Mercury. There are however a few extra operators defined by the Mercury term parser (see the “Builtin Operators” section of the “Syntax” chapter of the Mercury Language Reference Manual).

In addition, Mercury implements both existential and universal quantification using the syntax

```
some Vars Goal
```

and

```
all Vars Goal
```

The constructor for lists in Mercury is ‘`[]/2`’, not ‘`./2`’.

Terms with functor ‘`{}/N`’ are treated slightly differently in Mercury than in ISO Prolog. ISO Prolog specifies that “`{1, 2, 3}`” is parsed as ‘`{}(, , (1, , (2, 3)))`’. In Mercury, it is parsed as ‘`{}(1, 2, 3)`’.

Mercury does not (yet) allow users to define their own operators.

3 Input and output

Mercury is a purely declarative language. Therefore it cannot use Prolog’s mechanism for doing input and output with side-effects. The mechanism that Mercury uses is the threading of an object that represents the state of the world through the computation. The type of this structure is ‘`io.state`’. The modes of the two arguments that are added to calls are ‘`di`’ for “destructive input” and ‘`uo`’ for “unique output”. The first means that the input variable must be the last reference to the original state of the world, and that the output variable will be the only reference to the state of the world produced by this predicate.

Predicates that do input or output must have these arguments added. For example the Prolog predicate:

```

write_total(Total) :-
    write('The total is '),
    write(Total),
    write('.') ,
    nl.

```

in Mercury becomes

```

:- pred write_total(int, io.state, io.state).
:- mode write_total(in, di, uo) is det.

```

```

write_total(Total, IO0, IO) :-
    print("The total is ", IO0, IO1),
    print(Total, IO1, IO2),
    print('.', IO2, IO3),
    nl(IO3, IO).

```

Definite Clause Grammars (DCGs) are convenient syntactic sugar to use in such situations. The above clause can also be written

```

write_total(Total) -->
    print("The total is "),
    print(Total),
    print('.') ,
    nl.

```

In DCGs, any calls (including unifications) that do not need the extra DCG arguments are escaped in the usual way by surrounding them in curly braces (`{ }`).

Note that in Mercury you normally use strings ("`...`") rather than atoms ("`'...'`") for messages like "The total is". (It is possible to use atoms, but you have to declare each such atom before-hand, so it is more convenient to use strings.) However, for strings and characters, `'write'` prints out the quotes; to avoid this, you need to use `'print'` instead of `'write'`.

Both `'write'` and `'print'` are defined in the `'io'` module in the Mercury standard library.

One of the important consequences of our model for input and output is that predicates that can fail may not do input or output. This is because the state of the world must be a unique object, and each IO operation destructively replaces it with a new state. Since each IO operation destroys the current state object and produces a new one, it is not possible for IO to be performed in a context that may fail, since when failure occurs the old state of the world will have been destroyed, and since bindings cannot be exported from a failing computation, the new state of the world is not accessible.

In some circumstances, Prolog programs that suffer from this problem can be fixed by moving the IO out of the failing context. For example

```

...
( solve(Goal) ->
    ...
;
    ...
),
...

```

where ‘`solve(Goal)`’ does some IO can be transformed into valid Mercury in at least two ways. The first is to make ‘`solve`’ deterministic and return a status:

```
...
solve(Goal, Result, IO6, IO7),
( Result = yes ->
    ...
;
    ...
),
...
```

The other way is to transform ‘`solve`’ so that all the input and output takes place outside it:

```
...
io.write_string("calling: ", IO6, IO7),
solve.write_goal(Goal, IO7, IO8),
( solve(Goal) ->
    io.write_string("succeeded\n", IO8, IO9),
    ...
;
    IO9 = IO8,
    ...
),
...
```

4 Failure driven loops, assert and retract

Because Mercury is purely declarative, the goal ‘`Goal, fail`’ is interchangeable with the goal ‘`fail, Goal`’. Also because it is purely declarative, there are no side effects to goals (see also the section on input and output). As a consequence of these two facts, it is not possible to write failure driven loops in Mercury. Neither is it possible to use predicates such as `assert` or `retract`. This is not the place to argue it, but we believe most programs that use failure driven loops, `assert` and `retract` to be less clear and harder to maintain than those that do not.

The use of `assert` and `retract` should be replaced with a collection data structure threaded through the relevant part of the program. Data which is truly global may be stored in the ‘`io.state`’ using the predicates ‘`io.get_globals`’ and ‘`io.set_globals`’. These predicates take an argument of type ‘`univ`’, the universal type, so that by using ‘`type_to_univ`’ and ‘`univ_to_type`’ it is possible to store data of any type in the ‘`io.state`’.

The standard library contains several abstract data types for storing collections, each of which will be useful for different classes of problems.

The ‘`list`’ ADT is useful if the order of the asserted facts is important. The ‘`set`’ ADT is useful if the order is not important, and if the asserted facts are not key-value pairs. If the asserted facts are key-value pairs, you can choose among several ADTs, including ‘`map`’, ‘`bintree`’, ‘`rbtree`’, and ‘`tree234`’. We recommend the ‘`map`’ ADT for generic use. Its current implementation is as a 234 tree (using ‘`tree234`’), but in the future it may

change to a hash table, or a trie, or it may become a module that chooses among several implementation methods dynamically depending on the size and characteristics of the data.

Failure driven loops in Prolog programs should be transformed into ordinary tail recursion in Mercury. This does have the disadvantage that the heap space used by the failing clause is not reclaimed immediately but only through garbage collection, but we are working on ways to fix this problem. In any case, the transformed code is more declarative and hence easier to maintain and understand for humans and easier for the compiler to optimize.

5 Cuts and indexing

The cut operator is not part of the Mercury language. In addition, the conditional operator ‘`-> ;`’ does not do a hard cut across the condition — only a soft cut which prunes away either the ‘then’ goal or the ‘else’ goal. If there are multiple solutions to the condition, they will all be found on backtracking.

Prolog programs that use cuts and a ‘catch-all’ clause should be transformed to use if-then-else.

For example

```
p(this, ...) :- !,
    ...
p(that, ...) :- !,
    ...
p(Thing, ...) :-
    ...
```

should be rewritten as

```
p(Thing, ...) :-
    ( Thing = this ->
      ...
      ; Thing = that ->
      ...
    ;
      ...
    ).
```

The Mercury compiler does much better indexing than most Prolog compilers. Actually, the compiler indexes on all input variables to a disjunction (separate clauses of a predicate are merged into a single clause with a disjunction inside the compiler). As a consequence, the Mercury compiler indexes on all arguments. It also does deep indexing. That is, a predicate such as the following will be indexed.

```
p([f(g(h)) | Rest]) :- ...
p([f(g(i)) | Rest]) :- ...
```

Since indexing is done on disjunctions rather than clauses, it is often unnecessary to introduce auxiliary predicates in Mercury, whereas in Prolog it is often important to do so for efficiency.

If you have a predicate that needs to test all the functors of a type, it is better to use a disjunction instead of a chain of conditionals, for two reasons. First, if you add a new functor to a type, the compiler will still accept the now incomplete conditionals, whereas if

you use a disjunction you will get a determinism error that pinpoints which part of the code needs changing. Second, in some situations the code generator can implement an indexed disjunction (which we call a *switch*) using binary search, a jump table or a hash table, which can be faster than a chain of if-then-elses.

6 Accumulators and Difference lists

Mercury does not in general allow the kind of aliasing that is used in difference lists. Prolog programs using difference lists fall in to two categories — programs whose data flow is “left-to-right”, or can be made left-to-right by reordering conjunctions (the Mercury compiler automatically reorders conjunctions so that all consumers of a variable come after the producer), and those that contain circular dataflow.

Programs which do not contain circular dataflow do not cause any trouble in Mercury, although the implicit reordering can sometimes mean that programs which are tail recursive in Prolog are not tail recursive in Mercury. For example, here is a difference-list implementation of quick-sort in Prolog:

```
qsort(L0, L) :- qsort_2(L0, L - []).

qsort_2([], R - R).
qsort_2([X|L], R0 - R) :-
  partition(L, X, L1, L2),
  qsort_2(L1, R0 - R1),
  R1 = [X|R2],
  qsort_2(L2, R2 - R).
```

Due to an unfortunate limitation of the current Mercury implementation (partially instantiated modes don't yet work correctly), you need to replace all the ‘-’ symbols with commas. However, once this is done, and once you have added the appropriate declarations, Mercury has no trouble with this code. Although the Prolog code is written in a way that traverses the input list left-to-right, appending elements to the tail of a difference list to produce the output, Mercury will in fact reorder the code so that it traverses the input list right-to-left and constructs the output list bottom-up rather than top-down. In this particular case, the reordered code is still tail recursive — but it is tail-recursive on the first recursive call, not the second one!

If the occasional loss of tail recursion causes efficiency problems, or if the program contains circular data flow, then a different solution must be adopted. One way to translate such programs is to transform the difference list into an accumulator. Instead of appending elements to the end of a difference list by binding the tail pointer, you simply insert elements onto the front of a list accumulator. At the end of the loop, you can call ‘`list.reverse`’ to put the elements in the correct order if necessary. Although this may require two traversals of the list, it is still linear in complexity, and it probably still runs faster than the Prolog code using difference lists.

In most circumstances, the need for difference lists is negated by the simple fact that Mercury is efficient enough for them to be unnecessary. Occasionally they can lead to a significant improvement in the complexity of an operation (mixed insertions and deletions from a long queue, for example) and in these situations an alternative solution should be

sought (in the case of queues, the Mercury library uses the pair of lists proposed by Richard O’Keefe).

7 Determinism

The Mercury language requires that the determinism of all predicates exported by a module be declared. The determinism of predicates that are local to a module may either be declared or inferred. By default, the compiler issues a warning message where such declarations are omitted, but this warning can be disabled using the ‘`--no-warn-missing-det-decls`’ option if you want to use determinism inference.

Determinism checking and inference is an undecidable problem in the general case, so it is possible to write programs that are deterministic, and have the compiler fail to prove the fact. The most important aspect of this problem is that the Mercury compiler only detects the clauses of a predicate (or the arms of a disjunction, in the general case) to be mutually exclusive (and hence deterministic) if they are distinguished by the unification of a variable (possibly renamed) with distinct functors in the different clauses (or disjuncts), so long as the unifications take place before the first call in the clause (or disjunct). In these cases, the Mercury compiler generates a *switch* (see the earlier section on indexing). If a switch has a branch for every functor on the type of the switching variable, then the switch cannot fail (though one or more of its arms may do so).

The Mercury compiler does not do any range checking of integers, so code such as:

```
factorial(0, 1).
factorial(N, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is F1 * N.
```

would be inferred “nondeterministic”. The compiler would infer that the two clauses are not mutually exclusive because it does not know about the semantics of ‘>/2’, and it would infer that the predicate as a whole could fail because the call to ‘>/2’ can fail.

The general solution to such problems is to use an if-then-else:

```
:- pred factorial(int, int).
:- mode factorial(in, out) is det.

factorial(N, F) :-
    ( N =< 0 ->
      F = 1
    ;
      N1 is N - 1,
      factorial(N1, F1),
      F is F1 * N
    ).
```

8 All-solutions predicates.

Prolog's various different all-solutions predicates (`findall/3`, `bagof/3`, and `setof/3`) all have semantic problems. Mercury has a different set of all-solutions predicates (`solutions/2`, `solutions_set/2`, and `unsorted_solutions/2` — all defined in the library module `'solutions'`) that address the problems of the Prolog versions. To avoid the variable scoping problems of the Prolog versions, rather than taking both a goal to execute and an aliased term holding the resulting value to collect, Mercury's all-solutions predicates take as input a single higher-order predicate term. The Mercury equivalent to

```
intersect(List1, List2, Intersection) :-
  setof(X, (member(X, List1), member(X, List2)), Intersection).
```

is

```
intersect(List1, List2, Intersection) :-
  solutions((pred(X::out) is nondet :-
    (list.member(X, List1), list.member(X, List2))), Intersection).
```

Alternately, this could also be written as

```
intersect(List1, List2, Intersection) :-
  solutions(member_of_both(List1, List2), Intersection).

:- pred member_of_both(list(T)::in, list(T)::in, T::out) is nondet.
member_of_both(List1, List2, X) :-
  list.member(X, List1), list.member(X, List2).
```

and in fact that's exactly how the Mercury compiler implements lambda expressions.

The current implementation of `solutions/2` is a “zero-copy” implementation, so the cost of `solutions/2` is proportional the number of solutions, but independent of the size of the solutions. (This may change in future implementations.)