The Prolog to Mercury transition guide

Version rotd-2021-05-04

Thomas Conway
Zoltan Somogyi
Fergus Henderson

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 the syntax of 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 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 object is `io.state`, or just `io` for short. Each operation that affects the state of the world must have two arguments of this type, representing respectively the state of the world before the operation, and the state of the world after the operation. 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 the latter means that the output variable is guaranteed to be the only reference to the state of the world produced by this predicate.

For example, the direct translation of the Prolog predicate

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

into Mercury yields this Mercury predicate:

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

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

The variables `IO0`, `IO1` etc each represent one version of the state of the world. `IO0` represents the state before the total is printed, `IO1` represents the state after just `The total is` is printed, and so on. However, programmers usually don't want to give specific names to all these different versions; they want to name only the entities that all these variables represent different versions of. That is why Mercury supports state variable notation. This is syntactic sugar designed to make it easier to thread a sequence of variables holding the successive states of an entity through a clause. You as the programmer name only the entity, and let the compiler name the various versions. With state variables, the above clause would be written as

```
write_total(Total, !IO) :-
    print("The total is ", !IO),
    print(Total, !IO),
    print('.', !IO),
    nl(!IO).
```

and the compiler will internally convert this clause into code that looks like the previous clause. (The usual convention in Mercury programs is to name the state variable representing the state of the world `!IO`.)

In the head of a clause, what looks like an argument that consists of a variable name prefixed by an exclamation mark actually stands for two arguments which are both variables, holding the initial and final state of whatever entity the state variable stands for. In this case, they stand for the state of the world, respectively before and after the line about the total has been printed. In calls in the body of a clause, what looks like an argument that consists of a variable name prefixed by an exclamation mark also stands for two arguments which are both variables, but these hold respectively, the current and the next state.

In Prolog, it is quite normal to give to `print` an argument that is an atom that is not used anywhere else in the program, or at least not in code related to the code that does the printing. This is because the term being printed does not have to belong to a defined type. Since Mercury is strongly typed, the atom being printed *would* have to be a data constructor of a defined type. A Mercury programmer *could* define a meaningless type just to give one of its data constructors to a call to `print`, but it is far better to simply call a predicate specifically designed to print the string, or integer, or character, you want to print:

```
write_total(Total, !IO) :-
    io.write_string("The total is ", !IO),
    io.write_int(Total, !IO),
    io.write_char('.', !IO),
    io.nl(!IO).
```

The `io.` prefix on the predicates called in the body indicates that the callees are in the `io` module of the Mercury standard library. This module contains all of Mercury's primitive I/O operations. These *module qualifications* are not strictly necessary (unless two or more modules define predicates with the same names and argument types, the Mercury compiler can figure out which modules called predicates are in), but Mercury convention is to make the module qualifier explicit in order to make the intent of the code crystal clear to readers.

The above could also be written more compactly like this:

```
write_total(Total, !IO) :-
    io.format("The total is %d.\n", [i(Total)], !IO).
```

The first argument of `io.format` is a format string modelled directly on the format strings supported by `printf` in C, while the second is a list of the values to be printed, which should have one value for each conversion specifier. In this case, there is one conversion specifier, '`%d`', which calls for the printing of an integer as a decimal number, and the corresponding value is the integer `Total`. Since Mercury is strongly typed, and different arguments may have different types, in the argument list integers must be wrapped inside `i()`, floats must be wrapped inside `f()`, strings must be wrapped inside `s()`, and chars must be wrapped inside `c()`. Despite appearances, in the usual case of the format string being constant, the wrappers and the list of arguments have neither time nor space overhead, because the compiler optimizes them away, replacing the call to `io.format` with the calls to `io.write_string`, `io.write_int` etc above.

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 I/O operation destructively replaces it with a new state. Since each I/O operation destroys the current state object and produces a new one, it is not possible for I/O 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 I/O out of the failing context. For example

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

where '`solve(Goal)`' does some I/O 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, !IO),
(
    Result = success(...),
    ...
;
    Result = failure,
    ...
),
...
```

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

```
...
io.write_string("calling: ", !IO),
solve.write_goal(Goal, !IO),
( solve(Goal) ->
    io.write_string("succeeded\n", !IO),
    ...
;
    ...
),
...
```

4 Assert and retract

In Prolog, calls to the builtin predicates `assert` and `retract` can change the set of clauses of the program currently being executed. This makes compilation very tricky, and different Prolog systems react differently when the program alters the definition of a predicate that has active calls. It also makes program analysis almost impossible, since the program that the compiler should analyze is not actually available at compilation time. Since Mercury is a compiled language, it does not allow the compiled program to be altered in any way.

Most uses of `assert` and `retract` in Prolog programs are not actually intended to alter the program. Their purpose is just to maintain a set of facts, with semantically separate sets of facts being stored in separate predicates. (Most Prolog systems require these predicates to be marked as `dynamic` predicates.) A Mercury programmer who wants to store a set of facts would simply store those facts as data (not as code) in a data structure.

The standard library contains several abstract data types (ADTs) for storing collections of items, each of which is useful for different classes of problems.

If the order of the items in the collection is important, consider the `list` and `cord` ADTs. `list` has lower constant factors, but the `cord` ADTs supports concatenation in constant time. The `stack` and `queue` ADTs implement lists with specific semantics and operations appropriate to those semantics.

If the order of items in the collection is not important, and if the items are key-value pairs, you can store them in ADTs implementing several different kinds of trees, including `rbtree` and `tree234`. In the absence of a compelling reason to choose a different implementation,

we recommend the `map` ADT for generic use. Maps are implemented using 234 trees, which are guaranteed to be balanced and thus have good worst-case behavior, but also have good performance in the average case. `bimap`, `injection`, `multi_map` and `rtree` are specialized version of maps.

If the items in the collection are not key-value pairs, then consider the `set` and `bag` ADTs. The `set` ADT itself has several versions, some based on trees and some based on bit vectors, each with its own tradeoffs.

The Mercury standard library has some modules for more specialized collections as well, such as graphs. And of course, if needed, you can always create your own ADT.

If for some reason you cannot thread variables holding some data through the parts of your program that need access to that data, then you can store that data in a '`mutable`', which is as close as Mercury comes to Prolog's dynamic predicates. Each Mercury mutable stores one value, though of course this value can be a collection, and that collection may be (but doesn't have to be) implemented by one of the Mercury standard library modules listed above.

Each mutable has a getter and setter predicate. You can set things up so that the getter and setter predicates both function as I/O operations, destroying the current state of the world and returning a new state of the world. This effectively considers the mutable to be part of the state of the world *outside* the Mercury program. The `io` module also provides another way to do this, by allowing the storage of information 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`.

Alternatively, you can set things up so that the getter and setter predicates of a mutable are *not* I/O operations, but in that case calls to those predicates are not considered pure Mercury, and must instead use Mercury's mechanisms for controlled impurity. These mechanisms require all code that is not pure Mercury to be explicitly marked as such. They are intended to allow programmers to implement pure interfaces using *small* pieces of impure code, for use in circumstances where there is no feasible way to implement that same interface using pure code. Most Mercury programs do not use impure code at all. The ones that do make use of it use it very sparingly, with 99.9+% of their code being pure Mercury.

5 Failure driven loops

In pure Mercury code, the goal `Goal, fail` is interchangeable with the goal `fail, Goal`, and `Goal` cannot have any side effects. As a consequence of these two facts, it is not possible to write failure driven loops in pure Mercury code. While one could try to use Mercury's mechanisms for controlled impurity to implement failure driven loops using impure Mercury code, this is not part of the culture of Mercury programming, because failure driven loops are significantly less clear and harder to maintain than other means of iterating through a sequence. Since they are inherently imperative and not declarative, they are also very hard for compilers to optimize.

If the sequence must be generated through backtracking, then a Mercury programmer could just collect all the solutions together using the standard Mercury library predicate

`solutions`, and iterate through the resulting list of solutions using an ordinary tail recursive predicate.

However, most Mercury programmers would prefer to generate a list of solutions directly. This can be easily done by replacing code that generates alternative solutions through backtracking, using predicates like this:

```
generate_solutions(In1, In2, Soln) :-
    (
        % Generate one value of Soln from In1 and In2.
        generate_one_soln(In1, In2, Soln)
    ;
        % Compute a new value for the second input.
        In2' = ....
        % Generate more values of Soln from In1 and In2'.
        generate_solutions(In1, In2', Soln)
    ).
```

in which the different solutions are produced by different disjuncts, with predicates in which the different solutions are produced by different conjuncts, like this:

```
generate_solutions(In1, In2, [Soln | Solns]) :-
    generate_one_soln(In1, In2, Soln),
    In2' = ....
    generate_solutions(In1, In2', Solns).
```

Unlike predicates following the previous pattern, predicates following this pattern can exploit Mercury's determinism system to ensure that they have considered all the possible combinations of the values of the input arguments. They are also more efficient, since choice point creation is expensive.

They can be made even more efficient if the consumption of the solutions can be interleaved with their generation. For example, if the solutions are intended to be inputs to a fold (i.e. each solution is intended to update an accumulator), then this interleaving can be done like this:

```
generate_and_use_solutions(In1, In2, !Acc) :-
    generate_one_soln(In1, In2, Soln),
    use_solution(Soln, !Acc),
    In2' = ....
    generate_and_use_solutions(In1, In2', !Acc).
```

6 Cuts and indexing

The Prolog cut operator is not part of the Mercury language. Most Prolog code that uses cuts should probably be translated into Mercury using if-then-elses.

In both Prolog and Mercury, the behavior of an if-then-else '`C -> T ; E`' depends on whether the condition '`C`' has any solutions. If it does, then they both execute the then-part '`T`'; if it does not, then they both execute the else-part '`E`'. However, Prolog and Mercury differ in what they do if the condition has *more* than one solution.

In most versions of Prolog, if the condition of an if-then-else has more than one solution, the if-then-else throws away all its solutions after the first. The way this is usually

implemented is that after the condition generates its first solution, and before execution continues to the then-part goal with the bindings in that first solution, the Prolog implementation cuts away all the choice points in the condition. This prevents backtracking into the condition, which thus cannot generate any of its *other* solutions.

Mercury does *not* prune away later solutions in conditions. If the condition has more than one solution, Mercury will execute the then-part goal on every one of them in turn, using the usual rules of backtracking.

Mercury allows if-then-elses to be written not just as '`C -> T ; E`', but also as '`if C then T else E`'. These two syntaxes have identical semantics.

Prolog programs that use cuts and a 'catch-all' clause should be transformed to use if-then-else in Mercury.

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.

7 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).

8 Determinism

The Mercury language requires the determinism of all predicates exported by a module to be declared. The determinism of predicates that are local to a module may be declared but don't have to be; if they are not declared, they will be inferred. By default, the compiler issues a warning message where such declarations are omitted, but if you want to use determinism inference, you can disable this warning using the '`--no-warn-missing-det-decls`' option.

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, allowing the execution of at most one disjunct at runtime, if the clauses or disjuncts each unify the same variable (or a copy of that variable) with distinct functors, with these unifications all taking place before the first call in the clause or disjunct. For such disjunctions, the Mercury compiler generates a *switch* (see the earlier section on indexing). If a switch has a branch for every functor in the type of the switched-on variable, then the switch guarantees that exactly one of its arms will be executed. If all the arms are deterministic goals, then the switch itself is deterministic.

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 to be "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 (a) the unification of the first argument with `0` can fail, so the first clause is not guaranteed to generate a solution, and (b) the call to `>/2` can fail, and so the second clause is not guaranteed to generate a solution either.

The general solution to such problems is to use a chain of one or more if-then-elses.

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

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

The `unexpected` predicate is defined in the `require` module of the Mercury standard library. Calls to it throw an exception, and unless that exception is caught, it aborts the program. The terms `$pred` is automatically replaced by the compiler with the (module-qualified) name of the predicate in which it appears.

9 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 is 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 independent of the *size* of the solutions, though it is proportional to the *number* of solutions.