The design and implementation of Mercury

Zoltan Somogyi
Fergus Henderson

{zs,fjh}@cs.mu.OZ.AU
http://www.cs.mu.oz.au/mercury
Department of Computer Science
University of Melbourne
Australia

Objective

- general purpose
- program reliability
- programmer productivity
- support for large programs
- support for team programming
- support for program maintenance
- program efficiency

Mercury is related to Prolog the way Ada is related to Basic. They are both based on logic, but programming in Mercury is quite different to programming in Prolog.

“The superior pilot uses his superior judgement to avoid situations that would require his superior skill.”

Outline of Mercury

- unified logic/functional paradigm
- strong polymorphic type system
- strong mode system
- strong determinism system
- strong module system
- inference within modules
- higher-order
- purely declarative (even I/O)

The language design deliberately restricts the class of allowed programs in return for significant increases in productivity, reliability and efficiency.

The type system

The Mercury type system is a Hindley / Milner system very similar to the type systems of Miranda, Haskell and Godel.

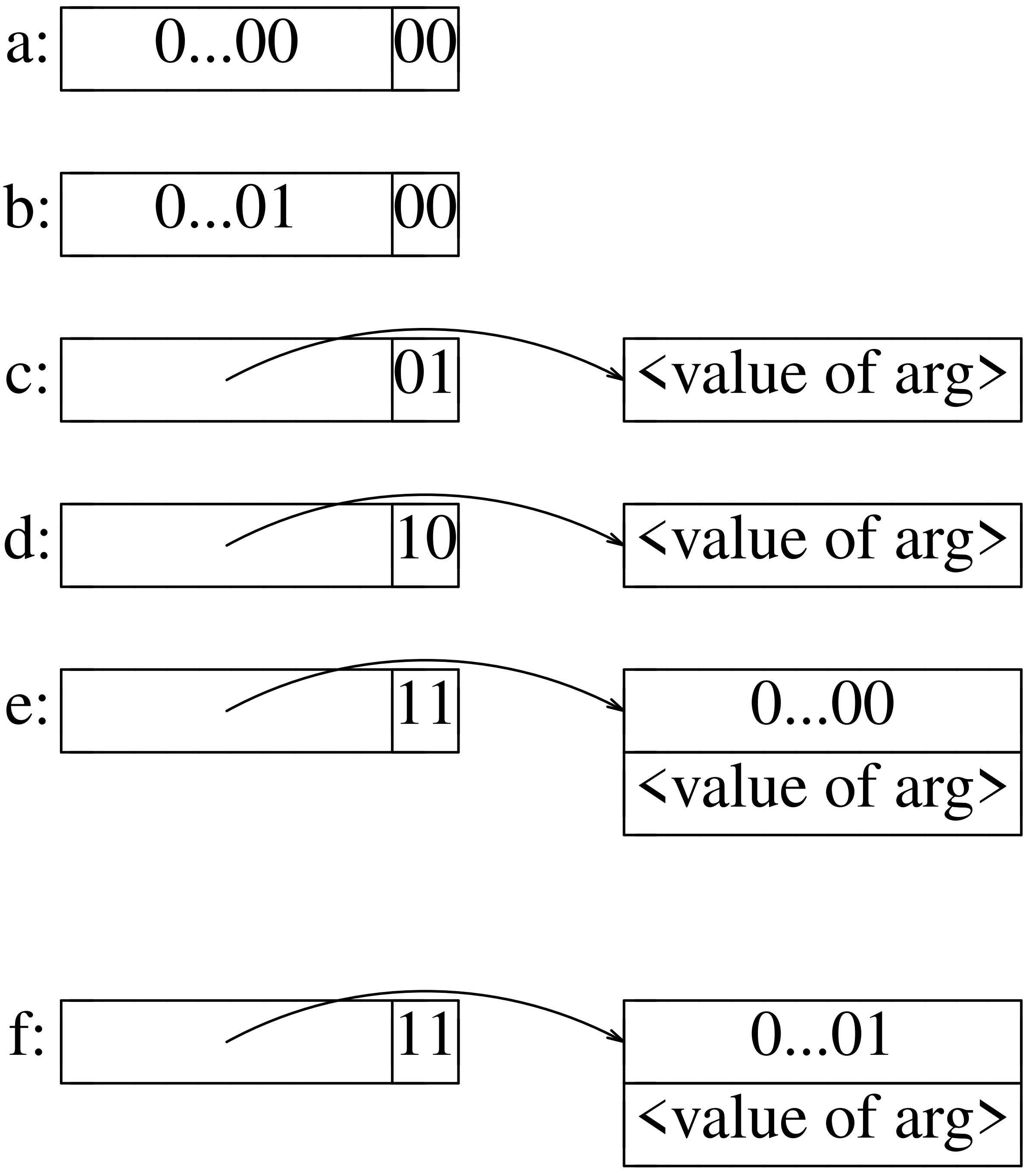
It has parametric polymorphism and higher order types, but there is no support for type classes or subtyping (yet).

```
:- type tree(K,V) --->
        empty
    ;   node(K, V, tree(K,V), tree(K,V)).

:- type student_table ==
    tree(student_id, student_info).

:- pred append(list(T), list(T), list(T)).
```

:- type t ---> a ; b ; c(t1) ; d(t2) ; e(t3) ; f(t4).



Two views of modes

Prescriptive view: modes allocate responsibility to instantiate (each part of) each argument, to either this predicate or its caller.

Descriptive view: modes describe the instantiation states of arguments at entry to and (possibly) return from this predicate.

Mercury takes the prescriptive view, which is order independent, and in which modes convey design level information.

The compiler transforms the program so that the descriptive view of the transformed program conforms to the prescriptive view.

Mode declarations

For each argument of a predicate, a mode declaration says whether the argument is input to the call (ground->ground) or output by the call (free->ground).

A predicate or function may have more than one mode. We call each such mode a *procedure*.

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

The compiler generates separate code for each procedure.

Mode analysis

Works on each procedure separately.

- identifies the modes of all the calls and builtins in the definition of the procedure;
- reorders conjunctions to ensure that the producer of a variable comes before all its consumers;
- ensures that goals do not unify two free variables;
- ensures that goals in negated contexts bind no non-local variables;
- checks the final instantiation states of the arguments.

Mode analysis

```
% append(in, in, out)
append(A, B, C) :-
    A == [],
    C := B.
append(A, B, C) :-
    A == [H | T],
    append(T, B, NT),   % no LCO
    C := [H | NT].

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

Implied modes

Some calls specify more arguments than required by any mode declaration, e.g. calls to append with three ground lists.

Mode analysis picks a mode compatible with the call, and introduces extra unifications after the call.

This turns e.g. append([1,2], [3], [4,5]) into append([1,2], [3], C), C = [4,5] or into append(A, B, [4,5]), A = [1,2], B = [3].

This is necessary because the code for e.g. append(in, in, out) assumes that output arguments are free variables when the call is made.

Theoretical implications

Well moded programs do not need the occur check; X = f(X) is a mode error unless some other goal makes X ground.

Well moded programs must have a producer for every consumed variable, which means that they can never flounder.

Since mode analysis checks that goals in negated contexts cannot instantiate any variables visible from the outside, negation as failure is sound without runtime checks.

The reordering of conjunctions makes reversible predicates easier to write.

Implementation implications

When the program looks at a variable, it always knows whether the variable is bound or not; if the variable is free, it cannot be linked to any other variable.

Dereferencing is therefore never needed.

It is OK for free variables to be represented by garbage. Therefore free variables do not have to be initialized at all.

There is no need for a trail, since resetting a word to garbage is a null operation.

Implementation implications

Builtins (e.g. assignments, indexing tests) need not check that their inputs are bound, need not handle unbound inputs, and can simply overwrite their outputs.

Callers can pass input arguments by value, and callees can return output arguments by value.

Generating separate code for each mode of a predicate or function does not lead to program size explosion because the code of each mode is specialized, and because few predicates (<5%) have more than one mode.

Determinism

The determinism of a procedure gives upper and lower bounds on the number of its solutions. This is a declarative notion, and undecidable. The compiler actually checks stronger conditions:

- maximum number of times it can succeed, and
- whether it can fail before succeeding once.

can fail?	max number of successes		
	0	1	>1
no	erroneous	det	multi
yes	failure	semidet	nondet

Determinism analysis

Determinism declarations are attached to mode declarations:

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

If it terminates, append(in, in, out) is guaranteed to have exactly one solution because

- the recursive call is deterministic;
- the assignments are deterministic;
- exactly one of the test unifications will succeed.

Specialized execution algorithms

The execution algorithm we use for procedures that can succeed at most once is very similar to the algorithms used by imperative languages. These procedures use the det stack, whose frames are popped on success.

The execution algorithm for nondet and multi procedures is more Prolog-like. Such procedures use the nondet stack, whose frames are bigger, more complex and are popped only on failure.

Our use of two specialized execution algorithms ensures that the costs of nondeterminism are paid only by the parts of the program that need it.

```
append_3_0:
    if ((r1 == mkword(mktag(0), mkbody(0))))
        goto append_3_0_i1000;
    incr_sp(2);
    detstackvar(2) = succip;
    detstackvar(1) = field(mktag(1), r1, 0);
    r1 = field(mktag(1), r1, 1);
    succip = &&append_3_0_i5;
    goto append_3_0;
append_3_0_i5:
    r1 = r3;
    tag_incr_hp(r3, mktag(1), 2);
    field(mktag(1), r3, 0) = detstackvar(1);
    field(mktag(1), r3, 1) = r1;
    succip = detstackvar(2);
    decr_sp(2);
    goto *succip;
append_3_0_i1000:
    r3 = r2;
    goto *succip;
```

Functions

```
:- func sum(list(int)) = int.
% :- mode sum(in) = out is det.
sum([]) = 0.
sum([X|Xs]) = X + sum(Xs).

% conditional equations
:- func max(int, int) = int.
max(X, Y) = Max :-
    (X >= Y -> Max = X ; Max = Y).

% if-then-else expressions
:- func min(int, int) = int.
min(X, Y) = (if X >= Y then X else Y).

:- func pi = float.
pi = 3.14159265359.
```

Functional logic programming

Functions can have multiple modes, just like predicates.

```
:- func '+'(int, int) = int.
:- mode '+'(in, in) = out is det.
:- mode '+'(in, out) = in is det.
:- mode '+'(out, in) = in is det.
:- mode '+'(in, in) = in is semidet.
```

The Mercury language semantics allows implementations to use lazy evaluation for det modes of functions or predicates, but does not require it. (The current implementation uses flattening.)

Defining higher-order procedures

```
:- func mf(func(T1) = T2, list(T1)) = list(T2).
:- mode mf(func(in) = out is det, in) = out
     is det.
mf(_F, []) = [].
mf(F, [X|Xs]) = [apply(F, X) | mf(F, Xs)].

:- pred mp(pred(T1, T2), list(T1), list(T2)).
:- mode mp(pred(in, out) is det, in, out) is det.
:- mode mp(pred(out, in) is semidet, out, in)
     is semidet.
% ... more modes possible
mp(_P, [], []).
mp(P, [X|Xs], [Y|Ys]) :-
     call(P, X, Y),
     mp(P, Xs, Ys).
```

Using higher-order procedures

```
% a simple example
mf(sum, [[1,2],[3,4]], L)
gives L = [3,7].

% predicate currying
mp(append([1]), [[1,2],[3,4]], L)
gives L = [[1,1,2],[1,3,4].

% lambda functions
mf((func(X) = X+1), [10,20], L)
gives L = [11,21].

% lambda predicate, "backwards" mode
mp((pred(X::out, Y::in) is semidet :-
            X >= 0, Y >= 0, X + Y = 10),
        L, [1,2,3])
gives L = [9,8,7].
```

Unique modes

Declarative programming languages need destructive update in some form to implement input/output, to interface to imperative languages, and to achieve the efficiency of imperative languages.

Mercury uses unique modes to enable the implementation to use destructive update without sacrificing referential transparency.

```
% "unique output"
:- mode uo :: free -> unique.

% "destructive input"
:- mode di :: unique -> dead.

:- mode append(di, di, uo) is det.
```

Input/output

Mercury views I/O predicates as describing a relationship between two states of the world: the one before the action and the one after.

```
:- pred main(io__state, io__state).
:- mode main(di, uo) is det.

main(S0, S) :-
    io__write_string("Hello, ", S0, S1),
    io__write_string("world\n", S1, S).
```

Since the world has only one state at any one time, the compiler must make sure that old states of the world are never referred to again.

Unique modes and backtracking

We call variables that are unique during forward execution but may be needed again on backtracking “mostly unique”; one can apply *trailed* destructive update to such variables.

Variables that are not needed on backtracking either are fully “unique”; updates to them need not be trailed. I/O actions cannot be trailed.

```
(
    io__write_string("testing...", S0, S1),
    1 = 2
->
    io__write_string("yes\n", S1, S)
;
    io__write_string("no\n", S0, S)
)
```

Modules

```
:- module stack.
:- interface.

:- type stack(T). % abstract data type
:- func empty_stack = stack(T).
:- func push(stack(T), T) = stack(T).
:- pred pop(stack(T)::in, T::out,
      stack(T)::out) is semidet.

:- implementation.

:- import_module list.
:- type stack(T) == list(T).

empty_stack = [].
push(Xs, X) = [X | Xs].
pop([X | Xs], X, Xs).
```

Optimizations

The literature has many source-to-source optimizations. Almost all require pure programs, and most depend on dataflow information.

inlining
specialization of higher-order procedures
(constraint propagation)
middle-recursion optimization
short-circuiting jumps, calls and returns
delaying stack frame creation
...

Our philosophy is that if there is a feasible algorithm for an optimization, then it should be done by the compiler, not the programmer.

Ten small benchmarks

system	speed
SWI-Prolog	1.00
NU-Prolog	1.95
wamcc	3.14
Quintus	3.75
SICStus compact	2.22
SICStus fast	7.41
Aquarius	19.01
Mercury gc	9.24
Mercury no gc	36.52

The gap between Mercury and Aquarius is greatest for the two biggest benchmarks, poly and crypt.

A medium-sized benchmark

The pseudoknot benchmark from Uni of Montreal is a number crunching program; its size is about 3,500 lines.

	time (s)		space
system	compile	run	run
cc -O4	634	1.1	<1 Mb
gcc -O2	849	1.3	<1 Mb
Mer, no gc	487	4.8	21 Mb
Mer, gc	487	7.1	<1 Mb
SICStus	66	71	15 Mb
BinProlog	22	N/A	N/A
NU-Prolog	16	N/A	N/A

A large benchmark

We have compared the performance of Mercury against SICStus Prolog and NU-Prolog on a very large benchmark, the Mercury compiler itself, which is 3.6 Mb of source (100,000 lines).

system	speed	size
NU-Prolog	N/A	2.8 Mb
SICStus compact	1.0	6.0 Mb
SICStus fast	1.5	10.3 Mb
Mercury gc	5.6	2.3 Mb

Reliability

Mercury requires programmers to declare several aspects of the intended behavior of their predicates.

By checking the code against these declarations, the Mercury compiler can find a large fraction of the errors in the program. The declarations also serve as documentation.

Many types of bugs cannot happen at all in Mercury. Many other types are guaranteed to be caught by the compiler.

Student A: What causes core dumps?
Student B: Programming in C.

Error messages

```
list.m:31: In clause for ‘append(in, in, out)’:
list.m:31:    in call to predicate ‘list:append/3’:
list.m:31:    mode error: arguments ‘T, L1, NT’
list.m:31:    have insts ‘ground, free, free’,
list.m:31:    which does not match any of the modes
              for predicate ‘list:append/3’.

list.m:26: In clause for ‘append(in, in, out)’:
list.m:26:    mode mismatch in disjunction.
list.m:26:    ‘HeadVar__3’ :: ground, unique(’.’(free, ground)).

list.m:10: In ‘append(in, in, out)’:
list.m:10:    Error: determinism declaration not satisfied.
list.m:10:    Declared ‘det’, inferred ‘semidet’.
list.m:16:    in argument 1 of clause head:
list.m:16:    unification of ‘HeadVar__1’ and ‘[H | T]’ can fail.
list.m:11: In ‘append(out, out, in)’:
list.m:11:    Error: determinism declaration not satisfied.
list.m:11:    Declared ‘multi’, inferred ‘nondet’.
list.m:16:    in argument 3 of clause head:
list.m:16:    unification of ‘HeadVar__3’ and ‘[H | NT]’ can fail.

draw.m:42: The switch on ‘Shape’ does not cover ‘rhombus’/3.
```

Experience from our project

The Mercury compiler is written in Mercury itself. It is now over 100,000 lines of code in over 150 modules.

Most of this was written by three people (1 f/t, 2 p/t) over two years. This implies productivity around 300 lines per week per person.

By now eleven people have worked on the compiler; mostly students, some only second years. They have found the language easy to learn and to use.

Most modules written by other people are easy to modify. Concurrent changes rarely interfere. New code works first time surprisingly often.

Status

Project started in October 1993. Compiler bootstrapped on 24 February 1995. Latest public release was version 0.6 on 2 August 1996.

The system runs on most Unix systems (Sun, DEC, SGI, IBM, HP, Linux, BSDI) and on Windows 95.

The system is available under the General Public License.

About ten people are working on the system, some part time.

Work in progress

We are working on a declarative debugger. This will use mode information to guide the search.

We are working on more optimizations, including destructive update.

We are working on interfacing Mercury to the Aditi deductive database system, and compiling Mercury to relational algebra operations.

We are starting to explore the integration of CLP(R) into Mercury.

Summary

- useful semantics
- high reliability
- high productivity
- supports team programming
- supports maintenance
- supports formal methods
- supports declarative debugging
- fast code
- compact code
- automatic parallelization feasible
- automatic parallelization profitable
- interface to and from C
- portable
- free

http://www.cs.mu.oz.au/mercury