Unification in Mercury

Zoltan Somogyi
Fergus Henderson
Thomas Conway

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

Mercury

- a new general purpose programming language based on the paradigm of *purely* declarative logic programming

- intended to support the creation and maintenance of large, reliable, efficient applications by teams of programmers

Declarations of the intended behavior of each predicate

- serve as documentation;

- enable the compiler to find most errors in the program;

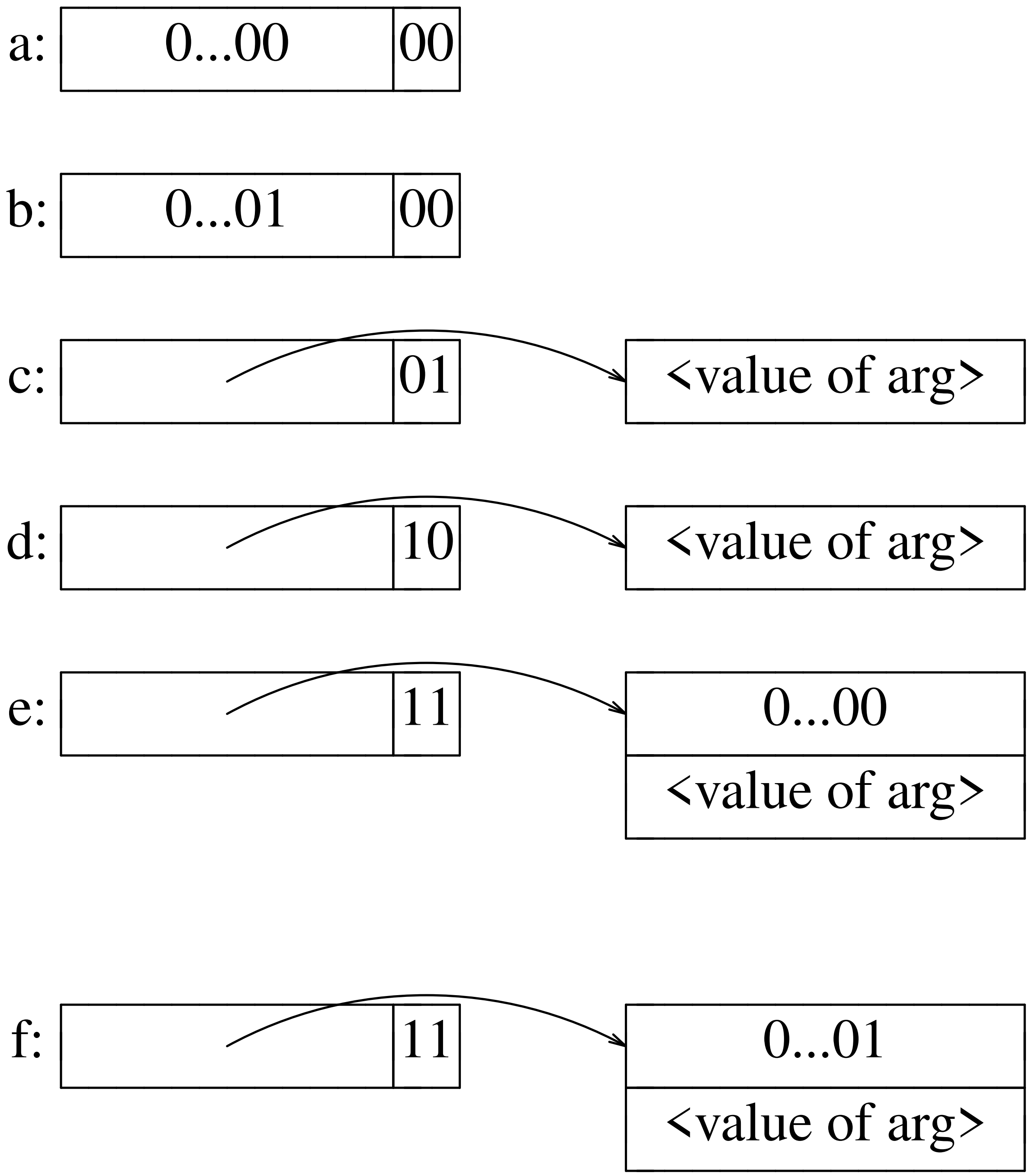
- enable the compiler to generate very good code.

The type system

The Mercury type system is very similar to the type systems of Miranda, Haskell, Godel, and other languages. It supports parametric polymorphism, but not (yet) subtyping or type classes.

```
:- type suit ---> club ; diamond ; heart ; spade.

:- type pair(T1, T2) ---> T1 - T2.

:- type list(T) --->
        []
    ;   [T | list(T)].

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

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

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



The mode system

A mode declaration gives the initial and final instantiation states of all the arguments of a predicate.

The declaration represents a contract between the predicate and its caller. The contract assigns responsibility for the binding of each argument to either the caller or the callee.

```
:- mode in :: ground -> ground.
:- mode out :: free -> ground.

:- 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 mode of a predicate. The separate treatment begins with mode analysis.

Superhomogeneous form

- all atoms have distinct variables as arguments, including the head

- all unifications are of the form X = Y or X = f(Y1, ..., Yn), where the Yi are distinct variables

- each predicate is defined by one well formed formula

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

Mode analysis

Mode analysis initializes the instantiation state of each argument from the mode declaration, performs abstract interpretation on predicate bodies, and checks that the final instantiations of the arguments match the declaration exactly.

If necessary, it reorders conjunctions to ensure that the producer of a variable comes before all its consumers. If this is not possible, it reports an error.

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

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.

Unifications and other builtins 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.

Simple unifications

assignment: X = Y,   X free, Y ground

Code generator records that the locations now holding X also hold Y. When the value of X is needed, it will get it from one of these places.

If the value of X is not needed, or if it is needed in e.g. a register where it is already present, the compiler generates no code.

construction: X = f(Yi),   X free, Yi ground

Code generator records that when the value of X is needed, it should create a cell on the heap containing f's secondary tag (if it has one) and the arguments Yi, and to tag the pointer to this cell with f's primary tag.

Simple unifications

test: X = Y,   X and Y ground and atomic

The compiler emits code to test the words holding X and Y for equality.

deconstruction: X = f(Y*i*),   X ground, Y*i* free

The compiler emits code to test the primary and possibly secondary tag of X. In the success continuation, it records that the values of the Y*i* are available in the various fields of the cell pointed to from X's current location(s).

The compiler does not emit a test if it knows the outcome, e.g. it may be a guaranteed success if X is already known to be equal to Y or bound to f from its type, from indexing, or from an earlier test.

append(in, in, out)

```
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;
```

Complex unifications

X = Y,   not atomic, result ground

For such unifications, the compiler emits a call to a compiler-generated procedure.

Since we use different representations for different types, this procedure must be specific to the type shared by X and Y. The structure of the procedure is derived from the type.

The procedure is available in mode (in, in), as well as any other modes required by the program (synthesized on demand).

```
unify_intlist(A, B) :-
    (
        A = [AH | AT], B = [BH | BT],
        AH = BH, unify_intlist(AT, BT)
    ;
        A = [], B = []
    ).
```

Polymorphism

In the unification procedure for list(T), AH = BH must call the unification procedure for whatever actual type T stands for in this call.

For every type variable a procedure has, the compiler adds an extra argument that at runtime will describe the actual type bound to that type variable.

If the elements of the list are of type tree(int, string), then this *typeinfo* argument will have

- the arity of tree as a type constructor (2)
- pointer to the procedure unify_tree
- pointer to the procedure index_tree
- pointer to the procedure compare_tree
- pointer to the typeinfo for int
- pointer to the typeinfo for string

The compiler has a pass that removes unused arguments, including unused typeinfos.

Higher order

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

p(Item, Pred) :-
        Pred = lambda([S0::in, S::out] is det,
                (set__insert(S0, Item, S))).
```

Pred will point to this closure structure:

- the number of curried arguments (2)
- pointer to the procedure LambdaGoal1
- the curried argument I
- the curried argument TypeInfo_for_T

```
LambdaGoal1(I, TypeInfo_for_T, S0, S) :-
        set__insert(TypeInfo_for_T, S0, I, S).
```

The unification procedures for higher order types cause a runtime abort.

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.

system	time (s)		space
	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

Status

Project started in October 1993. Compiler bootstrapped on 24 February 1995. The 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.