Mercury

The lecturer is Zoltan Somogyi (zs), whose office is at CITRI.

You will get copies of the slides.

The project will be handed out on wednesday or next monday.

The home page of the Mercury project, http://www.cs.mu.oz.au/~zs/mercury.html, contains documentation on the language as well as several papers on the system.

/home/stude/data/247/mercury-0.4 contains the latest Mercury distribution, including sample programs.

Logic programming

The point of basing a programming language on logic is that it makes it easier for programmers to reason about their programs. (Mathematicians have been reasoning about logical formulae for a century.)

This reasoning can also be automated. This opens the door to

- automatic program transformation
- automatic parallelization
- semi-automatic debugging

Such tools promise to improve programming productivity.

Prolog

Prolog was the first logic programming language, and its first implementation was done in 1970.

At that time, computers were slow and memories small, and hence optimizing compilers were rare.

Prolog adopted the approach that a programmer who wanted efficiency would have to tell the Prolog system how to achieve it. This was done through non-logical constructs.

Unfortunately, these constructs destroy the declarative reading of the program.

Mercury

Mercury is a new logic programming language intended to

- realize the promised benefits of logic programming, and
- support the creation and maintenance of large, reliable, efficient application programs by teams of programmers.

The Mercury compiler is written in Mercury itself. It has about 78,000 lines of code, written over two years by three people.

It is the fastest logic programming system by a wide margin (1.9x Aquarius, ~5x SICStus, ~10x Quintus).

Outline of Mercury

- strong polymorphic type system
- strong mode system
- strong determinism system
- module system
- higher-order
- purely declarative (even I/O)

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.

The type system

The Mercury type system is very similar to the type systems of Miranda, Haskell, Godel, and other languages.

In Mercury, all variables have a type. The definition of the type lists the function symbols that these variables may take as their value, and for each function symbol, list the types of the arguments.

The simplest types represent enumerations and what in C would be called structures. More complex types represent unions of structures.

Examples of types

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

:- type card --->
            card(suit, rank).

:- type expr --->
            const(int)
        ;   plus(expr, expr)
        ;   minus(expr, expr).
```

The same word can be used both as a type name and as a function symbol (e.g. card).

Parametric polymorphism

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

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

T, K and V are *type variables*, i.e. variables that stand for a type.

Polymorphic types can be viewed as *type constructors*, in that they construct a new type from some existing types.

Parametric polymorphism yields homogeneous data structures.

Type synonyms

Programmers can introduce new names for old types.

```
:- type enrolmentno == int.
:- type name == string.
:- type mark == int.

:- type student_info --->
        student(name, mark).

:- type student_table ==
        tree(enrolmentno, student_info).
```

The types int and string are built into Mercury.

Types and predicates

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

append([], L, L).
append([H | T], L, [H | NT]) :-
        append(T, L, NT).
```

Mercury requires the programmer to declare the types of all arguments of all predicates.

From this and from the definitions of types, the compiler can infer the types of all the variables in the program, *if* the program is well-typed.

Mode declarations

For each argument of a predicate, a mode declaration says whether the argument is input to the call or output by the call.

A predicate may have more than one mode. We call each mode of a predicate a *procedure*.

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

Input arguments are *consumed* by the procedure, output arguments are *produced* by the procedure.

Mode analysis

The mode analysis phase of the compiler identifies the modes of all calls in the program.

Mode analysis reorders conjunctions to ensure that the producer of a variable comes before all its consumers. This makes it easier to write reversible predicates.

```
:- pred append3(list(T), list(T), list(T), list(T)).
:- mode append3(in, in, in, out).
:- mode append3(out, out, out, in).

append3(A, B, C, D) :-
        append(A, B, X),
        append(X, C, D).
```


Mode analysis

```
% append(in, in, out)
append(A, B, C) :-
    A == [], C := B.
append(A, B, C) :-
    A == [H | T],
    append(T, B, NT),
    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 necessary, 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 compiler generates separate code for each declared mode of a predicate, and the code assumes that output arguments are free variables when the call is made.

Determinism declarations

The determinism of a procedure depends on how many solutions it can produce:

det:	exactly one
semidet:	at most one
multi:	at least one
nondet:	any number

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.
```


Switch detection

Why is append(in, in, out) guaranteed to have exactly one solution?

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

Switch detection looks for disjunctions in which each disjunct unifies an input variable with a different functor; such disjunctions are *switches*.

The switch is complete if the disjuncts cover all the alternatives in the type of the switched-on variable.

Common subexpression elimination

CSE looks for disjunctions in which each disjunct unifies an input variable with the *same* functor.

```
( L = [H1 | T1], T1 = [], ...
; L = [H2 | T2], T2 = [H3 | T3], ...
)
```

It pulls such unifications out of the disjunction; switch detection will then recognize the switch on T1.

```
L = [H1 | T1],
( T1 = [], ...
; H1 = H2, T1 = T2, T2 = [H3 | T3], ...
)
```


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.

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

Of course, we want the implementation to update the state of the world destructively.

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

Uniqueness information

The mode di (destructive input) says that the storage of this argument will be destroyed (reused) in the predicate, so the argument must be the last reference to that storage. The mode uo (unique output) says that there are no other references to the value returned for this argument.

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

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


Uniqueness on backtracking

The compiler should also make sure that variables destroyed on forward execution are not referred to again even on backtracking.

```
(
        io__write_string("testing...", S0, S1),
        1 = 2
->
        io__write_string("1 = 2", S1, S)
;
        io__write_string("1 \= 2", S0, S)
)
```

The current compiler only does a subset of the forward uniqueness tests.

Destructive update

Once the compiler fully implements uniqueness tests, we will start exploiting uniqueness information to generate code that does destructive update. This is necessary to approach the speed of imperative languages.

When appending two lists, one should be able to seek to the end of one list and overwrite the NULL link with a pointer to the other list.

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

append([], L, L).
append([H | T], L, [H | NT]) :-
        append(T, L, NT).
```


Modules

Mercury programs are composed of modules. Each module has an interface section, containing the declarations of items exported from the module, and an implementation section giving their definitions.

If module A wants to use something exported by module B, it must import module B.

A module that exports the name of a type but not its definition represents an abstract data type.

At the moment predicate names must be unique across the whole program.

The standard example

```
:- module hello.

:- interface.
% module io is in the standard Mercury library
:- import_module io.

% main/2 is the standard entry point
:- pred main(io__state, io__state).
:- mode main(di, uo) is det.

:- implementation.

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


mc

The Mercury compiler is called “mc”.

Single-module programs can be compiled simply by “mc module.m”. This will convert the Mercury source to C, compile that to object code, and link it with the Mercury standard library, the Mercury runtime system etc. The executable will be put in the file “module”.

For multi-module programs, one wants to recompile only the minimal set of modules after a change. We have a front-end for make called “mmake” that handles Mercury programs.

mmake

At the start of the project and whenever the inter-module dependencies change, run “mmake mainmodule.depend”. After other changes, run “mmake mainmodule”.

Mmake will recompile a module not only if that module has been modified but also if it imports another module whose interface has changed.

These interfaces are stored in files with .int and .int2 suffixes. These files are created with “mc -i module”.

Mmake puts error messages into files with a .err2 suffix.

System location

The directory /home/mercury/public has two subdirectories, mercury-0.4 and mercury-latest, which contain the last released version and the latest version of the system.

Each of these directories has several subdirectories:

mips-sgi-irix5.2	for mundil
mips-sgi-irix5.3	for muse
sparc-sun-solaris2.3	for Suns

The executables are one level further down, in bin.

File operations

The Mercury standard library supports two approaches to I/O.

In the C-like approach, one can open a file, get back a stream id, and pass it to I/O operations:

```
io__open_output(Filename, Result, S0, S1),
Result = ok(Stream),
io__write_string(Stream, "xyzzy", S1, S2),
...
```

In the Prolog-like approach, there is a current input and current output, and programmers can redirect these with io__see/io_seen and io__tell/io__told.

Reading input

```
io__read_line(Result, S0, S1),
(
    Result = error(_),
    error("Error reading from stdin")
;
    Result = eof
;
    Result = ok(Line),
    ...
)
```

When writing a deterministic predicate, one must prepare for all contingencies. This gives a significant boost to reliability.

Definite Clause Grammars

A clause is in DCG notation if it uses --> instead of :-. When processing a DCG clause, the compiler adds a pair of arguments both to the head of the clause and to every call in the body of the clause not protected by braces.

These extra arguments thread variables representing a state through the predicate.

```
sort_output([]) --> [].
sort_output([Line | Lines]) -->
    { string__from_char_list(Line, LineStr) },
    io__write_string(LineStr),
    sort_output(Lines).
```



```
main -->
    io__command_line_arguments(Args),
    (
        { Args = [] },
        handle_args(no, no),
        sort
    ;
        { Args = [Input] },
        handle_args(yes(Input), no),
        sort
    ;
        { Args = [Input, Output] },
        handle_args(yes(Input), yes(Output)),
        sort
    ;
        { Args = [_, _, _ | _] },
        io__write_string("Usage: ...\n")
    ).
```



```
:- pred handle_args(maybe(string),
    maybe(string), io__state, io__state).
:- mode handle_args(in, in, di, uo) is det.

handle_args(InArg, OutArg) -->
    (
        { InArg = yes(InFilename) },
        io__see(InFilename, InResult),
        (
            { InResult = ok }
        ;
            { InResult = error(InErr) },
            { io__error_message(InErr, InMsg)
            { error(InMsg) }
        )
    ;
        { InArg = no }
    ),
    ...
```



```
sort -->
    sort_2([]).

sort_2(Lines0) -->
    io__read_line(Result),
    (
        { Result = error(Error) },
        { io__error_message(Error, Msg) },
        { error(Msg) }
    ;
        { Result = eof },
        sort_output(Lines0)
    ;
        { Result = ok(Line) },
        { insert(Lines0, Line, Lines1) },
        sort_2(Lines1)
    ).
```

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

insert([], I, [I]).
insert([H | T], I, L) :-
    compare(R, I, H),
    ( R = (<) ->
        L = [I, H | T]
    ;
        insert(T, I, NT),
        L = [H | NT]
    ).
```

Mercury automatically two operations on every type, unification and comparison.

Programmers should not worry about repeated terms such as [H | T].

A simple formatter

```
get_wordlist(Words) -->
    io__read_word(Result),
    (
        { Result = error(Error) },
        { io__error_message(Error, Msg) },
        { error(Msg) }
    ;
        { Result = eof },
        { Words = [] }
    ;
        { Result = ok(Word) },
        get_wordlist(Words1),
        { Words = [Word | Words1] }
    ).
```



```
put_wordlist([], Sofar, _Max) -->
    io__write_string("\n").
put_wordlist([Word | Words], Sofar, Max) -->
    { list__length(Word, WordLen) },
    { NewSofar is Sofar + WordLen + 1 },
    { string__from_char_list(Word, WordStr) }
    ( { NewSofar > Max } ->
        io__write_string("\n"),
        io__write_string(WordStr),
        put_wordlist(Words, WordLen, Max)
    ;
        ( { Sofar > 0 } ->
            io__write_string(" ")
        ;
            []
        ),
        io__write_string(WordStr),
        put_wordlist(Words, NewSofar, Max)
    ).
```


Optimization philosophy

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

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

These apply to Mercury directly. Applying them to other logic languages require extensive program analysis. For acceptable running time, these analyses must make approximations that degrade the effectiveness of the optimizations (e.g. don't know whether this variable is bound here or not).

Some optimizations

Deforestation takes programs with two or more traversals of a data structure and tries to produce an equivalent program with one traversal. Applied to the formatter, it would reduce the maximum storage requirement.

Put_wordlist is naturally tail recursive, but get_wordlist is not. The compiler could transform get_wordlist to be tail recursive by introducing an input argument (called an accumulator) containing the words seen so far in reverse order. The base case reverses this list and returns it as the output of get_wordlist.

Term representation

The representation of a term depends on its type. Enumerations are stored as integers, other types as pointers to cells on the heap with tags in the bottom two bits.

For types with up to four alternatives, the two-bit tag specifies the function symbol directly.

For types with more than four alternatives, some tag values must be shared, either between several constants (distinguished by the rest of the word) or between several non-constants (distinguished by an extra word at the start of the cell).

Mode information

The Mercury compiler generates separate code for each mode of a predicate. The implementations of append(in, in, out) and append(out, out, in) are completely separate.

This does not lead to program size explosion because few predicates (less than 5%) have more than one mode.

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.

Mode information

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.

Assignment unifications can simply overwrite their output variable.

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

Target language

The Mercury compiler generates C code that looks like assembler.

This makes Mercury quite portable with very little effort on our part. It means we don't have to worry about object file formats, instruction sets, etc.

The two drawbacks are increased compilation time and a potential loss of efficiency.

To minimise the efficiency loss, we can exploit some of GNU C's extensions over ANSI C, such as the ability to put a global variable in a register and to take the address of a label.

append(in, in, out)

```
append_3_0:
    incr_sp(2);
    detstackvar(2) = succip;
    if ((r1 == mkword(mktag(0), mkbody(0))))
        goto append_3_0_i4;
    detstackvar(1) = field(mktag(1), r1, 0);
    r3 = r1;
    r1 = field(mktag(1), r3, 1);
    localcall(append_3_0, append_3_0_i5);
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;
    goto append_3_0_i3;
append_3_0_i4:
    r3 = r2;
append_3_0_i3:
    succip = detstackvar(2);
    decr_sp(2);
    proceed();
```


Standard optimizations

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


Middle recursion optimization

```
append_3_0_l:
    if (r1 == mkword(mktag(0), mkbody(0)))
        goto append_3_0_i1;
    r4 = sp;
append_3_0_i4:
    incr_sp(1);
    detstackvar(1) = field(mktag(1), r1, 0);
    r1 = field(mktag(1), r1, 1);
    if (r1 != mkword(mktag(0), mkbody(0)))
        goto append_3_0_i4;
    r3 = r2;
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;
    decr_sp(1);
    if (sp > r4)
        goto append_3_0_i5;
    proceed();
append_3_0_i1:
    r3 = r2;
    proceed();
```


Jump optimizations

Mercury is an easy language in which to write optimizers, as shown by the following extract from the Mercury compiler.

```
:- type instruction == pair(instr, string).
:- type blockmap == map(label, list(instruction)).

:- pred optimize_jumps(list(instruction), list(instruction)).
:- mode optimize_jumps(in, out) is det.

optimize_jumps(Instrs0, Instrs) :-
    map__init(Blockmap0),
    build_maps(Instrs0, Blockmap0, Blockmap),
    jumpopt(Instrs0, Blockmap, Instrs).

:- pred build_maps(list(instruction), blockmap, blockmap).
:- mode build_maps(in, in, out) is det.

:- pred jumpopt(list(instruction), blockmap, list(instruction)).
:- mode jumpopt(in, in, out) is det.
```



```
build_maps([], Blockmap, Blockmap).
build_maps([Instr0 | Instrs0], Blockmap0, Blockmap) :-
    ( Instr0 = label(Label) - _Comment0 ->
        find_no_fallthrough(Instrs0, Block),
        map__det_insert(Blockmap0, Label, Block,
            Blockmap1)
    ;
        Blockmap1 = Blockmap0
    ),
    build_maps(Instrs0, Blockmap1, Blockmap).

:- pred find_no_fallthrough(list(instruction), list(instruction)).
:- mode find_no_fallthrough(in, out) is det.

find_no_fallthrough([], []).
find_no_fallthrough([Instr0 | Tail0], Instrs) :-
    (
        Instr0 = Uinstr0 - _Comment0,
        can_instr_fall_through(Uinstr0, no)
    ->
        Instrs = [Instr0]
    ;
        find_no_fallthrough(Tail0, Tail1),
        Instrs = [Instr0 | Tail1]
    ).
```



```
:- pred can_instr_fall_through(instr, bool).
:- mode can_instr_fall_through(in, out) is det.

can_instr_fall_through(comment(_), yes).
can_instr_fall_through(livevals(_), yes).
can_instr_fall_through(block(_, _), yes).
can_instr_fall_through(assign(_, _), yes).
can_instr_fall_through(call(_, _, _, _), no).
can_instr_fall_through(call_closure(_, _, _), no).
can_instr_fall_through(mkframe(_, _, _), yes).
can_instr_fall_through(modframe(_), yes).
can_instr_fall_through(label(_), yes).
can_instr_fall_through(goto(_), no).
can_instr_fall_through(computed_goto(_, _), no).
can_instr_fall_through(c_code(_), yes).
can_instr_fall_through(if_val(_, _), yes).
can_instr_fall_through(incr_hp(_, _, _), yes).
can_instr_fall_through(mark_hp(_), yes).
can_instr_fall_through(restore_hp(_), yes).
can_instr_fall_through(incr_sp(_), yes).
can_instr_fall_through(decr_sp(_), yes).
```

If someone adds a new alternative to the type “instr”, this predicate will get a determinism error as a reminder.


```
jumpopt([], _Blockmap, []).
jumpopt([Instr0 | Tail0], Blockmap, Instrs) :-
    jumpopt(Tail0, Blockmap, Tail1),
    (
        Instr0 = goto(label(Label)) - _Comment0,
        map__search(Blockmap, Label, Block)
    ->
        filter_out_labels(Block, NewInstrs),
        list__append(NewInstrs, Tail1, Instrs)
    ;
        Instrs = [Instr0 | Tail1]
    ).

:- pred filter_out_labels(list(instruction), list(instruction)).
:- mode filter_out_labels(in, out) is det.

filter_out_labels([], []).
filter_out_labels([Instr0 | Tail0], Instrs) :-
    filter_out_labels(Tail0, Tail1),
    ( Instr0 = label(_) - _ ->
        Instrs = Tail1
    ;
        Instrs = [Instr0 | Tail1]
    ).
```

Performance results

prog	NP	SP	AP	Mer
cqueen	1.6	6.1	25.9	49.1
crypt	1.4	6.0	7.6	29.1
deriv	1.5	6.8	15.8	25.8
nrev	3.4	12.1	36.9	40.2
poly	1.8	7.9	18.9	40.0
primes	1.7	5.7	12.9	23.6
qsort	1.6	6.9	32.8	39.8
queen	33.8	30.8	26.1	33.9
query	1.9	4.4	18.2	40.2
tak	2.2	11.6	94.2	116.8
h. avg	1.9	7.4	19.0	36.5

Performance results

The compiler is written mostly in the intersection of NU-Prolog, SICStus Prolog and Mercury, and can serve as a large benchmark.

Compile time is the time to compile the compiler. File size is the size of the executable or save file. Run time is the time to compile a medium-size file.

	NP	SP	Mer
compile time (m)	8.4	13.6	61.8
file size (Mb)	1.9	7.4	2.2
run time (s)	116	71.1	18.5

Status

Project started in October 1993. Compiler bootstrapped on 24 February 1995. Public releases on 18 July and 15 September 1995.

The compiler and standard library consist of about 79,000 lines of Mercury code totaling 2.6 Mb. The runtime system is 5,000 lines of C.

The system runs on Suns, SGIs, DECstations, Alphas and Linux PCs.

A profiler now exists. Higher order predicates have been implemented. We do limited checking of uniqueness declarations.

Work in progress

We are working on a native garbage collector. At present we use a conservative garbage collector for C.

We are working on making it easier to interface Mercury code with C code, not least to get access to C libraries (e.g. tk).

We are working on a source-level debugger.

We are working on more optimizations, both source level and low level.

During the summer we will explore the integration of CLP(R) into Mercury.