The Mercury Programming Language
Paul Bone
based on slides by Zoltan Somogyi
Mercury from 30,000ft

Mercury is a purely declarative logic/functional programming language. It is aimed at *programming in the large*.

Mercury *looks* like Prolog, but it *feels* like strict Haskell or pure OCaml.
Goals

Declarative programming (for example in Prolog) has always been very powerful. However creating large pieces of software is difficult.

We aim to make programming in the large easier:

• large programs
• large teams
• better program reliability
• better program maintainability
• program efficiency
This talk

I do not have enough time to teach you Mercury.

Instead I will aim to give a guided tour of Mercury's main features, and how it differs from languages you may be familiar with.

Time permitting I will introduce two very cool Mercury technologies:

• declarative debugging, and
• automatic parallelization.

The *purity* of Mercury is key in making both of these feasible.
Some syntax

fibs/2 is the hello world of declarative programming. \( F \) is the \( N \)th Fibonacci number.

\[
\text{fibs}(N, F) :- \\
( \ N < 2 \rightarrow \\
F = 1 \\
; \\
\text{fibs}(N - 1, FA), \\
\text{fibs}(N - 2, FB), \\
F = FA + FB \\
).
\]

**Predicates** do not have return values per-se. They are either *true* or *false* for a given set of arguments. Arguments may be either *input* or *output*. 
Some syntax

fibs/2 is the hello world of declarative programming. \( F \) is the \( N \)th Fibonacci number.

\[
fibs(N, F) :-
  \begin{align*}
    & ( N < 2 \rightarrow \\
    & \quad F = 1 \\
    & ) ; \\
    & \quad \text{fibs}(N - 1, FA), \\
    & \quad \text{fibs}(N - 2, FB), \\
    & \quad F = FA + FB
  \end{align*}
\]

A clause is made up of goals and goals can be conjoined (logical AND) with a \( , \) and disjoined (logical OR) with a \( ; \).
Some syntax

fibs/2 is the hello world of declarative programming. $F$ is the $N$th Fibonacci number.

```prolog
fibs(N, F) :-
    ( N < 2 ->
        F = 1
    ;
    fibs(N - 1, FA),
    fibs(N - 2, FB),
    F = FA + FB
    ).
```

This code also uses an *if-then-else* which joins three goals:

\[ \text{Condition} \rightarrow \text{Then} ; \text{Else} \]
Purity

Imperative programs are based on side effects. You call a function such as

```c
strcat(str1, str2);
```

and it returns a value, but the *reason* you call it is for its side effect (modifying str1).
Purity

Purely declarative programs have *no side effects*. If a predicate has an *effect*, it has to be reflected in its argument list.

```
hello(IO0, IO) :-
    io.write_string("Hello, ", IO0, IO1),
    io.write_string("world\n", IO1, IO).
```

Because a predicate can return more than one item, it is easy to work with more than one *state*.

In purely declarative languages, data structures are *immutable*. Instead of updating an existing data structure, programs create slight variants of existing data structures, typically reusing *almost all their memory*. 
Purity

Typing out all the intermediate versions of a value can become tedious.

```prolog
hello(I00, I0) :-
    io.write_string("Hello, ", I00, I01),
    io.write_string("world\n", I01, I0).
```

So we created a useful syntactic sugar:

```prolog
hello(!I0) :-
    io.write_string("Hello, ", !I0),
    io.write_string("world\n", !I0).
```

It is now easy to update this code without renumbering all the variables.
Types

Mercury has a *strong, static* type system similar to Haskell's.

There are several built-in types (int, float, char...). Developers can define new types easily.

:- type playing_card
   ---> normal_card(
       c_suit :: suit,
       c_num :: int
    )
   ;       joker.

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

A predicate's arguments' types are declared in its pred declaration.

:- pred fibs(int, int).

fibs(N, F) :-
    ( N < 2 ->
        F = 1
    ;
        fibs(N - 1, FA),
        fibs(N - 2, FB),
        F = FA + FB
    ).
Modes

The basic modes are in and out.

:- pred fibs(int, int).
:- mode fibs(in, out).

When there is a single mode for a predicate we can write this more succinctly:

:- pred fibs(int::in, int::out).
Modes

in and out are defined as:

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

Where free means doesn't have a value, and ground means has a value. These are instantiation states.
Modes

Modes can also be used to track uniqueness. These modes are di for destructive input and uo for unique output.

:- pred hello(io::di, io::uo).

They are defined as:

:- mode di == unique >> clobbered.
:- mode uo == free >> unique.

clobbered means that the memory that used to contain a value has been written-over: a program cannot read its value. In practice the io type is special: it is optimised away and doesn't consume any memory.
Genealogy Example

:- pred mother(person, person).
:- mode mother(in, out).
:- mode mother(out, in).

mother(paul, faye).
mother(james, faye).

This predicate has two modes. We can call it in either mode:

- **Who is Paul's mother?**
  
  mother(paul, M)

- **Who is Faye the mother of?**
  
  mother(C, faye)
Determinisms

:- pred mother(person, person).
:- mode mother(in, out) is det.
:- mode mother(out, in) is nondet.

mother(paul, faye).
mother(james, faye).

The second mode may have multiple answers or none at all so it it nondeterministic, we indicate this with is nondet.

The first mode has exactly one answer it is det.

Fun Science Fact: A person may have two biological mothers when they have their normal DNA from one woman and their mitochondrial DNA from another woman.
Disjunction syntax

These are equivalent:

mother(C, M) :-
    M = faye,
    ( C = paul ; C = james ).

mother(paul, faye).
mother(james, faye).

Whether you write one clause containing disjunctions or multiple clauses depends on the kind of thing you're expressing. In this case using facts (clauses without bodies) is clearer.
Determinisms

There are six basic determinisms. They form a matrix based on what they allow

<table>
<thead>
<tr>
<th>cannot fail</th>
<th>erroneous</th>
<th>at most zero solutions</th>
<th>at most one solution</th>
<th>no limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>can fail</td>
<td>failure</td>
<td>det</td>
<td>semidet</td>
<td>nondet</td>
</tr>
</tbody>
</table>

The two remaining determinisms cc_multi and cc_nondet are used in committed choice contexts.
Geneology Example

**Nondeterministic search** can be a very useful programming tool:

```prolog
mother(paul, faye).
mother(james, faye).

parent(C, P) :-
    mother(C, P).
parent(C, P) :-
    father(C, P).

sibling(A, B) :-
    parent(A, P),
    parent(B, P).
```

Mercury generates specialised code for each **mode** of each **predicate**.
Using predicates and modes we can now make IO safe in this purely declarative language.

```mercury
:- pred main(io::di, io::uo) is det.
main(!IO) :-
    write_string("Hello ", !IO),
    write_string("World\n", !IO).
```

- The **mode system** ensures that we can't reference an old version of IO (because it's **clobbered**)
- the **determinism system** ensures that we cannot **backtrack** over IO (because code is **det**)

The Mercury Programming Language
An old version of a state variable

An old version of !I0 that's impossible (mode system). But an old version of some other !StateVariable, that's easy!

!Name
stands for both current and next variables,

!.Name
stands for only the current variable

!(:Name
stands for only the next variable
An old version of a state variable

An old version of !IO that's impossible (mode system). But an old version of some other !StateVariable, that's easy!

Before we do something complex, we can save a copy of the state in the state variable.

\[
\text{SavedState} = !.\text{MyProgramsState},
\]

Now, if the user clicks "undo" I can restore that old state.

\[
!:\text{MyProgramsState} = \text{SavedState}
\]
Higher order programming

Mercury supports the usual **higher order** programming features.

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

:- pred map(pred(T, U), list(T), list(U)).
:- mode map(pred(in, out) is det, in, out) is det.

map(_, [], []).  
map(P, [X | Xs], [Y | Ys]) :-  
P(X, Y),  
map(P, Xs, Ys).
```

The Mercury Programming Language