The Mercury project

Zoltan Somogyi
The University of Melbourne

Linux Users Victoria

7 June 2011

Mercury: objectives

Classic AI languages like Lisp and Prolog are well suited for exploratory programming, where flexibility and quick turn-around time are the most important objectives.

Mercury is a declarative programming language, but it has different objectives:

- program reliability
- program maintainability
- scalability to large programs and teams
- programmer productivity
- program efficiency

This talk

I do not have enough time to give a tutorial on Mercury (one is available on the Mercury web site).

I will instead tell you about two very useful technologies that are available in Mercury, but not in imperative languages like C and Java.

These two technologies are

- declarative debugging, and
- automatic parallelization.

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

Purity

Imperative programs are based on side effects. You call a function such as `strcat(str1, str2)`, and it returns a value, but the *reason* you call it is for its side effect.

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

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

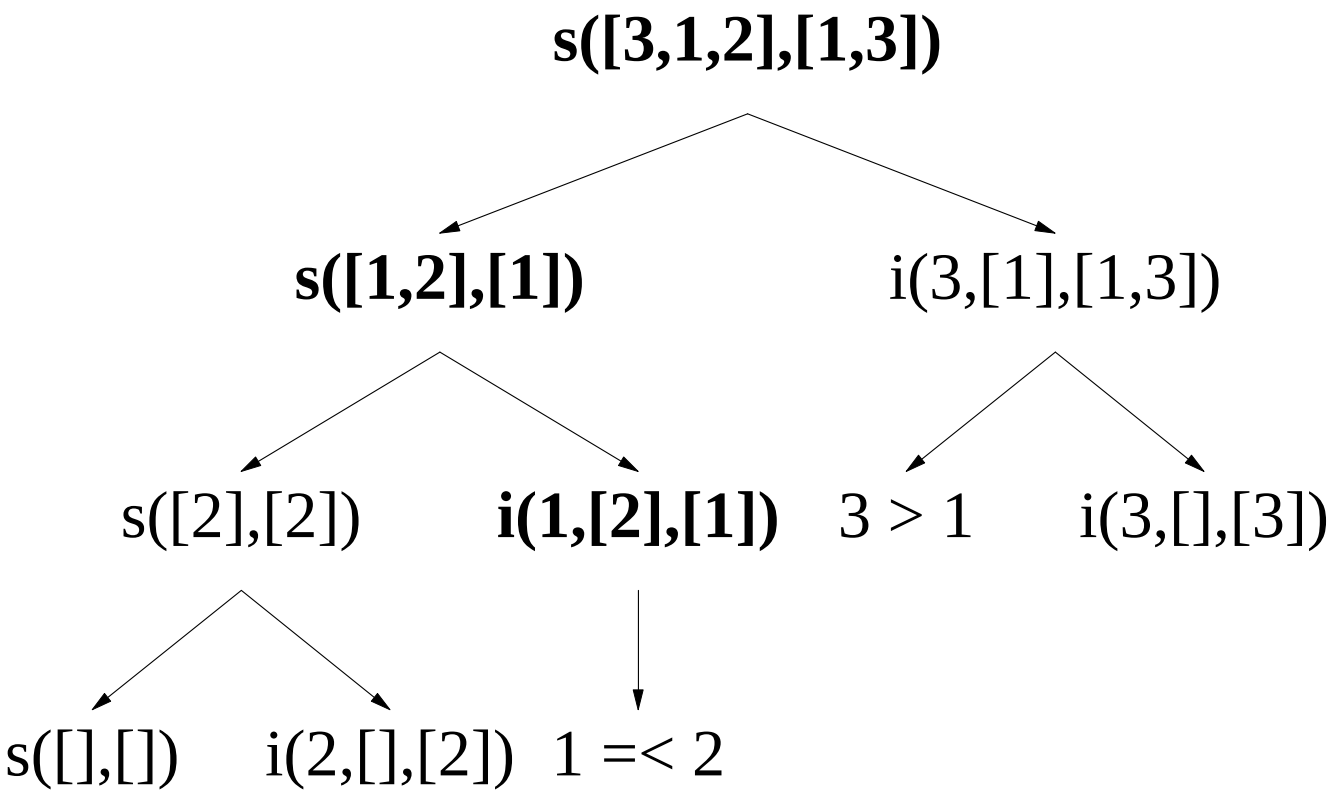
In purely declarative languages, data structures are immutable. Instead of updating an existing data structure, programs create slight variants of existing data structures, typically reusing almost all their memory.

Declarative debugging

- The execution of the program is represented by a tree: each node corresponds to a procedure call.
- The children of each node are the calls to functions / predicates / procedures / methods in the body of the parent.
- A node is valid if its actual result matches its intended result; otherwise, it is erroneous.
- A bug is an erroneous node with no erroneous children.
- The declarative debugger searches for a bug by asking questions about the validity of nodes.

Example

The *proof tree* for insertion_sort([3,1,2], [1,3]):



Example

```
mdb> p
insertion_sort([3, 1, 2], [1, 3])
mdb> dd
insertion_sort([3, 1, 2], [1, 3])
Valid? n
insertion_sort([1, 2], [1])
Valid? n
insertion_sort([2], [2])
Valid? y
insert(1, [2], [1])
Valid? n
The bug occurs during the call
insert(1, [2], [1])
```

Advantages of declarative debugging

- It automates recordkeeping: users need not keep a record in their heads of everything they have done.
- It avoids repeated testing of hypotheses, which puts an upper bound on the effort required to find the bug.
- The computer directs the bug search, not the programmer. This is especially good for novices, who often don't know where to begin.
- The search algorithms programmed into the declarative debugger can automate heuristics followed by humans who are expert debuggers.

Why is it not widely used?

Reason 1

The "Is this valid?" question must describe exactly what "it" is.

- For a pure predicate or function in Mercury, this is given by the values of the input and output arguments.
- For a Mercury predicate that does I/O, it also includes the sequence of executed I/O primitives.
- For a function or method in an imperative program, it also includes the value of *every global variable in the program* that the function or method has access to, directly or indirectly.

Therefore for the questions to be of a manageable size, the program must be declarative.

Why is it not widely used?

Reason 2

The tree does not fit in memory for computations that run for more than a second or two.

The only option is to keep only the parts of the tree needed right now, recomputing the other parts on demand by reexecuting the calls that generate them. This requires rebuilding the state of the computation at the time of the call.

In Mercury, data structures are immutable; you *never* need to restore them to a previous state. Mercury does need to remember the results of previously executed I/O actions. Being relatively rare, this is feasible.

In imperative language programs, *every assignment statement* destroys earlier state. Recording every assignment in an undo/redo log is possible in theory, but infeasible in practice except for toy programs.

Parallelization and side effects

Can you execute these two C calls in parallel?

```
c = p(a, b);
f = q(d, e);
```

Answering the question requires computing *all* the side-effects that `p` may have, and figuring out whether `q` is affected by *any* of them.

In multi-module programs using libraries, this is very hard to do.

What about these two Mercury calls?

```
p(A, B, C), % produces C
q(D, E, F)  % produces F
```

There is no need for any program analysis, and the answer is *yes*.

Overheads and dependencies

```
p(A, B, C, G),  % produces C and G
q(C, D, E, F)   % produces F
```

Can you execute these two calls in parallel? Yes, of course. The compiler will handle the synchronization on `C`.

Should you execute these two calls in parallel? Answering that is harder.

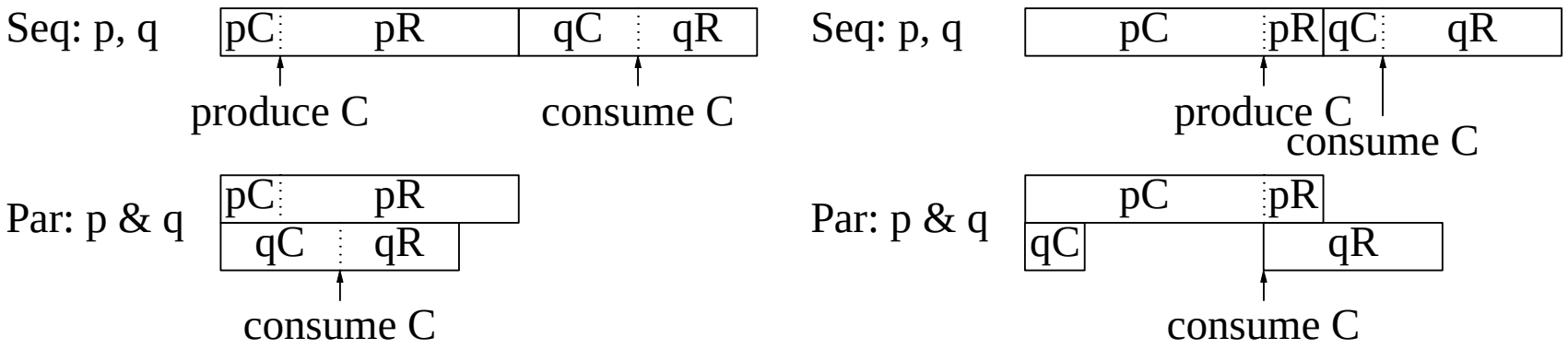
First, we need to know how expensive the two calls are. If e.g. the call to `q` executes in twenty instructions, there is no point in spawning it off, since the spawn operation itself executes a few hundred instructions.

Second, we need to know *when* `q` needs the value of `C`. There is no point in spawning it off if it will immediately block waiting for `C` (the usual case).

The main problem in parallelizing declarative programs is too *much* parallelism, not too little.

Execution overlap

The overlap between the executions of p and q depends on when shared variable C is produced and consumed.



In the scenario on the left, executing p and q in parallel yields the best possible speedup.

In the scenario on the right, parallel execution looks like yielding a smaller speedup, and may yield a slowdown in practice due to overheads.

Estimating execution overlap

The Mercury system has a profiler than can tell us not just the average time taken by calls to a given a predicate or function, but also the average time taken by a given *call site*.

The compiler also arranges for programs being profiled to put a description of their own structure into the profiling data file. This includes complete information about where the value of each variable is produced and consumed.

Our automatic parallelization tool can use this information to estimate the times taken by `p` and `q`, both in total, and up to the production or first consumption of `C`.

From this, one can estimate the speedup from executing `p` and `q` in parallel.

Automatic parallelization workflow

- Compile the program with profiling enabled.
- Run the program on some typical input. This generates a profiling data file.
- Invoke our automatic parallelization tool. This tool
  - reads the profiling data file,
  - finds conjunctions containing two or more expensive calls,
  - estimates the speedups from of parallelizing them in many possible ways (1: `p & q & r`, 2: `(p & q), r`, etc)
  - if the best way yields a nontrivial speedup *even when taking estimated overheads into account*, then record a recommendation for this parallelization,
  - put all the recommendations into a feedback file.
- Compile the program asking for automatic parallelization, specifying the feedback file.

Results

Program	Seq	1 CPU	2 CPUs	3 CPUs	4 CPUs
matrixmult	11.0	14.6 (0.75)	7.5 (1.47)	6.2 (1.83)	5.2 (2.12)
raytracer	22.7	25.1 (0.90)	16.0 (1.42)	11.2 (2.03)	9.4 (2.42)
mandelbrot	33.4	35.6 (0.94)	17.9 (1.87)	12.1 (2.76)	9.1 (3.67)

Parallel code needs to use a machine register to point to thread-specific data, so enabling parallel execution but not using it leads to slowdowns.

Matrixmult has one memory store for each FP multiply/add pair. Its speedup is limited by memory bus bandwidth, which it saturates relatively quickly.

Raytracer generates many intermediate data structures. The gc system consumes 40% of the execution time in stop-the-world collections on 4 CPUs (it is parallel, but does not always scale well).

Conclusion

In our experience, programming in Mercury is significantly more productive than programming in imperative languages.

Programs written in Mercury are guaranteed to be free of several classses of bugs that can occur in most other languages.

Programmers working in Mercury have powerful tools they can use to chase down any bugs not caught by the compiler.

We expect that Mercury programmers will soon have tools they can use to help them parallelize programs with relatively little effort.

For more information about the Mercury project, visit `http://www.mercury.csse.unimelb.edu.au/`.

Any questions?

Powerful type system

The type system used by Mercury is both safe and flexible. It allows programmers to describe their intentions *much* more closely than the type systems of languages like C and Java.

```
:- type token
    --->    ident(string)
    ;       int_const(int)
    ;       float_const(float)
    ;       left_paren
    ;       right_paren
    ;       ...
```

You cannot access e.g. the float without checking that the token is a float_const.

Weak type system

```
typedef enum { IDENT, INT_CONST, FLOAT_CONST,
    LEFT_PAREN, RIGHT_PAREN, ...}
    TokenKind;

typedef struct {
    TokenKind   kind;
    char        *name;
    int         int_const;
    float       float_const;
    ...
} Token;
```

You *can* access token.float_const without checking that token.kind == FLOAT_CONST.

Some bug types

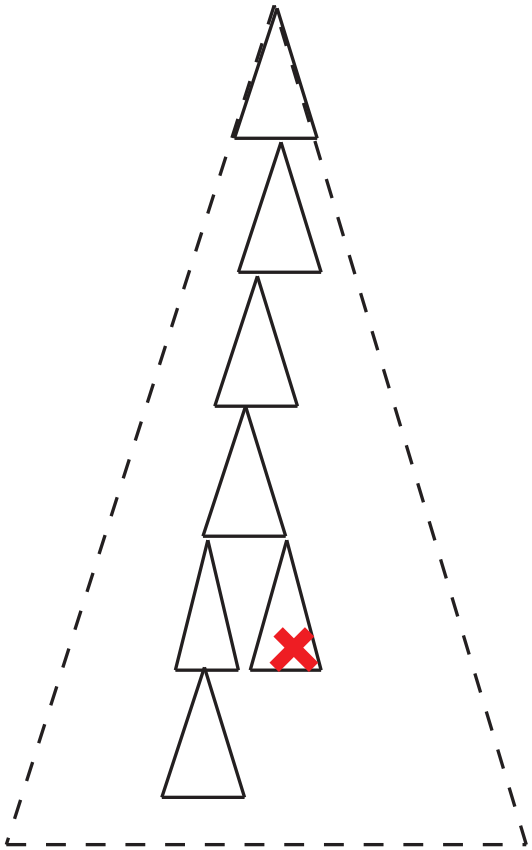
- Memory allocation errors. Cannot be expressed in Mercury.
- Pointer arithmetic errors. Cannot be expressed in Mercury.
- Can-be-null vs cannot-be-null confusion. Cannot arise in Mercury.
- Uninitialized variables. The Mercury compiler is guaranteed to catch this.
- Uninitialized fields in structures. The Mercury compiler is guaranteed to catch this.
- Uncovered cases in a switch. The Mercury compiler is guaranteed to catch this.
- Incorrect casting (C) or dynamic type tests (Java). Casts do not exist, and the need for dynamic type tests can be trivially avoided in almost all cases.
- Use of union or structure fields when not valid. Can be trivially avoided in almost all cases.

Why is it not widely used?

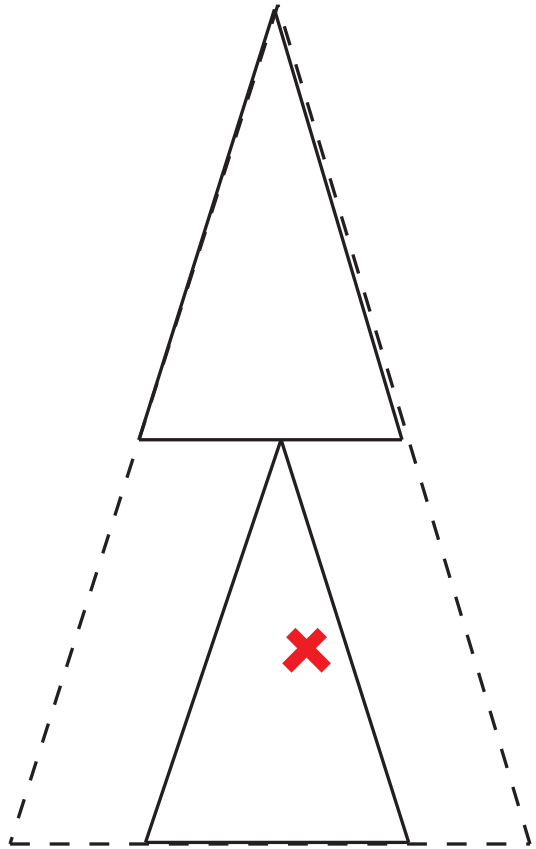
Some other reasons:

- The search algorithm may ask too many questions.
  We have implemented better search algorithms finding bugs with fewer questions. Some focus on a user-selected wrong *part* of the answer, while some others use coverage information from successful vs failed test cases to focus on parts of the program that were executed relatively more frequently during failed test cases.
- The questions asked may be difficult to answer.
  The Mercury declarative debugger has tools to make it easier, and allows users to skip questions if they wish to.

Tree materialization on demand



Less memory, more time



More memory, less time

Updating a tree