Franz Pfluegl, Fotolia

Introducing the Mercury declarative, logical-functional, OO programming language

OUT OF THE LAB

The Mercury programming language offers the expressive power of logic programming with the performance of an imperative language like C or C++. **BY NICK RUDNICK**

Australia, the land of many unusual and spectacular creatures, has brought forth yet another sensational beast. But before natural history buffs start running for their cameras, let me explain that it was computer scientists at the Melbourne University Campus [1] who created this new exotic creature that is the programming equivalent of an egg-laying mammal.

This new language, Mercury, includes features associated with imperative languages like C and C++, but also contains features of functional and logic programming languages like Haskell or Prolog. According to the Mercury project, the reason for developing Mercury was that although logic programming languages offer several powerful benefits for the programmer, logic languages suffer from two significant disadvantages:

- compilers for logic programming languages pick up fewer errors at compile time;
- programs written in logic programming languages tend to run slower than programs written in imperative languages like C.

The Mercury project is an attempt to provide the advantages of a logic programming language without the penalties in run-time efficiency, reliability, and manageability. Mercury is a 100 percent declarative language. According to the

Benchmarking

Our Fibonacci listing (Listing 1) is a simple benchmark for measuring the relative performance of programming languages – you can just enter *time ./fibonacci*. The average time on my lab machine was more or less eight seconds, whereas the Java JDK 1.5 took seven seconds, and a corresponding C program (GCC 4.1) took ten seconds. These results cautiously hint that Mercury is a front-runner in traditional application fields.

In fact, thanks to full declarativity, Mercury enjoys one of the world's most advanced profiling tools, the deep profiler *mdprof*. It provides much more content info for measurements than conventional graph profilers and allows profiling space and time in the same run.

For more rigorous benchmarks, check out the Mercury website [1]. The comparison with Prolog-style languages is slightly older, but at the time Mercury was 24 to 116 times faster than SWI Prolog and 3 to 10 times faster than SICStus Prolog (but only marginally faster than SICStus at solving the queens problem). Another benchmarking paper with constraint solving by Becket et al can also be found at the Mercury site [6]. Even though these results were published by the Mercury team, Mercury's performance seems plausible on basis of the distinct scientific concepts.

Mercury project, Mercury offers “strong type and mode systems that detect a large percentage of program errors at compile time” [3]. This sophisticated beast supports working with higher order logic, and Mercury includes a variety of options for object-oriented constructs (such as design patterns). According to all my reports, Mercury is by far the fastest language with respect to logic and constraints, and Mercury is an excellent player in the “fastest language” league. It is suitable for highest speed neuronal networks (with conventional hardware), has exemplary compile-time error detection and profiling, and provides an architecture designed for large-scale projects with hundreds of thousands of lines of code.

To achieve this ambitious collection of attributes, Mercury makes extensive use of *modes*. This concept will be familiar to Corba, IDL, or Ada programmers who use modes to define the read and write semantics of variables, in addition to the type. Mercury draws its strength from a similar, but far more fundamental, system that creates a descriptive dimension orthogonal to types.

The concept of strong modes – and Mercury therewith – was developed by Zoltan Somogyi [2] in the late 1980s. Parallel work on linear types by Philip Wadler [5], creator of the Haskell programming language, reveals that this approach offers benefits over classical logic, relating on a deeper layer discovered by French logician Jean-Yves Girard [4] known as linear logic, which is considered very computer friendly.

Getting Started

Although many Linux distributions include the Mercury source code, it is a good idea to download the source from the Mercury project website [1] and manually build it using *./configure*, *make*, *make install*. The compile can take a while even on fast hardware. To compensate for this, the install process sets up multiple compiler grades for different usages like compiling, debugging, parallelism, etc. These can also be manually disabled during compilation. Don’t forget the mercury-extras, which you can compile with Mercury by running *mmake depend*, *mmake*, *mmake install*.

Now it is a good idea to adapt your PATH, type (see Listing 1), and create your first Mercury executable with *mmake fibonacci.depend* and then *mmake fibonacci*. Mercury programming is home ground for Linux users, so don’t bother looking for a mouse-pushing front-end. Mercurians tend to prefer text-based editors such as vi and (X)Emacs (see Figure 1).

Figure 1: Colors of Mercury with XEmacs as an example.

Modes and Predicates

The predicate concept of logic programming might take some explaining for programmers more familiar with imperative languages. If you can imagine functions or methods as precisely defined units for converting input into output variables, then predicates are just a more relaxed system for handling incomplete data. You don’t need to define which variable defines which other variable up front. Instead, you just string a loose group of conditions together and let the system decide how everything works. SQL queries follow a fairly similar approach.

The idea of writing whole programs in this way – leaving the question of how to execute a program up to the machine – was what powered the Prolog craze at the end of the last century. In fact, this logic applies a number of constraints; thus, Prolog does not fulfill the promise of its design in real-life applications. The

Listing 1: fibonacci.m

```
:-module fibonacci.
:-interface.
:-import_module io.

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

:-func fib(int)= int.

:-implementation.
:-import_module int, list, string.

main(!IO) :-
 command_line_arguments( Args, !IO),
 (if Args = [Arg|_] then
   Number = det_to_int(Arg),
   format("Fibonacci number for %d is %d\n",
         [ i(Number), i(fib(Number)) ],
         !IO)
 else
   true
 ).

fib(Number)= (
 if Number < 2 then
   1
 else
   fib(Number - 1) + fib(Number - 2)
 ).
```

renunciation of this strategy is one of Mercury's key characteristics.

One of Mercury's defining properties is that it uses modes throughout. Other languages, like IDL or Ada, use modes too, but they enter a completely new syntactical dimension in Mercury. Four modes are all you need to know to start. *in* and *out* represent data that enters and leaves, respectively, a block of code. Mercury applies strict distinctions here – an *inout* mode does not exist.

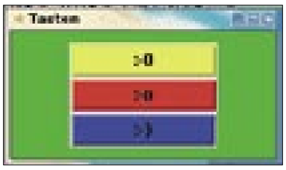

Figure 2: GUI building blocks from Tk are perfect with Mercury.

But Mercury reveals its true nature by two other modes that occur in the *mode* declaration for *main/2* in Listing 1 (line 6): *di* (destructive input) means that all other references to the data unit must be destroyed on entering the code block, and *uo* (unique output) means that the data unit from the code block can only be passed on exactly once. Used in combination, these modes ensure the unique occurrence of specific data units.

The most spectacular application of unique modes is input/output, which is handled declaratively, in contrast to other Prolog-style programs.

The power of threading reveals itself in the expression *!IO* – pure syntactic sugar that avoids the need to enumerate variables:

```
main(IO_0, IO_2) :-
   write_string( "First\n", ➥
   IO_0, IO_1),
   write_string( "Second\n", ➥
   IO_1, IO_2)
```

Generally speaking, unique modes support the use of critical "procedural" language constructs without sacrificing the declarative nature of the language.

The Type System

Mercury's type system is closely modeled on typed functional languages. To help explain this, consider the discriminated union *type* constructor, which C programmers will recall as a mixture of enumeration, union, and struct. It is a semicolon-separated collection of identifiers, which in turn can contain other fields as parameters:

```
:-type address --->
     address (street :: string,
     zip :: int,
     city:: string);
     always_on_the_road;
     unknown.
```

The names allow access, just like getters and setters in object-oriented languages. Of course, the Mercury type system supports generics (templates in C++). The following example shows a *type* constructor for lists of self-definable element types:

```
:-type list(Type) --->
         [] ;  % empty
         [ Type | list(Type) ] .
```

The next, more lengthy, example contains a predicate *append/3* for lists in which the third argument has to correspond to the concatenation of the first two arguments. Prolog very often fails to put its money where its mouth is and simply defines the following:

```
append([], List, List).
append([X|LXs], RXs, [X|Xs]) :-
   append(LXs, RXs, Xs).
```

This is quite problematic as the construct implies nonsensical and/or unintended modes such as *append(out, out, out)*. Also, this construct cannot be efficient for all modes equally; it is very expensive for *append(out, in, in)* when the left sublist must be determined. In contrast, Mercury can solve the problem correctly with the use of different predicates for these special cases (Listing 2).

All of these mode declarations mean a lot of typing, of course, but you are highly unlikely to find a real-life application in which you need every single

Listing 2: append/3

```
:-interface.

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

:-implementation.

:-pragma promise_pure(
        append/3).
append(
  LXs::in, RXs::in, List::out) :-
    append_1(LXs, RXs, List).
append(
  LXs::in, RXs::out, List::in) :-
    append_1(LXs, RXs, List).
append(
  LXs::out, RXs::out, List::in) :-
    append_1(LXs, RXs, List).
append(
  LXs::out, RXs::in, List::in) :-
    append_2(LXs, RXs, List).

:-pred append_1(
  list(T),list(T),list(T)).
:-mode append_1(in, in, in)
    is semidet.
:-mode append_1(in, in, out)
    is det.
:-mode append_1(in, out, in)
    is semidet.
:-mode append_1(out, out, in)
    is multi.
append_1([], List, List).
append_1(
  [X|LXs], RXs, [X|Xs]) :-
    append_1(LXs, RXs, Xs).

:-pred append_2(
  list(T),list(T),list(T)).
:-mode append_2(out, in, in)
    is semidet.
append_2(LXs, RXs, List) :-
   list.remove_suffix(List, RXs, LXs).
```

Listing 3: cget (embedded Tcl/Tk)

```
cget_string(Tcl, Widget,
    ConfName, Ergebnis, !IO) :-
  unwrap(Widget, WidgetId),
  eval(Tcl,
    WidgetId++" cget -"++ConfName,
    Success, Results, !IO),
  (if Success = tcl_ok then true
  else error(Results) ).
```

mode combination. Thus, typically, strong modes enhance reliability and performance at a look and feel more familiar to non-logical programming.

Mercury has an amazingly elegant approach to handling an unknown number of solutions. The *solutions* function can discover the number of solutions for a predicate that is not *det* in, say, a list of unique entries. It also supports higher order logic; in other words, functions or predicates themselves can be arguments in other functions or predicates.

Mercury's higher order logic equipment by no means is a half-hearted gimmick, but on a level with languages such as Haskell, Caml, and SML. In fact, I went through a number of exercises in popular textbooks for other programming languages and I had no trouble solving them with slight modifications to the syntax.

Tk GUI

Our first practical example is from the world of GUI programming. I want to assign a simple instruction to a button and run the instruction when the button is clicked. Like many other languages, Mercury has a Tcl/Tk interface: Tcl/Tk is extremely popular because of its ease of handling. The list of Tk GUI elements accessible via the Mercury interface is not complete but is easily extensible, assuming you have some basic knowledge of C and Tcl/Tk. You can embed native C code in Mercury by just entering it in the Mercury source code; everything else is handled automatically. Besides offering an elegant approach, Mercury is augmented by the extremely C-friendly Tcl/Tk interface. Also, Mercury and Tcl/Tk have a common string interface; thus, four lines of code is all it takes to add a *cget* for reading widget configurations. (Listing 3).

By systematically working my way through the Tk examples in a popular textbook, I converted them to Mercury. Required extensions were a question of minutes in most cases. However, GUI programming also provides a good approach to demonstrating some of the more advanced aspects of the language (Listing 4).

The example passes in two predicates as arguments. For one, this passes the action *reportColor/4* into a button to be triggered at clicking, and the Tcl/Tk interpreter receives its intended behavior that way, in the form of the *task/3* predicate that is passed in (task/output). If you use Mercury every day, you will soon discover that higher order expressions have more than curiosity value. In fact, they can become a good habit that makes your life much simpler, which does not compare with the clumsy way this kind of case is handled in Java. Note that this example uses namespaces such as *mtcltk.* or *mtk.*, which I have not referred to thus far.

Mercury preserves one of the major benefits of Tk syntax – widgets can be configured simply by chaining key value pairs together. This gives programmers the ability to express GUIs concisely.

Listing 4: GUI Buttons and Triggered Action

```
main(!IO) :-
  mtcltk.main(
    pred(Tk::in, I::di, O::uo) is det :-
      task(Tk, I, O),
      ["Keys"],
      !IO).

:-pred task(tcl_interp, io, io).
:-mode task(in, di, uo) is det.
task(Tk, !IO) :-
  Frame = mtk_core.root_window,
  configure(Tk,
    Frame,
    [height(40),
      width(400), background("green"),
      padx(50), pady(10)],
    !IO),
  newKey(":-O", "yellow",
    Tk, Frame, YellowKey, !IO),
  newKey(":-o", "red",
    Tk, Frame, RedKey, !IO),
  newKey(";-)", "blue",
    Tk, Frame, BlueKey, !IO),
  mtk.pack(Tk,
    [pack(YellowKey, []),
      pack(RedKey, []),
      pack(BlueKey, [])],
    !IO).

:-pred newKey(string, string,
    tcl_interp, widget, widget,
    io, io).
:-mode newKey(in, in, in,
    in(toplevel), out,
    di, uo) is det.
newKey(Label, Color, Tk,
    Frame, Key, !IO) :-
  mtk.button(Tk,
    [text(Label), background(Color),
      active_background(Color),
      padx(50)],
    Frame,
    Key,
    !IO),
  configure(Tk, Key,
    [command(reportColor(Key))], !IO).

:-pred reportColor(widget, tcl_interp,
    io, io).
:-mode reportColor(in(button), in,
    di, uo) is det.
reportColor(Key, Tk, !IO) :-
  cget_string(Tk, Key, "background", HgColor, !IO),
  io.write_string("Clicked: '-background "++HgColor++"'", !IO), nl(!IO).
```

Listing 5: Tk Instantiation

```
:- inst button_config
    --->    ...
    ;    background(ground)
    ;    ...
    ;    command(pred(in, di, uo) is det)
    ;    ...
    ;    text(ground)
    ;    ...
```

insts and modes

The *in*, *out*, *di*, and *uo* modes are just the very beginning of Mercury's powerful mode system. The logical result of Mercury's modes is an additional descriptive system.

First, you will need to understand that a mode can be broken down into two instantiation states and it can even describe the transition between one state and the other.

The simplest examples are *bound* (determined by the context) and *free* (not determined in any way by the context). Thus, *in* reflects the state change *bound* > > *bound* and *out* the state change *free* > > *bound*.

Listing 6: Type Class

```
:-typeclass lowlevel(STREAM)
    where [
  % Stream, Message, OnError, !IO
  pred get_error(STREAM, string,
      bool, io, io),
  pred get_error(in, out,
      out, di, uo) is det
].
```

Listing 7: Type Class + Methods

```
:-type stream --->    ...
:-instance lowlevel(stream) where [
  (get_error(Stream, Message, OnError, !IO) :-
        ...
    )
].
```

Listing 8: Interface

```
:-pred use(S, io, io) <= lowlevel(S).
:-mode useStream(in, di, uo).
useStream(Stream, !IO) :-
  ...
  get_error(S, Message, IsError, !IO),
  (if IsError = yes then
     write_string("Error: "++Message++"\n", !IO)
  else ...
```

The complexity of a Mercury mode can be traced back to its instantiation states, which can be defined with a separate declaration: *:-inst ...*. In Listing 3, GUI elements have their own instantiation states, from which relating modes can be derived with parenthesis operations: *in(toplevel)* or *in(button)*. Additionally, the prolific use of the functional modes and predicates is a defining characteristic of the language.

In Listing 5, the parameters in the Tk Widget configuration list have different individual instantiation states. Once you get used to thinking about this separately, the whole process becomes as easy as a type/class declaration.

Object Oriented

Mercury supports many different styles of programming, making it easily accessible to newcomers. Because the borders between paradigms are not strict, many roads can lead to Rome.

Because object-oriented (OO) programming is very popular, I thought it would be interesting to find out to what extent Mercury will allow OO programmers to stick to their old habits. Many of the details for OO programming in Mercury are still not in place, but Mercury still supports it for the most part. The developers are working on some OO features now, and more details will emerge in the near future.

Listing 9: Extending Type Classes

```
:-typeclass output(STREAM)
    <= lowlevel(STREAM) where [
  % Stream, Char, Success, !IO
  pred write_char(STREAM, char,
    bool, io, io),
  pred write_char(in, in,
    out, di, uo) is det
].
```

Mercury has a CORBA interface, and another type system is designed to allow programmers to easily render existing OO constructs from languages such as Java and C + + in Mercury. Thus, OO programmers can easily migrate their familiar architecture patterns to Mercury.

The notation could be hard to get used to at first because it does without the object name when functions or predicates are called.

A touch less of the syntactical candy might be better for developers with OO roots. The *:-typeclass ...* type classes correspond to Java interfaces for the most part. For example, a low-level stream should support reading the error status (Listing 6).

The implementation is anchored to specific *type*s, which actually represent the equivalent of a method-free class in OO languages. After finding a suitable type, the methods of the current type class interface are added, on the basis of the type, with an *:-instance ...* declaration (Listing 7).

Listing 10: Class-Specific Interfaces

```
:-typeclass ostreamCollection (OSTREAMS) where [
  pred writeCharInAll(OSTREAMS, char, io, io),
  mode writeCharInAll(in, in, di, uo) is det
].
:-instance ostreamSammlung(list(OSTREAM)) <= output(OSTREAM) where [
  writeCharInAll([], _Char, !IO),
  (writeCharInAll([OStream|Xs], Char, !IO) :-
    write_char(OStream, Char, _, !IO),
    writeCharInAll(Xs, Char, !IO)
  )
].
```

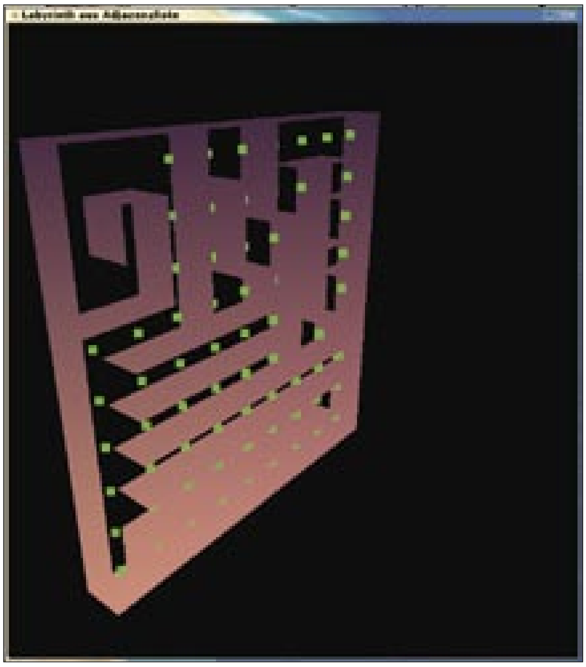

Figure 3: Mercury has an OpenGL library for more demanding GUI applications.

Referencing an interface is a little complex (Listing 8). You can follow similar approaches to extend type classes, and multiple inheritance is supported (see Listing 9). In a similar way, the *instance* declaration supports class-specific use of interfaces (Listing 10).

Recently, the typeclass system has found a further extension allowing instances upon polymorphic types. That being said, the architectural patterns the object-oriented community knows and loves can be implemented in Mercury right now without headaches. The limitations in this regard have become exceptions, rather than the norm.

Library Support

The Mercury compiler distribution already has all the major elements needed to build a compiler: aggregate types such as trees and sets, lexers, parsers, syntax processing, random numbers, benchmarking, error-handling, and so on. For other types of applications, it is advisable to check out the tools distribution, which includes CGI support, an ODBC interface, stream handling, and even sockets. This distribution also includes an XML processing feature, which is quite useful in real-life programming tasks.

Mercury offers several approaches to implementing GUIs. On the one hand, it supports the curses libraries and spartan-style, console-based access. On the other hand are GUI toolkits, such as the lean, Xlib-based Easy X Library or the tried and trusted Tcl/Tk. If this is not enough to keep you happy, you'll find an OpenGL interface for more exacting tastes (Figure 3).

Mercury's answers to Flex and Bison are located in the *lex* and *moose* packages. Libraries can be found for complex numbers and genetic algorithms for scientific calculations; the latest addition is a toolkit for neuronal networks, which is one of the fastest of its kind.

Constraint solving is in a state of flow at the moment, but one can already experiment with constraints or construct own solver types.

Not just part of the library, but definitely worth mentioning for all those who are not fans of opulent GUI systems, is the advanced development environment, which includes a convenient declarative debugger, a random generator unit testing, and a tool to check for test coverage, to name just a few. The collection of these libraries further reinforces the notion that Mercury is intended for doing ambitious projects.

Conclusion

All told, Mercury still needs a fair amount of hacker culture and some pioneering spirit to navigate the various minor bumps. Experienced Linux users who are not afraid of riding an untamed mustang should have no trouble with using Mercury productively. Even newcomers who just want to give Mercury a trial run are guaranteed an exciting afternoon of hacking. ■

Constraint Solving

Constraint solving means that a programming system contains a number of mandatory conditions and autonomously discovers solutions to problems on the basis of the idea that the user simply has to formulate the task. In practical applications, constraint solving has actually proved capable of resolving various planning issues, especially in high-technology areas. Although constraint solving is cited as a prime example of the use of Prolog, the efficiency of Prolog-based constraint solving has often proved unsatisfactory in the past. For this reason, constraint-solving systems have often been implemented in traditional, non-declarative languages to boost performance.

Mercury did not support constraint solving for a long time, but the HAL constraint logic programming system demonstrated considerable performance benefits based on Mercury. The HAL Project has been succeeded by the G12 project [7], a pan-Australian effort through which quite a lot of HAL has found its way into the solver-type system of Mercury. In distinction to other logical programming system, one departs from a rather black box approach to a highly customizable constraint solving system for uncompromised performance.

Again, constraint solving is reflected by an extension to the mode system. An *any* (meaning "not yet specified") state has been added to the *free* and *bound* instantiation states. The way the constraint task is postulated defines which data units are constrained by what.

INFO

[1] Mercury project website: *http://www.mercury.csse.unimelb.edu.au*

[2] Zoltan Somogyi, University of Melbourne, *http://www.cs.mu.oz.au/~zs/*

[3] The Mercury Project: Motivation and Overview: *http://www.mercury.csse.unimelb.edu.au/information/motivation.html*

[4] Jean-Yves Girard, Director of Research at CNRS: *http://en.wikipedia.org/wiki/Jean-Yves_Girard*

[5] Philip Wadler, University of Edinburgh, *http://homepages.inf.ed.ac.uk/wadler*

[6] Ralph Becket, Maria Garcia de la Banda, Kim Marriott, Zoltan Somogyi, Peter J. Stuckey, and Mark Wallace, "Adding constraint solving to Mercury," Proceedings of the Eighth International Symposium on Practical Aspects of Declarative languages, Charleston, South Carolina, January 2006, page 16ff. The paper is also available at *http://www.mercury.csse.unimelb.edu.au/information/papers.html#padl06solver*.

[7] G12 project: *http://www.g12.cs.mu.oz.au/*