Re-Implementing a Machine Learning Program in Mercury

Gregory D. Weber

Computer Science Department
Indiana University East
Richmond, IN 47374-1289
gdweber@indiana.edu

Abstract
This paper describes the author’s experience of using the programming language Mercury to re-implement the machine learning program CN2. Mercury is a purely declarative language, closely related to Prolog, and designed for speed.

Motivation
During the course of testing the machine learning program ICN (Weber 2003), it became clear that the program would need a significant speed-up in order to do serious work with large-scale data sets. ICN was able to train and test on all 581,012 examples of the forest cover data set (Bay 1999) in 4.4 hours. However, the experiments required running the program with a variety of parameter settings and repeating each run several times to smooth out random fluctuations. Some experiments ran for over 80 hours.

ICN is an incremental variant of the rule learning program CN2 (Clark & Niblett 1989; Clark & Boswell 1991). It is coded in Scheme and run by the Petite Chez Scheme interpreter. Since the “official” C language version of CN2 runs 10 times faster than the Scheme version, it is likely that a C language implementation of ICN would also run 10 times faster.

I was reluctant to get involved with such a low-level programming language as C. Indeed, it seemed desirable to work with an even higher-level language than Scheme, using logic programming in order to extend the program later with a first-order predicate calculus representation of data.

Mercury (Mercury Project 2004; Becket 2002; Henderson et al. 2002) is a purely declarative logical/functional programming language designed for high-speed computation. Closely related to Prolog, it would provide the desired support for logic-based data representation. To evaluate Mercury, I undertook two pilot projects: (1) developing a simple rule learning program in Mercury; (2) converting the Scheme version of CN2 to Mercury. This paper reports on the experiences and results of the second pilot project.

Listing 1: Non-declarative aspects of Prolog.

<table>
<thead>
<tr>
<th>alpha(X) :- bravo(X).</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha(X) :-</td>
</tr>
<tr>
<td>charlie(X),</td>
</tr>
<tr>
<td>delta(X).</td>
</tr>
<tr>
<td>alpha(X) :-</td>
</tr>
<tr>
<td>print('What is alpha?'),</td>
</tr>
<tr>
<td>read(X),</td>
</tr>
<tr>
<td>print('alpha is '),</td>
</tr>
<tr>
<td>print(X),</td>
</tr>
<tr>
<td>nl.</td>
</tr>
</tbody>
</table>

Language Features
Mercury is a language that blends the capabilities of logical and functional programming languages. Mercury procedures, like those of Prolog, are expressed in rules using a subset of predicate logic. As in Scheme and Common Lisp, procedures are first-class objects: a procedure can be the value of a variable, it can be an argument of a procedure, and it can be the value returned by a procedure. This makes it possible to develop “higher order” procedures; for example, the map procedure, like mapcar in Lisp, applies a given procedure to each element of a list and collects the list of results.

Mercury differs from Prolog in two important respects. First, Mercury is purely declarative. Second, Mercury requires type and mode declarations, which enable the compiler to detect errors early and to generate efficient code.

Declarative Programming
Prolog is not a purely declarative language, although it is far more declarative than commonly used languages like C and Java (Clocksin & Mellish 1987). Listing 1 shows three ways in which Prolog is procedural rather than declarative.

1. The order of the rules defining a predicate is significant. Given the goal alpha(jack), the Prolog inter-
preorder will try to apply the first rule first. Only if the first rule fails will it try rule 2, and only if that fails, rule 3.

2. Within the body of a rule, the order of goals (conditions) is significant. If the interpreter is applying rule 2, it will first try to achieve the goal charlie(jack), then the goal delta(jack). It will not attempt the goals in the reverse order.

3. Input and output are procedural operations because they change the state of a program. Reading the variable X consumes input characters, putting the program into a new I/O state. Similarly, each print or nl (newline) operation emits output characters, resulting in a new state. All I/O operations change state, just as surely as assigning a new value to a variable changes the program’s state.

In Mercury, the case is different. The Mercury compiler has great freedom to optimize code by executing operations “out of order.” For example, it may compile the rules in such a way that the program tries to apply the second rule first, or within the second rule, to achieve the second subgoal first.

The third rule is not valid in Mercury because of its non-declarative I/O. How Mercury deals with I/O will be explained in the section “Declarative I/O”.

Types and Modes

Mercury allows both predicate and function definitions. Both require type declarations.

Listing 2 shows the definition of a predicate that inserts a condition into a prule (probabilistic rule). The rule for the predicate is preceded by two declarations. The pred declaration names the predicate and lists the types of its 5 arguments. The mode declaration specifies the way in which each argument may be used: an in argument must be ground (fully instantiated) when the predicate is called; an out argument must be free at call time and is ground when the predicate exits. Thus in means “ground to ground,” and out means “free to ground.” The mode statement also declares that the predicate is deterministic (det), meaning that it succeeds exactly once for each call. A predicate that may either succeed once or fail is semidet; one that may succeed one or more times or fail is nondet.

The pred and mode declarations may be combined into one statement, as in Listing 3.

Mercury’s standard libraries provide the types int, float, bool, string, and list. The language provides a rich and flexible mechanism for programmer-defined types. These include abstract types, equivalence types, and discriminated union types.

Listing 4 illustrates the three sorts of programmer-defined type. The first statement declares constraint_type as an abstract type. In other words, it just says that constraint is the name of a type, without saying what type it is. An abstract type may be declared in the interface of a module; the details are hidden in the module’s implementation.

The next statement declares an equivalence type: varref is introduced as a synonym for the int type.

The most interesting kind of programmer-defined type is the discriminated union. At its simplest, a discriminated union is a set of constants. The third statement declares constraint_type as a discriminated union with six allowed values (eq, ne, etc.). It can get more complex. The next statement declares avalue (attribute value) as another discriminated union type. Examples of valid values of this type include s(" jill "), i(35), f(3.5), and unknown. This sort of discriminated union provides a tagged data structure, similar to the “union” construction in C. A discriminated union does not have to have more than one variant. The cond type, in the next example, is a simple tagged structure with two elements.

Mercury has parametric types, similar to the template types in C++. For example, the parametric types list and array are parameterized to specify their element types. The next two declarations say that conds is a list of cond, and object is an array of avalue. Programmers can also define their own parametric types.
Listing 4: Type declarations.

```prolog
:- type constraint.

:- type varref == int.

:- type constraint_type --> eq; ne; lt; le; ge; gt.

:- type avalue --> s(string); i(int); f(float); unknown.

:- type cond --> cond(varref, cpred).

:- type conds == list(cond).

:- type object == array(avalue).
```

Listing 5: Conditional structures in Prolog (templates): three ways to state that the goal $G$ is achieved by “if $C$ then $D$ else $E$”.

```prolog
% (i)  
G :- C, !, D.
G :- E.

% (ii)  
G :- C, D.
G :- not C, D.

% (iii)  
G :- (C -> D ; E).
```

Conditional Idioms

There are various ways to code the “if $C$ then $D$ else $E$” structure in Prolog (Listing 5). Of these, (i) Mercury does not support the cut operator (“!”), used in Prolog to commit to a particular solution and prevent backtracking. Method (ii) may be inefficient if evaluating $C$ is costly. Both (ii) and (iii) work in Mercury, but (iii) is preferred.

Another conditional structure in Mercury, the structural switch, is shown in Listing 6. The first argument of write_conditions, Conds, is a $cons$, which is a list of $cond$. Since it is a list, it is either the empty list, $[]$; or a list with head $CH$ and tail $CT$, written $[CH | CT]$. The switch operator $;$ expresses these as mutually exclusive alternatives, for which the compiler can generate very efficient code.

Declarative I/O

How in the world does one do input and output declaratively, since I/O by definition involves changing the state of the program? The answer will be familiar to those who have studied the situation calculus used in AI planning: an I/O operation is like a planning operator which transforms one state into another. That is, the I/O state of the program is represented explicitly by variables, and a call to any I/O procedure requires two such variables, “before” and “after”.

As Listing 6 shows, I/O states are “threaded” between the I/O procedures, so that the “after” state of one operation becomes the “before” state of the next operation. In the predicate write_conditions, IO0 is the initial I/O state, and IO is the final I/O state. These states have type $io$. They also have special modes. Since backtracking across I/O is impossible, the mode of “before” states is destructive input ($di$); the mode of “after” states is unique output ($uo$). That is, an I/O state is destroyed after it is “input” to a procedure, and the I/O state that is “output” from the procedure is unique.

In this pure form, declarative I/O is tedious and error-prone. The latest release of Mercury includes a better alternative, called state variables. A state variable !X stands for a pair of $io$ variables, representing a change in I/O state. State variables are syntactic sugar; the compiler takes care of the details by making up new variable pairs as needed. Listing 7 shows the write_conditions predicate using state variables.

Higher-Order Programming

In higher-order programming, one has functions that have other functions as arguments and functions that return functions as values. Mercury supports higher-order programming for both predicates and functions.

An interesting complication is that the modes of these variables are no longer simply in (ground to ground) or out (free to ground); because a predicate has parameters of its own, it is never ground until these parameters...
Listing 7: I/O states with “state variables.”

\[
\text{write CONDITIONS}(\text{Prefix}, \text{Conds}, \text{!IO}) :- \begin{cases}
\text{Conds} = [] ; \\
\text{Conds} = [\text{CH} | \text{CT}], \\
\text{print}(\text{Prefix}, \text{!IO}), \\
\text{write CONDITIONS}(\text{CH}, \text{!IO}), \\
\text{nl}(\text{!IO}), \\
\text{write CONDITIONS}(\text{AND}, \text{CT}, \text{!IO}) \\
\end{cases}
\]

are instantiated by a call. Thus for a \textbf{det} predicate \(P\) having two parameters, one \textbf{in} and one \textbf{out}, if \(P\) is an input parameter to another predicate, its mode is described as \textbf{in}(\text{pred(in, out)} \textbf{is} \text{det}); if \(P\) is an output parameter of another predicate, its mode is described as \textbf{out}(\text{pred(in, out)} \textbf{is} \text{det}).\footnote{In general, for any instantiation \(I\), the notation \text{in}(I)\) means “\(I\) to \(I\)”; and \text{out}(I)\) means “free to \(I\).”}

Examples of higher-order programming are shown in Listing 8.

1. The predicate \textit{accum} accumulates a 3-ary predicate over a list. For example, the call

\[
\text{accum}(\text{sum}, 0, \text{L}, \text{Total})
\]

would compute the sum of the elements of list \(L\).

2. The predicate \textit{curry_add} takes a number \(N\) and returns a predicate \(\text{Pred}\) which adds \(N\) to its argument. For example, the following expressions create an “add 5” predicate and call it to find \(X = 15\) and \(Y = 25\):

\[
\text{curry_add}(5, \text{Add5}), \\
\text{call(Add5, 10, X)}, \\
\text{Add5(20, Y)}
\]

The word \textit{call} is optional in calling higher-order predicates, as the last expression illustrates.

Reimplementing CN2 in Mercury

Design

With two existing implementations of CN2 in C and Scheme, most of the design decisions had already been made, and the plan was for the new, Mercury implementation to follow the same design as the others in most respects. However, one motivation for programming in Mercury was to change to a first-order logic representation of data and rules, and the temptation to do so right away was just too great. Given the desire to represent examples as Prolog-style facts and to use unification for matching rules to examples, the question was how to represent logic terms.

The rule learning program would start with a description of a classification task, which includes a specification of the predicates to be used, their names, arity, and argument types, and a collection of training examples. Rules are learned by general to specific search (as in the CN2 algorithm): to learn a rule, the program starts with the most general term for some predicate and incrementally adds conditions to it. For example, given the knowledge that there is a widget predicate with 2 arguments, the first argument’s values may be yellow, red, or black, and the second argument’s values are numbers, the program could begin with the general term

\[
\text{widget}(A, B).
\]

and form specializations such as

\[
\text{widget(yellow, B)}. \\
\text{widget(A, B), B > 10, B \leq 16.}
\]

The \textit{term} library and its associated libraries (\textit{term_jo}, \textit{varset}, \textit{lexer}, \textit{parser}) provide predicates for building, unifying, and otherwise manipulating terms. These libraries are used in the Mercury compiler and are said to be fairly efficient. I decided to use them.

Implementation

Debugging

Naturally, the program had to be debugged. Mercury has a sophisticated interactive debugger (Figure 1). Like many Prolog debuggers, it abstracts the execution of a program into a sequence of trace events, including CALL, EXIT, REDO, and FAIL. The debugger’s commands include forward movement (step, continue), backward movement (retry a goal), setting breakpoints, browsing the call stack, and browsing terms. When the debugger runs with the GNU Emacs interface, the lines of source code being executed are shown in another window.

Although the debugger is said to be highly customizable, I had difficulty getting it to trace events without showing too much or too little detail. Probably, greater familiarity with the debugger would have enabled me to solve these problems.

I/O Problems

In a declarative language, a predicate can perform I/O only if it has access through its parameters to the current I/O states. The predicate \textit{main} has two I/O state parameters, representing the states at the beginning and end of the program, and it can pass them to any predicates it calls. If one wants I/O from a predicate which is nested, say, ten calls deep under \textit{main}, then it is necessary to pass the I/O parameters through all ten levels of calls to the desired predicate. Inserting the chains of I/O parameters was a complicated chore.

Mode Problems

The most difficult conceptual hurdle for me was understanding how types and modes work for higher-order predicates and functions. Since
Listing 8: Higher-order programming.

\begin{verbatim}
:- pred sum(int, int, int).
:- mode sum(in, in, out) is det.

sum(I, J, Sum) :- Sum = I + J.

:- pred accum(pred(int, int, int), int, list(int), int).
:- mode accum(in(pred(int, in, out) is det), in, in, out) is det.

accum(_, Ident, [], Ident).
accum(Pred, Ident, [H | T], AccumAll) :-
    accum(Pred, Ident, T, AccumT),
    call(Pred, AccumT, H, AccumAll).

:- pred curry_add(int, pred(int, int)).
:- mode curry_add(in, out(pred(int, out) is det)) is det.

curry_add(N, Pred) :-
    Pred = (pred(I::in, O::out) is det :- O = I + N).
\end{verbatim}

Figure 1: Mercury Debugger, stopped at a breakpoint for \texttt{find\_best\_rule}, with stack trace.
Implementation | Soybean (sec) | Forest (min)
--- | --- | ---
C | 1 | 12
Scheme | 22 | 129
Mpre | 648 | —
Mpost | 3 | 36

Table 1: Running times (user CPU times) of CN2 on the soybean and forest cover data sets. Mpre and Mpost are Mercury before and after code tuning.

the program used data structures that contained predicates as elements, the type and mode declarations were horrendously complex. Subsequently I learned that much of this could be simplified, by using functions rather than predicates, if a function type has mode \( \text{func}(\text{in}, \ldots, \text{in}) = \text{out is det} \) (Becket 2004). For example, the declaration

\[
:- \text{mode} \ \text{foo} (\text{in} (\text{func}(\text{in}) = \text{out is det}), \text{out} (\text{func}(\text{in}) = \text{out is det})) \text{ is det}.
\]

can be simplified as

\[
:- \text{mode} \ \text{foo}(\text{in}, \text{out}) \text{ is det}.
\]

This simplification does not apply to predicates.

Performance The first timing tests using the soybean data set (Blake & Merz 1998) were very disappointing. I had hoped that Mercury would be significantly faster than Scheme. Instead, the Mercury CN2 program was 30 times slower than Scheme, and 645 times slower than C!

Refinement Running the Mercury profiler showed that the program spent 65% of its time in unification to determine if a rule matched an example. Using the term library had made development easier but imposed an intolerable penalty on runtime performance.

I decided to throw out the term library and redesigned the representation of data and rules, reverting to the array-based data representation and special-purpose match procedure used in the Scheme program. This change produced an immediate speed-up of 7x.

I found that, overconfident in the speed of the language, I had neglected to implement in Mercury some of the code optimizations used in the Scheme program. Eliminating redundant tests of old conditions in new rules during rule specialization doubled the speed. Other enhancements included eliminating duplicate rules and compiling each rule’s conditions into a high-speed match procedure (an application of higher-order programming). The net result of all of these code optimizations was a Mercury program which ran in 3 seconds, 8 times faster than Scheme, and 245 times faster than the initial Mercury version (Table 1, Figure 2).

Additional tests using the larger forest data set (Bay 1999) were less favorable for Mercury (Table 1, Figure 3). This time, Mercury’s speed was only 4 times faster than Scheme. It is possible that the forest data creates different “hot spots” in the program than soybean, which might be identified by further profiling, suggesting new opportunities for speedup.

On both data sets, the C program is about 3 times faster than Mercury.

Lessons Learned
1. I would probably have had less trouble if I had invested more time in learning to use the Mercury debugger effectively.
2. The Mercury profiler enabled me to identify opportunities for speedup.
3. The Mercury user community is wonderfully helpful and friendly. I should have asked for advice earlier and more often on the user mailing list.
4. The term library and its unification procedure were handy for initial development, but ultimately unsatisfactory; a special match procedure was required for efficiency.

Evaluation and Future Work
Compared to Scheme, the major benefit of Mercury is a speedup of 4x to 8x depending on the data in the learning problem. It remains to be seen whether Mercury, as
a logic-based programming language, will make it easy to extend the existing program to relational learning.

There are some disadvantages. Mercury has neither a read-eval-print loop nor an interpreter, prohibiting the highly interactive development process characteristic of Lisp and Scheme programming. Type declarations are an additional burden on the programmer, although they do allow the compiler to catch errors which in Scheme would have become apparent only at run time. The greatest difficulty in this project was for the programmer to learn to think in terms of Mercury’s mode system, especially in the case of higher-order programming. It was in fact the greatest conceptual challenge this programmer has ever faced, harder even than learning the first programming language! It is likely that the difficulty of using the mode system will diminish with greater experience.

While Mercury does not provide as much speedup as C, its advantages compared to that language include garbage collection, high-level syntax based on predicate logic, and full support for higher-order programming.

Code size is sometimes taken as an indicator of coding effort. As expected, Mercury had the least number of lines of code, and C had the greatest (Figure 4). But the differences are surprisingly small; the C program is only 14% larger than the Mercury program. This comparison is somewhat rough, because the three programs did not implement exactly the same features.

Although the initial encounter with Mercury was extremely frustrating in some respects, it has stretched my mind, and I am eager for more. Future plans include re-implementing the incremental learning program ICN (Weber 2003) in Mercury and extending it to relational and regression learning.

Acknowledgments Thanks to Ralph Becket, for helpful suggestions concerning the design and implementation of the Mercury version of CN2; to Ian MacLarty, Peter Hawkins, Ralph Becket, and an anonymous reader, for comments on earlier drafts of this paper.

References
Becket, R. 2004. Personal communication.

