PROGRAMMING LANGUAGE CONCEPTS AND PERL

Chuck Liang
Department of Computer Science
Hofstra University
Email: cscccl@hofstra.edu

ABSTRACT

Perl is sometimes overlooked in high-level courses in programming languages. Recent versions of Perl come equipped with an array of features that can in fact be helpful in demonstrating programming language principles and design choices. Higher-order, functional style programming is fully supported by Perl. As a language designed for purely practical as opposed to theoretical purposes, it offers a contrast as well as a companion to languages such as Scheme and ML, which are traditionally used in programming language courses.

INTRODUCTION

Like many instructors, the author has taught undergraduate courses that attempt to introduce students to high-level programming language concepts and alternative programming paradigms. Students have a tendency to think of programs only at the level of their syntax and behavior. Most students are comfortable with conventional, procedural programming but find other paradigms such as functional and logic programming intimidating and awkward. Increased computational power and the advent of modern languages such as Java has narrowed the gap between conventional practice and higher-level approaches to programming. For example, the availability of language runtimes that perform automatic garbage collection has made it possible to teach programming in a more abstract and declarative manner. The latest popular languages such as Java exhibit a sometimes-surprising set of abilities that were once thought to be of only theoretical importance. However, it remains often the case that by the time students take a course on programming languages, they are still only familiar with one particular style of programming. In such courses, functional languages such as Scheme and ML are often used to help students think at a higher level about the nature of programming and programming language design.

The purpose of this paper is to show how the popular language Perl can be used to teach and demonstrate advanced programming language concepts that are usually taught in Scheme and ML. The author does not argue that these functional languages should be replaced by Perl entirely. However, the author does believe that as an important and practical language, Perl should not be ignored in a general programming languages course and can in fact offer exciting examples of alternative approaches to programming. How one actually structures a course may depend on the background of students at a particular institution. Scheme is sometimes taught to undergraduates at an early stage, in which case

Perl can be used to complement the students' existing knowledge, i.e., to *"provide an alternative realization of the same concepts".* It is also often the case that students lack exposure to any form of functional programming, in which case the flexibility of Perl can help to ease the adjustment to a radically different paradigm.

In this paper, a minimal level of familiarity with Perl is assumed, including the meaning of the symbols `$` (scalar), @ (sequence), sub (lambda), my (let) and `->` (dereference). Scalar variables are universal in that they can represent primitive values as well as references to arbitrary structures and functions. A basic familiarity with functional programming and the lambda calculus is also assumed.

Why Perl?

An important principle of Perl, expressed by its designers, is that "*there's more than one way to do it.*" Although Perl began humbly as a scripting language, it is a credit to Larry Wall and other Perl designers that they are well-aware of advanced programming language principles. The recent features incorporated into Perl (especially since version 5) represent an excellent example of the merging of theory and practice. In contrast, Scheme and ML, despite possessing novel capabilities, never in fact achieved wide-spread acceptance as practical languages. To be fair, part of the appeal of Perl lies in that it can be used as a structurally mundane, imperative language. Indeed many Perl programs can be found that use only global variables and include no subroutines. For this reason it is generally considered inappropriate to use Perl in introductory level courses. However, the designers of Perl also recognize the vast range of possibilities in programming and have endowed the language with a surprising host of characteristics one would not have expected from any imperative language. Perl is a full-featured, *higher-order* programming language. It implements the principle of *"functions as first class citizens"* in that functions (subroutines) can be created dynamically and passed as parameters to other functions. Furthermore, a function referring to nonlocal variables form a *closure* consisting of the function body and a persistent local environment with bindings for these variables. Automatic garbage collection frees programmers from having to deal with memory management and pointers. Perl also offers flexibilities, such as both lexically and dynamically scoped variables, which are not found in most other languages but are useful for demonstrating the impact of key choices in programming language design. Being an untyped language, Perl naturally cannot be used to replace strongly typed languages such as Ada, Java and ML (in its role as a typed language). However, there are many advantages in using an untyped language to provide a contrast to, and to motivate, the usage of types.

THE LAMBDA CALCULUS

The Lambda calculus is often used to introduce students to the foundations of programming language design, and to discuss in a formal context issues such as normal versus applicative orders of evaluation. The Perl "subroutine" constructor `sub` is flexible enough to be called "lambda" in that it can be nameless and appear anywhere values can appear.

The syntax of Perl sometimes appear cryptic to those new to the language. Recent developments, however, have greatly alleviated this problem. Since version 5, Perl has allowed the use of '`->`' to dereference pointers to functions and apply them to arguments, thereby simulating Curried application. That is, `a->b->c` should be thought of as ((a b) c). Perl version 6 [2,4] introduces many further refinements, some of which are already available as special packages for Perl 5. They can be downloaded from http://search.cpan.org. The code in this paper will sometimes make use of the *Perl6::Currying* package[1]. With this package, subroutine parameters can be named, as in `sub($x)`.

The following subroutines implement the foundational I, K and S combinators of the (untyped) lambda calculus:

```
use Perl6::Currying;

sub I ($x) { $x }

sub K ($x) { sub ($y) {$x} }

sub S ($x) { sub ($y) { sub ($z)
       { $x->($z)->($y->($z)) } } }
```

Using named variables in this fashion also forces the call-by-value style of parameter passing. Without the new Perl6 style syntax, these lambda terms can still be written, but in a less natural form. The K combinator, for example, would have to written as:

```
sub K
{  my $x = $_[0];
   sub { $x }
}
```

Other than the use of named parameters, the syntax of code in this paper is consistent with Perl 5.80 and has been tested on several platforms.

A certain student once commented that the rules of beta reduction apply only to languages such as Scheme. The formulation of these combinators in Perl demonstrates otherwise. For example, one can show that the combination *(S K I)*, written in Perl as `S(\&K)->(\&I)` and which beta-reduces to I, does indeed have the expected behavior when used as a Perl function.

Call by Value, Call by Name

Developing the basic constructs of a programming language in the lambda calculus is an effective way to discuss fundamental choices in language design. Like most modern languages Perl uses the call-by-value style of evaluation (except for individual (nameless) variables, which are passed by reference). The following code formulates booleans as choice routines (uncurried for convenience), and demonstrate the consequences of

1 This package also provides a limited form of automatic Currying in the "prebind" directive. Given

```
sub f($x,$y) {$x-$y},
```

`&f.prebind(y=>4)` will return a function that expects the remaining parameter and subtract 4 from it.

implementing the if-else construct as a call-by-value routine:

```
sub T { $_[0]; }    # true
sub F { $_[1]; }    # false
sub IFELSE0 ($bool,$A,$B) { $bool->($A,$B) }
```

However, assuming `$x` holds the value 3, calling `IFELSE0(\&T, 1/($x-3))` will lead to a division by zero. Fortunately, the syntax of Perl allows for an efficient simulation of normal-order evaluation. One can wrap the parameters inside subroutines. Like most functional languages, Perl implements *weak beta-reduction*, whereby the bodies of lambda terms are not evaluated until applied.

```
sub IFELSE { IFELSE0(@_)->(dummy) }
IFELSE(\&T, sub{"ok"}, sub{1/($x-3)})  # returns "ok"
```

The Perl syntax clarifies that in order to effect normal-order reduction, one must be able to pass pointers to *code* as opposed to precomputed values.

Similarly, the *applicative-order* fixed-point combinator for recursion, namely

$$\lambda m.(\lambda x.(m\ \lambda y.(x\ x\ y)))\ (\lambda x.(m\ \lambda y.(x\ x\ y)))$$

can be encoded in Perl as follows:

```
sub Y ($M)
{
   my $N = sub ($x) { $M->(sub ($y) {$x->($x)->($y)}) };
   $N->($N);
}
```

One can then define the factorial function as:

```
my $fact =  Y sub ($f) { sub ($n)
                { if ($n<2) {1} else {$n * $f->($n-1)} } };
```

Data Structures and Encapsulation

The encoding of pairs and other data structures inside lambda terms is useful for formalizing the principle of data encapsulation. It is similar to the notion of object-style encapsulation by separating data from outside access through an interface or *dispatch* function. The following code formulates Lisp-style pairs:

```
sub cons ($head,$tail)
{
    sub { IFELSE($_[0], sub{$head}, sub{$tail}); }
}
sub car { $_[0]->(\&T); }
sub cdr { $_[0]->(\&F); }

my $list = cons(2, cons(4, cons(6, cons(8, nil))));
```

```
      my $second = car (cdr $list);
```

Infinite streams can likewise be encoded using constructs similar to those described above.

Higher-Order and Declarative Programming

Given the above formulations, one can naturally expect Perl to support higher-order programming, such as mapping a function over a sequence or array:

```
sub mapfun
  {   my(@rlist);  # list to be returned
      my($fun,@list) = @_;
      for ($i=0;$i<=$#list;$i++)
      {
        $rlist[$i] = $fun->($list[$i]);
      }   # for loop
      @rlist;
  }
```

With Perl, such higher-order features can be combined with conventional language constructs such as the for-loop above. What is the advantage of such a combination? When students study a functional language such as Scheme for the first time, they often see it as having nothing in common with the conventional languages they are used to. The point that many of the ideas one find in Scheme are universally relevant is often lost on them. They have a tendency to see Scheme as a completely alien language where, for example, "*everything must use recursion.*" With Perl, one can demonstrate more convincingly that advanced capabilities such as *mapfun* can be used to enhance any programming language. The above routine also demonstrates the value of automatic memory management. That is, the decision and task to deallocate the original array is made by the garbage collector, freeing the programmer to concentrate on higher-level aspects of program design and implementation.

STATIC AND DYNAMIC SCOPING

It is programming language folklore that John McCarthy misunderstood the lambda calculus and formulated early versions of Lisp to use dynamically scoped variables. The distinction between lexical (static) or dynamic scoping is central to language design. Perl is unique among modern languages in that it offers both. The construct `my` can be thought of as equivalent to *let* in Scheme and ML, while the construct `local` has an effect that is equivalent to dynamic scoping. The following segments demonstrate the distinct consequences of static and dynamic scoping, a contrast that cannot be made directly with most other languages.

```
      # Static (lexical) scoping:
      {
         my $x = 2;
         my $f = sub { $x };
         {  my $x = 3;
            print $f->(), "\n";  # prints 2
```

```
    }
}

# Dynamic scoping:
{
   local $x = 2;
   local $f = sub { $x };
   {  local $x = 3;
      print $f->(), "\n";  # prints 3
   }
}
```

Perl uses a special, alternate stack for implementing dynamic scoping. The old value of a global (non-lexical) variable is saved on the special stack until the closing brace of the `local` declaration. One advantage that Perl brings to a discussion on such language design choices is that, as a well-used language, it provides clear, practical explanations for the consequences of such choices. Perl manuals (such as [6]) encourage the use of lexical variables because of their safety and efficiency, but also explain why and when dynamic variables should be used. They are useful for temporarily replacing the values of built-in global variables such as `@ARGV`, the argument vector of a program.

CLOSURES AND OBJECTS

Readers familiar with Abelson and Sussman's classic text [1] should recognize the following use of closures to model "bank account" objects. The Perl code given above do not make destructive changes to state variables, and can therefore be considered purely functional. By combining the notions of encapsulating data inside lambda terms and the idea that functions are paired with mutable environments, a form of object-orientation can be achieved. The `$deposit` and `$withdraw` routines are local (thus *private*), and because they modify the environment variable `$balance`, are more properly considered instance methods as opposed to functions. The principal difference between the traditional Scheme formulation and what follows is the use of a Perl hash table in place of a dispatch function.

```
sub makeaccount
{  my $balance = $_[0];
   my $deposit =
      sub { $balance += $_[0]; $balance; };
   my $withdraw =
      sub { if ($_[0]<=$balance)
                { $balance-=$_[0]; $balance; }
           else { print "insufficient funds\n"; }
         };
   my $dispatch =
   {
     withdraw => $withdraw,
     deposit => $deposit
   };
   $dispatch;
}
```

```
$account0 = makeaccount(100);
$account1 = makeaccount(200);
print $account0->{deposit}->(50);
print $account1->{withdraw}->(100);
```

An interesting variation of the above program is to repalce the uses of`my` with `local`. The same `$balance` variable will then be modified by the deposit and withdraw routines for *both* bank accounts. Such emphatic demonstrations are instrumental in getting the point across to students, who are often tied to their preconceptions and habits and thus reluctant to accept new concepts when presented purely in the abstract.

Late versions of Perl offer built-in support for another form of object orientation that is also interesting to consider. There are certainly advantages in showing students how to build capabilities from scratch as opposed to taking features such as classes and inheritance for granted. For example, it is possible to motivate the capabilities of modern object-oriented languages such as Java by trying to emulate them in (traditional) C. To simulated the notion of subclasses and dynamic dispatch, one would have to use unions and encode type tags explicitly inside structs, then construct dispatch (or proxy) functions that check the tag explicitly so as to invoke the correct operation. Perl automates these capabilities in a relatively simple manner. It uses packages to model the concept of a class. A special operation `bless` tags a structure with the name of the package. In this way, Perl structures can carry runtime type information, which is required for automatic dispatch[2]. That is, if one "blesses" a structure, such as a hash table, with the name of a package:

```
bless $mystruct, 'mypackage';
```

then subsequent calls to `$mystruct->routine()`will in fact be equivalent to

```
mypackage::routine($mystruct);
```

An `ISA` predicate associates packages with inherited packages. With closures and package-based classes, Perl offers an interesting contrast to the conventional approach to object-orientation exemplified by C++ and Java. Both forms of encapsulation are included in Perl for practical reasons (see [5]). "Blessed" structures are the default Perl means for a moderate form of object-orientation. Closures, which are more expensive, are useful as event-handling routines in graphical user interfaces.

Meta-Programming, Reflection, and *eval*

The flexibilities of the Perl language can be useful for more than just processing email addresses. Perl supports the `eval` operation (similar to that of Scheme) which allows any expression, including strings, to be treated as a piece of Perl code and executed. The indifference to types may not be conducive to disciplined software engineering, but it gives rise to certain possibilities that can challenge even Scheme's *eval* operator. This flexibility, combined with Perl's well-known facilities for regular expression matching and substitution, can be exploited to enhance Perl itself. ML is well-known for its ability to

2 The package and bless method, however, does not protect data from random access the way closures do.

define functions by cases, as in:

```
fun factorial 1 = 1
    | factorial n = n*factorial(n-1);
```

We can attempt to effect this form of programming by using regular expressions to match cases, and storing the various cases of the function definition inside a hash table that contain subroutines as entries:

```
$fact->{1} = sub{1};
$fact->{'\d*'} = sub{ my $n=$_[0]; $n * app($fact,$n-1) };
```

The regular expression in the second case matches any integral expression. The definition of `app` (application) below uses the `patfind` procedure to look up the hash table for an matching pattern and invokes the associated subroutine:

```
sub patfind
{   my ($pat,%hash) = @_;
    foreach $ent (keys %hash)
      { if ($pat =~ /$ent/)
        { return $hash{$ent}; }
      }
    die "no pattern matches";
}

sub app
{   my ($f,@args) = @_;   # get hastable and args
    my $hkey = join ",",@args;
    my $func = patfind($hkey,%$f);  #need to join args
    $func->(@args);
}
```

Multiple arguments to a function are considered as one hash key using Perl's `join` operation. An invocation such as `app($fact,6)` will then apply the intended function.

It is possible to carry this concept a step further. We can imagine a manner of function definition that allows us to write, for example, the Fibonacci function as:

```
$fib->{0} = 0;
$fib->{1} = 1;
$fib->{'$n:$n>1'} = '$fib->($n-1) + $fib->($n-2)';
```

Here, the hash table contains either exact patterns to be matched, or a variable followed by a boolean condition. The entries of the table are unevaluated bodies of functions. Using `eval` and operations such as `split` and `join`, it is possible to transform this procedure into a proper Perl subroutine that has the intended behavior. That is, it is possible to generate a Perl subroutine dynamically that will match the input with an appropriate table entry, bind the named variables (such as `$n`) to its parameters (`$_[0]`), check the boolean condition, and *eval* the associated body. Due to the length of this procedure (`mkfun`), it is included in the Appendix of this paper. With this routine, we can define the Fibonacci function as follows:

```
$fib = mkfun($fib);
print "fib(8) is ", $fib->(8), "\n";
```

Functions with multiple parameters are also possible using the `join` operation:

```
# Euclid's algorithm:
$gcd->{'0, $m:$m>0'} = '$m';
$gcd->{'$n:$n>0, $m:1'} = '$gcd->($m % $n, $n)';
$gcd = mkfun($gcd);
```

Such possibilities with Perl also makes for interesting student projects that involve both research and implementation.

HIGHER-ORDER SUPPORT IN OTHER LANGUAGES

Among languages outside of the functional variety, one can usually find some level of support for higher-order programming. For example, C allows pointers to functions to be used as values. However, it is difficult to dynamically create functions, and there is no notion of closure. The Java programming language allows instances of interfaces to be created dynamically, which in some sense allows functions to be manipulated as values. It is in fact possible to fashion a representation of lambda terms and beta-reduction inside Java. One can represent lambda terms as classes that implement a method *app*, which defines the body of the term (i.e., what should be returned when the term is applied). The following classes represent the I, K and S combinators:

```
public interface lambda
{  public Object app(Object x); }

class I implements lambda
{
    public Object app(Object x) { return x; }
}

class K implements lambda
{
    public Object app(Object x)
    {
      final Object xf = x;
      return new lambda()
          {
            public Object app(Object y) { return xf; }
          };
    }
}

class S implements lambda
{
    public Object app(Object x)
    {
      final Object xf = x;
```

```
            lambda A = new lambda()
                 {
                   public Object app(Object y)
                   {
                        final Object yf = y;
                        lambda B = new lambda()
                          {
                               public Object app(Object z)
                               {
                                 lambda x = (lambda)xf;
                                 lambda y = (lambda)yf;
                                 return
                                   ((lambda)(x.app(z))).app(y.app(z));
                               }
                          }; // B
                        return B;
                   }
                 };
            return A;
          }
       }  // S
```

One can then define specific lambda terms as instances of the classes:

```
       lambda s = new S();
       lambda k = new K();
       lambda i = new I();
       lambda ski = (lambda)((lambda)s.app(k)).app(i);
```

Compositions of such lambda "objects" do indeed respect the semantics of beta-reduction. Unlike Perl, however, free variables that appear in inlined Java routines must be "finalized." Closures are not created in the sense of Perl or Scheme. In addition, the rigid syntax and type system of Java renders the formulation somewhat awkward: the natural relationship between lambda terms and program elements is not apparent. The Java classes above may be useful in illustrating the expressiveness of the Java language, but they do not clarify the fundamental role of the lambda calculus in language design.

CONCLUSION

It is interesting that a language designed with purely practical motivations have so many advanced features that not long ago were ignored by most practitioners. It is difficult for Perl to fit into one of the traditional categories of "imperative," "object-oriented" or "functional" languages. Perhaps it is an indication that such categorizations are becoming obsolete. Despite novel characteristics, the usage of Perl in teaching computer science is usually restricted to specialized topics, such as web programming. Although not suitable for most introductory level students, the vast flexibility of Perl can be a valuable addition to discussions on programming language foundations and design. It can complement functional languages such as Scheme and ML, and serve as a bridge between the characteristics of these (mainly) research-oriented languages and realistic programming practice. The academic community should take advantage of the success of this unique language in disseminating the most advanced ideas in programming language theory and design.

REFERENCES

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press; 2nd edition 1996

[2] Damian Conway. *A Taste of Perl 6.* The Linux Magazine, April 2003.

[3] M-J Dominus. *Perl contains the lambda calculus.* http://perl.plover.com/lambda/

[4] Allison Randal, Dan Sugalski, and Leopold Totsch. *Perl 6 Essentials*. O'Reilly & Associates, 2003

[5] Sriram Srinivasan. *Advanced Perl Programming.* O'Reilly & Associates, 1997.

[6] Larry Wall, Tom Christiansen, Jon Orwant. *Programming Perl* (3rd ed.). O'Reilly & Associates, 2000.

APPENDIX : The *mkfun* routine

In the following procedure, a user-defined hash table representing a declarative function is processed and transformed into a genuine Perl subroutine. Replacement by pattern substitution (`s///`) is only used to generate the initial preamble of the subroutine. The names of bound variables in the body of the user-defined function is not effected.

```
sub mkfun
  {
      my $fhash = $_[0]; # hash table with user defined function
      my @cles = (keys %$fhash); #cases of the definition
      my @vals = (values %$fhash); # bodies of the cases
      my @paramvars; my ($var0, $bool0);
      sub {
          my $indp;
          my $match0 = 0;  # boolean
          my $counter0 = 0;
          my @mykeys; my $val; my @vars; my $bool;
          my $funbody = 'print "no matches\n"'; # body
          my $varform;  my $preamble;
          @paramvars = @_;
          while ((!$match0) && (counter <= $#vals))
          {
             if ($#paramvars > 0)
                 { @mykeys = split / *, */, $cles[$counter0]; }
             else
                  { $mykeys[0] = $cles[$counter0]; }
               $val = $vals[$counter0++];  # a function body
             $bool = '1';  $preamble = '';  # identities
             for ($indp=0;$indp<=$#paramvars;$indp++)
               { $key = $mykeys[$indp];  # e.g., '$n:$n>1' or '1'
```

```
              $varform = split /:/, $key;  # form of parameter
              if ($varform < 2) # not in name var:boolean form
                   { $vars[$indp]="\$paramvars[$indp]";
                          $bool0="\$paramvars[$indp] eq $key";
                   }
                 else
                  {
                    ($vars[$indp],$bool0) = split /:/, $key;
                    $preamble = $preamble .
                          "my $vars[$indp] = \$paramvars[$indp]; ";
                  }
                 if ($indp>0)
                    {$bool0 =~ s{$var0}{\$paramvars[$indp-1]};}
                    # above line applies patterns previously seen.
                 $var0 = $vars[$indp]; # don't change original
                 $var0 =~ s/\$/\\\$/;
                   $bool0 =~ s{$var0}{\$paramvars[$indp]};
                 $bool = $bool." and ($bool0)";
                 $val =~ s{$var0}{(eval $vars[$indp])};
               } #for $indp loop
               if (eval $bool)
                 { $match0 = 1;
                   $funbody = "$preamble  $val";
                 } # if
              } # while more cases
           eval $funbody;
         }  # returned sub
    }  # mkfun
```

Procedures that can be processed by the above routine has the restriction that a boolean condition for a parameter can only contain variables that occur to the left of the :. That is,

```
      $max->{'$n:1, $m:$m<$n'} = '$n';
```

is acceptable, but not:

```
      $max->{'$n:$m>$n, $m:1'} = '$m';
```

The *my* variables in `mksub` should be considered reserved names. It is assumed that the user will define each case of the function as taking the same number of parameters. It is also necessary that the cases of the definitions are mutually exclusive.