J. Functional Programming **1** (1): 1–000, January 1999. Printed in the United Kingdom

Pure Untyped λ-Calculus and Popular Programming Languages

Mark-Jason Dominus
Plover Systems Co.
255 South Warnock Street, Philadelphia, Pennsylvania, USA 19107
`mjd-perl-lc@plover.com`

Abstract

Although the λ-calculus has very few features, these features are absent from most popular industrial and commercial programming languages. However, the Perl language does contain the necessary features and can therefore represent λ-calculus expressions natively, with λ-terms represented directly as anonymous function values and the language's own built-in evaluator serving to evaluate λ-calculus expressions; a synthetic evaluator is unnecessary. A Perl program is presented which demonstrates arithmetic computations using only λ-calculus forms, including a recursive addition function constructed with a fixed-point combinator.

The λ-calculus can be viewed as a very simple programming language, one which is a subset of, for example, Scheme. In fact the list of features necessary for a language to contain the λ-calculus is very short:

1. First-class function values
2. Lexical closure
3. No interfering semantic contraints

It must be possible to construct function values dynamically and pass them as arguments to other functions. Lexical closure is necessary to prevent conflict between formal parameter names in different functions. A language may also fail to contain the λ-calculus if it has certain kinds of constraints on function application. For example, the type system of Standard ML does not include a type suitable for lambda terms.

In languages with these properties, writing an interpreter for λ-calculus expressions is trivial, because the language itself is already such an interpreter. Instead of representing lambda terms as strings or more complex data structures, and instead of building an evaluation function, one represents lambda terms as the function values already supplied by the language, and uses the language's own built-in evaluation semantics.

Of the three features, first-class function values are the hardest to come by in the commercial and industrial programming world. For example, none of the popular languages C, Fortran, or COBOL has first-class function values, and so none contains the λ-calculus.

However, in the past ten years, a popular language has emerged which *does* have these features: Perl. Originally developed in 1987 by Larry Wall as a cross between AWK and C, it was used for many years as a systems programming language for Unix systems. Since then, it has become very widely used in many environments, most notable in programs that deliver dynamic content to Web browsers. Perl 5, released in late 1994, supported first-class function values with lexically scoped variables. This version of Perl contains the λ-calculus as a subset, as this article will demonstrate.

The program at the end of this article is an implementation of a λ-calulus-style recursive addition function, in Perl. When executed by the Perl interpreter, it computes and displays the value of $1 + 1$. It uses only the two λ-calculus operations of function abstraction and function application. Because of this, the code should be intelligible to anyone who understands these four syntactic features of Perl:

1. The variable x is denoted by `$x`.
2. `sub { my $x = shift; B }` is the lambda-term $(\lambda x.B)$. That is, it is the function which, when applied to an argument A, yields the value B with x replaced by A. The parameter x is lexically scoped, elimintating the need for α-conversions.
3. `$P->($Q)` is the application (PQ), in which the lambda-term P is applied to the lambda-term Q.
4. `#` introduces a comment that continues to the end of the line.

Perl's other features, such as arithmetic, string operations, and pattern-matching, are not used.

Perl uses call-by-value semantics, so delaying tactics are used to prevent premature evaluation of the consequents of the `IF` function and premature expansion of recursive function calls. To express $(IF\ condition\ x\ y)$ we write instead

$$(IF\ condition\ (\lambda q.x)\ (\lambda q.y))\ FORCE$$

where $FORCE$ is arbitrary. (In the code, it is taken to be the identity function.)

In the comments in the program, the symbol `%` represents λ, so, for example, the identity function $\lambda x.x$ is represented as `%x.x`. We sometimes abbreviate `%m.(%n.B)` as `%mn.B`.

The implementation uses a list representation of numerals rather than Church numerals.(In this representation, the numerals are constructed from pairs as follows:

0			(true, true)
1		(false,	(true, true))
2	(false,	(false,	(true, true)))
...			...

This is just to provide an excuse to demonstrate a recursive addition function. Church Numerals may also be implemented in the Perl λ-calculus subset in the usual way; such an implementation is available from the JFP web site.

```
################################################################
#
# Lambda Calculus demonstration in Perl
# 4 April 1999
# M-J. Dominus (mjd-perl-lc@plover.com)
# http://www.plover.com/~mjd/perl/lambda/
#

############
#
# Truth and truth testing

# IF = %btf.btf
$IF = sub { my $b = shift;
            sub { my $t = shift;
                  sub { my $f = shift;
                        ($b)->($t)->($f);
                      }
                }
          }
;

# TRUE = %xy.x
$TRUE = sub { my $x = shift;
              sub { my $y = shift;
                    $x;
                  }
            }
;

# FALSE = %xy.y
$FALSE = sub { my $x = shift;
               sub { my $y = shift;
                     $y;
                   }
             }
;
```

```
############
#
# Pairs (conses)

# PAIR = %xyb.bxy
$PAIR = sub { my $x = shift;
              sub { my $y = shift;
                    sub { my $b = shift;
                          ($b)->($x)->($y);
                        }
                  }
            }
;

# FIRST = %p.(p TRUE)
$FIRST = sub { my $p = shift;
               ($p)->($TRUE);
             }
;

# SECOND = %p.(p FALSE)
$SECOND = sub { my $p = shift;
                ($p)->($FALSE);
              }
;
```

```
############
#
# Numerals
#

$ZERO = ($PAIR)->($TRUE)->($TRUE);

$SUCC = sub { my $n = shift;
              ($PAIR)->($FALSE)->($n);
            }
;

$ONE = ($SUCC)->($ZERO);
$TWO = ($SUCC)->($ONE);

$IS_ZERO = sub { my $n = shift;
                 ($FIRST)->($n);
            }
;

$PRED = sub { my $n = shift;
              ($SECOND)->($n);
            }
;


############
#
# We use this utility function only to check to see that the
# addition was done correctly
#

sub convert_to_perl_number {
  my $n = shift;
  my $o = 0;
  until ($IF->($IS_ZERO->($n))->(1)->(0)) {
    $o++; $n = $PRED->($n);
  }
  return $o;
}
```

```
############
#
# Fixed-point combinator
# YM has the property that
#    (YM f Q) = f(YM f Q)
# for any Q
#
# YM = %f.(%x.(%y.f (x x)))(%x.(%y.f (x x)))

$YM = sub { my $f = shift;
            (sub { my $x = shift;
                   sub { my $y = shift;
                         $f->($x->($x));
                       };
                 })->
              (sub { my $x = shift;
                   sub { my $y = shift;
                         $f->($x->($x));
                       };
                 })

          }
;

# Q may be anything, so we use something simple:  %x.x
$FORCE = sub { my $x = shift; $x };
```

```
############
#
# Here's the basis for the addition function
# R = %g.(%mn.(IF (IS_ZERO m)
#                 (%q.n)
#                 (%q.(g FORCE) (PRED m) (SUCC n))
#             ) FORCE
#        )

$R = sub { my $g = shift;
           sub { my $m = shift;
                 sub { my $n = shift;
                       $IF->($IS_ZERO->($m))
                         ->(sub { my $q = shift; $n} )
                           ->(sub { my $q = shift;
                                    ($g->($FORCE))
                                         ->($PRED->($m))
                                         ->($SUCC->($n));
                                  })->($FORCE);
                     }
               }
         }
;

# This constructs the actual addition function
$ADD = $YM->($R)->($FORCE);

# Add 1+1 with recursive addition function
$mystery_result = $ADD->($ONE)->($ONE);
print "1+1 = ", convert_to_perl_number($mystery_result), "\n";
```