Adding Lambda Expressions to Forth

Angel Robert Lynas and Bill Stoddart

August 4, 2006

Abstract

We examine the addition of Lambda expressions to Forth. We briefly review the Lambda calculus and introduce a postfix version of Lambda notation to guide our approach to a Forth implementation. The resulting implementation provides the basic facilities of an early binding functional language, allowing the treatment of functions as first-class objects, manipulation of anonymous functions, and closures.

1 Introduction

The Lambda Calculus was developed by Alonzo Church during the 1930's as a general model of computation [4]. The original motivation was to investigate the notion of solvability [3], but the Calculus later formed a theoretical basis for functional programming. Recently there has been an interest in adding "lambda expressions" to imperative languages. They are proposed for vsn 3.0 of C#, but are not yet part of the ECMA C# Standard[1] which does however, largely support their functionality through "anonymous methods". The Open Standards Working Group for C++ recently produced a discussion paper "Lambda expressions and closures for C++"[8]. In this paper we discuss their incorporation within RVM Forth [5].

The functional programming (FP) paradigm arises from the mathematical idea of a function as a mapping from inputs to outputs; a procedure inside the function (the "body") operates on a variable to produce a return value in terms of that variable. The variable is instantiated by an argument to the function's single parameter, and the return can be numeric or some other type, including another function. In FP, therefore, functions are treated as first-class objects, that is, having the same status as variables and constants; they can also be arguments to functions, and manipulated as temporary anonymous objects.

While Forth does have some facilities to work with functions and operations with the stack, using execution tokens and allowing vectored execution, it does not provide the full generality required by the functional programming model.

Implementing a Lambda facility in Forth presents certain points of interest. A consistent postfix syntax must be developed, and to guide our design we have produced a postfix version of Lambda notation. The function objects produced must be dealt with in a consistent way to maintain compatibility with the abstract idea of the calculus, and the use of bound and free variables in Lambda calculus has to be integrated with the use of stack-frame local variables in RVM-Forth, requiring the provision of "closures" for dynamically-created functions.

In the rest of the paper, we provide a brief introduction to Lambda Calculus, its expression in postfix form, and the RVM-Forth implementation. We examine the roles of local variables and their binding in Lambda calculus, and how this can be integrated with a local variable system in a procedural language, such that persistent bindings can be maintained despite them having originated in a local context. Finally we review the ongoing project of which this work forms a part. A more detailed view of our implementation techniques is included as an appendix.

2 Lambda Calculus

2.1 Lambda Calculus

The general form for a lambda expression is:

$$\lambda < \text{name} > . < \text{body} >$$

In the above, <name> is the parameter or *bound variable* in the function — lambda functions have a single parameter only. The process of substituting an argument for the parameter in an application is known as β-reduction (shown as $\xrightarrow{\beta}$), which substitutes occurrences of the bound variable in the body with the argument expression. Once thus instantiated, the variable cannot change value.

The <body> itself can be anything from a simple operation on the variable to other nested functions, including embedded function applications.

The simplest example would be the identity function:

$$\lambda\, x.x$$

which returns its argument; so

$$(\lambda\, x.x)\ a \ \xrightarrow{\beta}\ a$$

For further examples, we allow the use of arithmetic operations[1] So the func-

[1] Neither these nor numbers are initially present in the basic lambda calculus, which is concerned with representing them in terms of more primitive substitution patterns.

tions:

$$\begin{aligned}&\lambda\, x.x+1\\&\lambda\, x.x*x\end{aligned}$$

would respectively increment their argument by one, and square it.

$$(\lambda\, x.x*x)\ 5 \ \xrightarrow{\beta}\ 5*5=25 \tag{1}$$

These examples all have a single bound variable in the body; variables in the body which are not bound by the immediate lambda declaration are known as *free variables*; these make little sense computationally unless they are in turn bound by an enclosing function. In this example:

$$\lambda\, y.x-y \tag{2}$$

the variable x is free. It may, however be bound by an enclosing function whose *body* is (2):

$$\lambda\, x.(\lambda\, y.x-y)$$

This is the basic way to deal with two or more arguments in Lambda calculus. The variable x has now become a local variable from an outer scope. We use an eager evaluation approach which requires it to be instantiated *before* the expression containing it is evaluated.

As regards application, parameters from left to right (i.e. outermost first) are substituted by arguments in the same direction, thus in an application of the above:

$$(\lambda\, x.(\lambda\, y.x-y))\ \ 10\ \ 3\ \xrightarrow{\beta}\ (\lambda\, y.10-y)\ \ 3 \tag{3}$$

The first reduction, substituting 10 for x, now returns a *function* which subtracts its argument from 10; here y would be substituted by 3 in the next reduction.

$$\xrightarrow{\beta}\ 10-3=7$$

An argument may well be another function. The expression:

$$\lambda f.(\lambda\, y.f\ \ y)$$

is a function that applies any given function f to any given argument y, for instance given the function $\lambda\, x.x+1$ for f and the number 7 for y

$$\begin{aligned}&(\lambda f.(\lambda\, y.f\ \ y))\ (\lambda\, x.x+1)\ 7\\&\xrightarrow{\beta}\ (\lambda\, y.(\lambda\, x.x+1)\ \ y)\ 7\\&\xrightarrow{\beta}\ (\lambda\, x.x+1)\ 7\\&\xrightarrow{\beta}\ 7+1=8\end{aligned}$$

2.2 A Postfix Notation for Lambda Calculus

Function application in Lambda notation is usually shown by a bracketed function followed by an argument. Postfix will require the argument to appear first. Conventional Lambda notation uses brackets to delimit the scope of a bound variable, and for that we will use a specific endλ symbol. Finally conventional Lambda notation uses brackets to control when a function is applied so it can be taken from the stack, followed by a definition body or a symbol (defined earlier) standing for it. However, this would not be automatically applied; an additional symbol is required analogous to Forth's `EXECUTE`. The symbol we use for this is a tick $\boxed{'}$. This usage has a history going back at least as far as Principia Mathematica, where the application of function f to an argument x is written as $f'x$.

We can now present some examples from the earlier section in an abstract postfix notation, with the actual RVM-Forth code in the next section. We use the notation *infix* $\quad\rightsquigarrow\quad$ *postfix* to show how the infix and postfix forms correspond. Taking example (1), we have:

$$(\lambda\, x.x * x)\ 5 \quad\rightsquigarrow\quad 5\ \ \lambda\, x.x\ \ x\ \ *\ \ \text{end}\lambda\ \ '$$

The keyword endλ ends the anonymous definition; at this point it could be assigned to a suitable variable, or applied, or left on the stack. The tick ensures application. Instantiation of the variable x by 5 (beta reduction) now yields the postfix expression "5 5 $*$".

The nested definition example (3), runs thus:

$$(\lambda\, x.(\lambda\, y.x - y))\ 10\ 3 \quad\rightsquigarrow$$
$$3\ 10\ \ \lambda\, x.\,\lambda\, y.x\ \ y\ \ -\ \ \text{end}\lambda\ \ \ \text{end}\lambda\ \ '\,'$$

Note that the arguments follow a stack order, first at the top. Above that is the function, however, with outer and inner variables as yet uninstantiated. The return value of the outer function will be the inner function with its unbound variables bound.

The first tick will execute the outer definition $(\lambda\, x)$, which will instantiate x (anywhere within scope) to the value 10 from the top of the stack:

$$\xrightarrow{\beta}\ 3\ \ \lambda\, y.10\ \ y\ \ -\ \ \text{end}\lambda\ \ '$$

This leaves 3 and the inner function on the stack — now with its x bound to a constant. The remaining tick executes this, instantiating y to 3, and returning "10 3 $-$".

3 RVM-Forth Implementation

3.1 Local Variables and Lambda Parameters

RVM-Forth already has a facility for local variables, with the syntax :

```
:  <opname> ... (:  VALUE <varname> ... :)  ... nLEAVE ... ;
```

The value for the local is taken from the stack — it can be an argument to the operation, or supplied by an expression within it. The keyword nLEAVE (where n can be 0, 1, 2, or 3) specifies the number of values left on the stack after the local environment goes out of scope.

As an example we have a program to calculate the greatest common divisor of two numbers using Euclid's algorithm, in which the smaller of the pair is subtracted from the larger to give a new pair. This process is repeated until the two numbers are equal:

```
: GCD0 ( n1 n2 -- n3, pre n1>0 & n2>0, post n3 = gcd(n1,n2) )
  (: VALUE X  VALUE Y :)
    BEGIN
      X Y <>
    WHILE
      X Y >
      IF
        X Y - to X
      ELSE
        Y X - to Y
      THEN
    REPEAT
    X
  1LEAVE ;
```

Values `X` and `Y` are initialised from the stack, `X` taking the value of n1 and `Y` the value of n2. `1LEAVE` specifies that just one item (the current top of stack) will be returned.

Additional locals may be declared between the `:)` and the `nLEAVE`; they will be initialised from the top of the stack, so suitable values should be found there.

For the lambda implementation this format is used to represent the parameter for the lambda expression, instantiated from the stack.

As a matter of style, it might be noted that we use the same word `VALUE` for both global and local variables, with the latter version being defined in a `COMPILER` wordlist which is only searched when in Compile mode.

Arrays and pointers to arrays are implemented in RVM-Forth, and they also have their local analogues. The declarations here for both global and local are `VALUE-ARRAY` and `VALUE-ARRAY^` (the full syntax is described in the RVM Manual [5]. We present a brief example below, though not of a kind that one would ever use. A global array is declared and initialised:

```
4 VALUE-ARRAY GLOBARR     ( 4-element storage )
HERE 4 , 10 , 20 , 30 , 40 , to GLOBARR
.GLOBARR  10 20 30 40 ok  ( Demo print defined offstage )
```

Next an operation is defined to reverse the elements in it. This has a local pointer to the global array, and an internal local array into which the reversed values are written, in a loop. Finally, the contents of the local array are copied to the global one.

```
: AREV ( -- )
   GLOBARR (: VALUE-ARRAY^ GRR :) ( points to GLOBARR )
   size of GRR VALUE-ARRAY LRR    ( empty local array )
   size of GRR 1+ VALUE ASIZE     ( loop size )
   ASIZE 1 DO
      ASIZE I - of GRR
      to << I >> of LRR
   LOOP
   LRR to GLOBARR                 ( copy to global... )
   0LEAVE ;
```

One would, of course, be more likely to use such a local array as a "safe" copy of a global, to manipulate temporarily or ensure read-only access.

3.2 Syntax and examples

The Forth uses two keywords for the λ symbol itself. The word `:LAMBDA` opens a definition at the outer level, that is, the system enters compile mode The corresponding endλ to this is `ENDLAM;`. The basic form is:

```
:LAMBDA (:  VALUE <name> :)  <body> 1LEAVE ENDLAM;
```

So far this is just an alternative syntax for the Forth Standard `:NONAME`. However, lambda definitions may also appear within compiled code, where they are bracketed with the `LAMBDA` and `ENDLAM`. They may be nested to any depth.

The simple example (1) from page 3 translates from the abstract postfix notation as:

$$5 \;\; \lambda\, x.x \;\; x \;\; * \;\; \text{end}\lambda \;\; ' \;\; \rightsquigarrow$$

```
5 :LAMBDA (: VALUE X :) X X * 1LEAVE ENDLAM; EXECUTE
```

When evaluated this will leave 25 on the stack.

The example where a function forms part of the body, (3) on page 3, provides an illustration of binding from outside the function itself:

$$3\ 10\ \ \lambda\, x.\, \lambda\, y. x\ y\ \ -\ \ \mathrm{end}\lambda\ \ \ \mathrm{end}\lambda\ {}'\,{}'\ \ \rightsquigarrow$$

```
3 10
:LAMBDA (: VALUE X :)
   LAMBDA (: VALUE Y :)
      X Y - 1LEAVE
   ENDLAM 1LEAVE
ENDLAM; EXECUTE EXECUTE
```

The fact that `X` is free in the inner lambda is unremarkable in a straightforward execution such as this. However it is possible to name the inner function, using the keyword `OP`. This picks up the name from the input stream and assigns it to the execution token at the top of the stack, creating a global named operation. Instead of the final line in the above code, we could, for instance, have:

```
  ... ENDLAM; EXECUTE OP MINUS
```

This gives us a named function which seems to refer to a variable `X` declared in a now-defunct scope and relating to a stack frame which no longer exists.

What has happened is that the evaluation of the outer `:LAMBDA` has instantiated the variable `X` to 10, so that when the inner lambda, i.e. the code:

```
 ... LAMBDA (: VALUE Y :) X Y - 1LEAVE
```

is evaluated, `X` already has a value, and this is what is copied in place of X within the inner lambda definition.

A final example demonstrates embedded function execution within the body of another function. The standard infix lambda expression:

$$(\lambda\, z.(\lambda\, y.y + (\lambda\, x.x + y * z)\ (y + z)))\ 3\ 4$$

contains an embedded function application inside the $\lambda\, y$ definition:

$$(\lambda\, x.x + y * z)\ (y + z)$$

in which the y and z will have been substituted by arguments by the time this is evaluated. The expression as a whole converts to postfix lambda notation as:

$$4\ 3\ \ \lambda\, z.\, \lambda\, y.y\ \ y\ \ z + \lambda\, x.x\ \ y\ \ z * +\ \mathrm{end}\lambda\ \ \ {}' +\ \ \mathrm{end}\lambda\ \ \ \mathrm{end}\lambda\ {}'\,{}'$$

The second argument to the final "+" is provided by the *result* of the inner function application

In RVM-Forth this becomes:

```
4 3 :LAMBDA (: VALUE Z :)
  LAMBDA (: VALUE Y :)
    Y Y Z +
    LAMBDA (: VALUE X :)
      X Y Z * +
    1LEAVE ENDLAM EXECUTE  ( embedded function application)
    + 1LEAVE ENDLAM
  1LEAVE ENDLAM;
EXECUTE EXECUTE
```

The evaluation can be calculated "by hand" as follows:

"$\longrightarrow$ substituting 3 for `Z`"

```
4  LAMBDA (: VALUE Y :)
    Y Y 3 +
    LAMBDA (: VALUE X :)
      X Y 3 * +
    1LEAVE ENDLAM EXECUTE
    + 1LEAVE ENDLAM
EXECUTE
```

"$\longrightarrow$ substituting 4 for `Y`"

```
    4 4 3 +
    LAMBDA (: VALUE X :)
      X 4 3 * +
    1LEAVE ENDLAM EXECUTE
    +
```

"$\longrightarrow$ substituting 7 for `X`; definition immediately compiled and executed, which evaluates `7 4 3 * +` as second argument to final `+`"

```
    4  19  +
```

Leaving 23.

3.3 Stack Frame Locals and Bindings

We see that the compilation procedure for local lambda definitions differs from normal procedure in the way it treats local variables declared outside the scope of the definition. Instead of being compiled to access the value from the associated slot in the current stack frame, a reference to this outer scope local is compiled initially as a push of some dummy value. The location of the data field for the push and the frame stack slot are recorded in a "bindings table". When *execution* reaches the local operation, these dummy values are replaced by the current values of the local variables referenced.

This process is known as "closure". The original variables cannot be assigned to from inside the resulting operation. The execution token thus produced must remain permanently viable. Since the code that produced it could be executed many times with different instantiations of any outer scope local variables, the execution token cannot point to the original compiled code for the operation, but must reference some relocated code[2], which embodies the particular closure that has been formed.

This relocated code could have been kept on the heap, but as heap space tends to be non-executable in modern configurations, it's kept instead in a separate area in the RVM's code space — and managed as a stack. This simplifies the management considerably, and also simplifies garbage collection (using the history stack) on reverse execution.

3.4 Closures and Local Operations

Moving beyond a purely functional approach in which Lambda expressions have no concept of state, we can use the technique of closures (i.e. instantiating outer scope locals by their current value) to define words which interface to data objects such as counters, stacks, or queues. These can use named local operations to provide persistent access to the data area created by the main operation.

The defining word `OP` encountered earlier in section 3.2 has a local analogue, also called `OP`, which is defined in the `COMPILER` wordlist and names operations local to an enclosing operation. The kind of local operations we now consider, however, must retain global scope, and are therefore named using the keyword `FORTHOP`. This is defined in the `COMPILER` wordlist, but is otherwise identical to the global version of `OP`.

In the stack example which follows, both the stack pointer and stack data area are declared as instance variables; respectively, as `INSTANCE-VALUE` and `INSTANCE-VALUE-ARRAY`. These are references to reserved space on the heap, and ensure that each created stack has its own independent pointer and data.

[2]RVM Forth is a native code Forth which has an *option* to compile relocatable code, as required here

```
: BUILD-STACK ( n --, n is size of stack, leaves tokens for push,
              pop, depth and clear)
  (: VALUE STACKSIZE :)
    0 INSTANCE-VALUE SP ( the stack pointer, 0 for empty stack  )
    STACKSIZE INSTANCE-VALUE-ARRAY STACK ( the stack body )
    LAMBDA ( x --, push )
      SP STACKSIZE = ABORT" Stack full"
      SP 1+ to SP
      to << SP >> of STACK
    ENDLAM FORTHOP
    LAMBDA ( -- x, pop )
      SP 0 = ABORT" Stack underflow"
      SP of STACK  SP 1- to SP
    ENDLAM FORTHOP
    LAMBDA ( -- n, depth of stack )
      SP
    ENDLAM FORTHOP
    LAMBDA ( -- , clear stack)
      0 to SP
    ENDLAM FORTHOP
  OLEAVE ;
```

This code leaves four execution tokens on the stack; the keyword `FORTHOP` picks up the names from the input stream, so they would typically be created and named in the same line:

```
4 BUILD-STACK PUSHA POPA DEPTHA CLRA
4 BUILD-STACK PUSHB POPB DEPTHB CLRB
```

The stacks themselves are anonymous, the named operations being the interface for each stack (A & B); these remain persistent, and operate on the data relating to their own stack.

Unlike the outer scope variables referred to earlier, which were declared with `VALUE` in their stack frame, the stack instance variables for pointer and data *can* be altered by the lambda operations.

Some sample runs and error checks:

```
10 PUSHA 20 PUSHA 30 PUSHA 40 PUSHA ok
DEPTHA . POPA . DEPTHA .   4  40 3 ok
5 PUSHB 15 PUSHB 25 PUSHB 35 PUSHB  ok
45 PUSHB Error: PUSHB
Stack full
reported at BUILD-STACK in file lamstack.r line 6
DEPTHB . 4 ok            ( error leaves data intact )
POPB .  35 ok
```

```
CLRA DEPTHA .  0 ok      ( clear the first stack )
POPA  Error: POPA
Stack underflow
reported at BUILD-STACK in file lamstack.r line 11
```

4 Conclusions and Further work

We have converted the standard Lambda Calculus into a postfix form and implemented it in Forth in order to extend the latter's Functional Programming capabilities. Along the way we have seen how the issue of bindings for lambda-style variables can be reconciled with the use of local variables in a procedural language, to provide persistence when execution scope passes beyond the original local scope. Also we have described an extension to this using instance variables to enable locally defined lambda operations to function as a global interface to an anonymous data structure.

The work reported here is part of a more general programmme of research in which we are seeking to exploit reversible computations to provide a more expressive implementation level language [6] for the the B Method [2] and similar formal development methods [7]. RVM Forth is a reversible version of Forth designed as an implementation platform for such methods. These methods typically provide a very expressive "specification" language in which to describe what a program is required to do. This language, which would generally include Lambda expressions, is not directly executable, and the developer must write the corresponding "implementation". This must then be proved correct with respect to its specification. Integration of the implementation level features described in this paper will allow Lambda expressions to be incorporated into the implementation level of a formal development, along with the features we have reported in previous articles, such as backtracking, general implementation of sets and automatic garbage collection on reverse computation.

References

[1] ECMA Technical Committee 39. Standard ECMA-334 C# Language Specification 4th edition, June 2006.

[2] J-R Abrial. *The B Book*. Cambridge University Press, 1996.

[3] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58, 1936.

[4] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[5] W. J. Stoddart. The Reversible Virtual Machine. User and technical manuals, 111 pages, University of Teesside, UK, July 2006. Available from www.scm.tees.ac.uk/formalmethods.

[6] W. J. Stoddart and F. Zeyda. Expression Transformers in B-GSL. In D. Bert, J. P. Bowen, S. King, and M. Walden, editors, *ZB2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 197–215. Springer, June 2003.

[7] W. J. Stoddart, F. Zeyda, and A. R. Lynas. A Design-based model of reversible computation. In *UTP'06, First International Symposium on Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, June 2006.

[8] J. Willcock, J. Jarvi, D. Gregor, B. Stroustrup, and A. Lumsdaine. Lambda expressions and closures for C++. Technical report, Open Standards Working Group 22, C++, 2006.

Appendix: A more Detailed View of the Implementation of Closures

When execution reaches `ENDLAM`, an execution token for the corresponding operation is left on the stack. A local variable declared before the local operation and used within it is treated in a special way. It is not possible to assign to it within the local operation, but it is possible utilise its current value. Instead of being compiled (as would normally be the case for local variables) to access the value from the associated slot in the current stack frame, a reference to such an "outer scope" local is initially compiled as a push of some dummy value, and the location of the data field for the push and the frame stack slot associated with the variable are recorded in a "bindings table". When execution reaches the local operation these dummy values are replaced by the current values of the local variables in question; this process is known as "closure".

The execution token produced in this way must remain permanently viable. Since the code that produced it could be executed many times with different instantiations of any outer scope local variables, the execution token cannot point to the original compiled code for the operation, but must reference some relocated code which embodies the particular closure that has been formed. An obvious place to hold such code would be on the heap, but since there is a growing tendency to configure heap space as non-executable we choose to use a separate area within the RVM's code space. It so happens that we can manage this area as a stack rather than as a heap, and this simplifies this aspect of our implementation considerably. The dynamic code area is managed by the `ANONCP` pointer, code is pushed to this stack by `PUSHCODE`, which also primes the history stack so that the memory utilised will be released on reverse execution.

We have now introduced all the elements required for an implementation of closures, and the next stage is to describe the form of the compiled code generated from an anonymous operation, let's say `LAMBDA S ENDLAM`. The compiled code is as follows:

```
| jmp | offset | btp | code for S | push code addr | PLUG | XT |
              |                    |
              ----------------------
```

The code begins with a jump which passes control past the code for S to code which pushes the address of the code for S onto the stack. This branch is followed by a pointer to the binding table for S and the code for S itself. Immediately following the push code address, we see two compiled operations, `PLUG` and `XT`.

`PLUG` has signature (xt – xt). It takes the address of the code for S and uses this to locate the binding table. It then writes in the actual values of any outer scope locals used in `S` into the reserved slots in the code, thus forming the closure. `PLUG` also leaves the address of the code for S still on the stack. `XT` has signature (xt1 – xt2). It relocates the code for S to the dynamic code area and leaves its new execution address as xt2.

The bindings table used for the evaluation of a local operation S exists on the heap and records all local references in S that are declared before `LAMDA`, i.e. all locals declared in an outer scope. For each such reference we record in the bindings table:

- the address within s to be instantiated, held as an offset from the start of the operation,
- the nesting level of the `LAMBDA..ENDLAM` construct;
- the slot number of the local within its stack frame.

The bindings table entries follow an entry count which is held in the first cell of the table.

The binding table is built during the compilation of the anonymous operation's definition, at which time the address of the binding-table-ptr entry for the function is held on a dedicated stack. Entries may be pushed to or popped from this stack with `>ANON` and `ANON>`. The use of such a stack is required to support the compilation of nested `LAMBDA .. ENDLAM` structures. Space for the table is requested by `LAMBDA` and the table is resized by `ENDLAM`.

When compilation encounters a local variable within a local operation, it must decide whether the instance is an outer scope local. To allow this to be done we maintain a process value `DLEVEL` which holds the current nesting level of a local operation. When a local variable is declared within a local operation its nesting level is recorded as an entry within its parameter field. When the local variable is subsequently encountered, its declaration time nesting level is compared with the actual nesting level recorded in `DLEVEL`. If it is less the local is an outer scope local. `DLEVEL` also serves to record when compilation is within a local operation, and this is used to select compilation of relocatable rather than absolute code.