Closures — the Forth way

M. Anton Ertl*
TU Wien

Bernd Paysan
net2o

Abstract

In Forth 200x, a quotation cannot access a local defined outside it, and therefore cannot be parameterized in the definition that produces its execution token. We present Forth closures; they lift this restriction with minimal implementation complexity. They are based on passing parameters on the stack when producing the execution token. The programmer has to explicitly manage the memory of the closure. We show a number of usage examples. We also present the current implementation, which takes 109 source lines of code (including some extra features). The programmer can mechanically convert lexical scoping (accessing a local defined outside) into code using our closures, by applying assignment conversion and flat-closure conversion. The result can do everything one expects from closures, including passing Knuth's man-or-boy test and living beyond the end of their enclosing definitions.

1 Introduction

The addition of locals, quotations[1] and `postpone` to Forth leads to the question of how these features work together. In particular, can a quotation access the locals of outer definitions (i.e., is the quotation a closure)? The Forth200x proposal for quotations chose not to standardize this, because there is too little existing practice in the Forth community; however, it does encourage system implementors to experiment with providing such support, and this is what we did for the present paper.

In other programming languages, particularly functional programming languages, access to outer locals is a valuable feature that increases the expressive power[2] of these languages.

Also, can a local be `postpone`d, and if yes, what does it mean? Forth-94 chose to not standardize this.

However, implementing these features in its most powerful and convenient form requires garbage collection, which is not really appropriate for Forth. So we have to find a good compromise between expressive power and convenience on one hand, and ease of implementation on the other. The contribution of this paper is to propose such a compromise.

In this paper, we first present the principles and syntax of our new features (Section 2); next we give usage examples for these features (Section 3), as well as alternatives that do not use them; next we give an overview of the implementation (Section 4); then we discuss the relation between our flat-closure feature and lexical scoping (Section 5); we also give some microbenchmark results that give an idea of the performance of our implementation (Section 6); finally, we discuss related work (Section 7).

2 Closures: Principles and Syntax

2.1 Overview and principles

Quotations have been accepted into the next version of the Forth standard in 2017, but they do not define what happens on access to locals of enclosing definitions. Consider the following minimal example:

```
: foo {: x -- xt :}
  [: x ;] ;

: bar {: x -- xt1 xt2 :}
  [: x ;]
  [: to x ;] ;

5 foo 6 foo
execute . execute . \ prints 6 5

5 bar over execute . \ prints 5
6 swap execute
execute . \ prints 6
```

Some people may wonder what this means. It is not necessary to know this to understand most of this paper (we use a different syntax), but in case you really want to know, the rest of this paragraph

*anton@mips.complang.tuwien.ac.at

[1] Nameless colon definitions inside other colon definitions, http://www.forth200x.org/quotations.txt

[2] The expressive power refers not just to what can be expressed (all interesting languages are Turing-complete and can compute the same things, given enough resources), but also to the ease and versatility of expression.

explains it. Following the example of Scheme[3], every invocation of `foo` (or `bar`) creates a new instance of the local `x`, and an xt (two xts for `bar`) for the quotation. Calling this xt (these xts) accesses the instance of `x` that was created in the invocation of `foo` that produced the xt. Yes, this means that different invocations of `foo` produce different xts.

Terminology: In the programming language literature, a nested definition (or quotation) that accesses a local of an enclosing definition is called a **closure**. This is also the name of a data structure used for implementing this feature. We provide the data structure, and call it **closure**, but leave closure conversion (the process by which other programming language compilers translate from source-code closures to data-structure closures) to the programmer. In most of the rest of this paper, **closure** refers to the data structure, and it's Forth source code representation.

We do not support the syntax shown above. Instead, in the minimal version of our syntax, these words can be written as follows:

```
: foo {: x -- xt :}
  x [{: x :}d x ;] ;

: bar ( x -- xt1 xt2 )
  align here swap , {: xa :}
  xa [{: xa :}d xa @ ;]
  xa [{: xa :}d xa ! ;] ;
```

`[{:` starts a closure and a definition of passed-in locals of the closure.

The decisive difference between a closure and a quotation that starts with a locals definition is that the locals of the closure are initialized from the values that are on the data (and FP) stack at the time when the quotation's xt is pushed on the data stack, while a quotation with locals at the start would take the locals from the stack when the xt is `execute`d (maybe much later). In this way, the closure gets data from its enclosing definition that it can use later.

The other difference is that the locals definitions in these closures end with `:}d`, and that means that the memory needed for the closure is stored in the dictionary (`allot`ed space).

These examples demonstrate the principles of our approach:

Explicit memory management of closures: Closures can live longer than the enclosing definition. The programmer decides where the memory for closure is allocated, and how it is reclaimed. The memory can be allocated and reclaimed like locals, allocated with `allocate` and reclaimed explicitly with `free`, allocated in the dictionary, or allocated with some user-defined allocator (such as the Forth garbage collector[4], or region-based memory allocation [Ert14]).

Copying locals into closures: Locals in a closure are a separate copy of the outer local when used in the way shown above. For read-only locals, this is no problem.

This approach of creating copies of values of read-only locals is known as **flat-closure conversion**. In other programming languages, the compiler performs flat-closure conversion implicitly (or uses a different implementation approach); in our Forth extension, the programmer performs it explicitly.

Explicit management of writable locals: For writable locals, we usually do not want separately modifyable copies, but want to access one *home location*. In our approach, home locations are allocated (and memory managed) explicitly (with `align here swap ,` in the `bar` example). The addresses of these home locations are read-only and copied into the closures, like other read-only values. The home locations are accessed with memory words, such as `@` and `!`, as shown in the `bar` example. This approach is called **assignment conversion**.

Our syntax is more verbose, but also more flexible than simply allowing access to outer locals: The locals in the closures can have a different name from the corresponding locals in the enclosing definition, and actually, there is no need to define a value as a local in in the enclosing definition. E.g., we could also define these words as follows, and achieve the same effect:

```
: foo ( x -- xt )
  [{: x :}d x ;] ;

: bar ( x -- xt1 xt2 )
  align here swap ,
  dup  [{: xa :}d xa @ ;]
  swap [{: xa :}d xa ! ;] ;
```

2.2 Closure words

These words are used for defining and memory-managing closures (without conveniences for dealing with read/write locals).

[3] The most popular of the early programming languages that got this right.

[4] http://www.complang.tuwien.ac.at/forth/garbage-collection.zip; however, the current version of the garbage collector does not recognize closures as live by seeing their xt, because the xts do not point to the start of the memory block.

`[{:` (C: -- closure-sys) Compilation: Start a closure, and a locals definition sequence.

`:}d` (C: closure-sys -- quotation-sys colon-sys) Compilation: End a locals definition sequence.

Enclosing definition run-time: Take items from the data and FP stack corresponding to the locals in the definition sequence, create a closure in the dictionary. The `;]` that finishes the closure pushes the xt of that closure.

`:}h` (C: closure-sys -- quotation-sys colon-sys) Like `:}d`, but the closure is `allocate`d (the `h` stands for *heap*).

`:}l` (C: closure-sys -- quotation-sys colon-sys) Like `:}d`, but the closure is created on the locals stack[5] in the enclosing scope. I.e., it lives as long as a local defined in the same place.

`:}*` (C: closure-sys xt -- quotation-sys colon-sys) A factor of `:}d :}h :}l`, usable for defining similar words for other allocators. The passed xt has the stack effect `( u -- addr )` and allocates `u` address units (bytes) of memory.

`:}xt` (C: closure-sys -- quotation-sys colon-sys) Similar to `:}*`, but the xt is pushed at the enclosing definition run-time, before `[{:`. Usage example: `['] allocd [{: x :}xt x ;]`

`>addr` (xt -- addr) `Addr` is the address of the memory block of the closure identified by `xt`. Typical use: `( xt ) >addr free throw`.

2.3 Gforth features

This subsection describes some Gforth features that make the closure words nicer to use, or that are used in the examples in the rest of the paper.

Locals definers

Gforth cannot just define cell-sized locals, but also, e.g., FP locals, by putting `f:` before the local. An old [Ert94], but (up to now) little-used feature is *variable-flavoured locals* where using a local pushes the address of its location on the data stack, and accesses to the values are performed with words like `@ !`. Variable-flavoured locals are defined by putting one of `w^ f^ d^ c^` before the name of the local (for a cell, a float, a double, or a char respectively). Given that writable locals in closures are based on passing the address of the home location of the local around, this feature finally becomes interesting. Example:

```
{: f: r w^ x :}
r f. 1e to r
x @ . 1 x !
```

This code fragment first defines a value-flavoured FP local `r`, and then a variable-flavoured local `x`, then shows a read and a write access to `r`, then a read and a write access to `x`.

VFX Forth supports defining local buffers, which can also be used for defining home locations for read/write locals that live until the definition is exited.

Gforth also has a *defer-flavoured* locals definer: if you define a local `x` with the definer `xt:`, an ordinary occurence of `x executes` the xt in `x`; you can also use `is` and `action-of` on `x`. Example:

```
['] . {: xt: y :}
5 y \ prints 5
['] drop is y
```

Convenient `postponeing`

Instead of writing a long sequence of `postpone`s, e.g.,

```
postpone a postpone b postpone c
```

you can write

```
]] a b c [[
```

An implementation of this feature in standard Forth is available at http://theforth.net/package/compat/current-view/macros.fs.

Modifying words

`Set-does>` `( xt -- )` is a modern variant of `does>`. It changes the last defined word to first push its body address, and then perform the xt. E.g., instead of

```
: myconst ( n -- )
  create ,
does> ( -- n )
  @ ;
```

you can write

```
: myconst ( n -- )
  create ,
  ['] @ set-does> ;
```

`Set-optimizer ( xt -- )` changes the last defined word `w` such that it `executes` xt whenever `compile,` is called with the xt of `w` as parameter. You can use this to generate better code for `w`. E.g., you can have myconst generate better code:

```
: myconst ( n -- )
  create ,
  ['] @ set-does>
  [: >body @ ]] literal [[ ;] set-optimizer ;
```

[5] Or on the return stack on systems that keep locals there.

2.4 Auxiliary closure words

The following are convenience features. One can eliminate them from code without requiring deep changes, but the code becomes longer and less readable.

Home location conveniences

We can use variable-flavoured locals to create home locations that live until the end of the definition, but for longer lifetimes, allocating home locations of multiple locals is inconvenient: If they are allocated separately, this may cost extra memory and require extra effort on deallocation; if they are allocated at once, we have to get individual home location addresses with address computations or with structure words.

Our current implementation reuses some of the existing code to provide the following convenience for creating home locations:

```
<{: w^ a f^ b :}h a b ;>
```

This creates a home location for cell `a` and float `b` on the heap, and then (between `:}h` and `;>` pushes the addresses on the stack; finally, the `;>` pushes the address of this home location block so that it can be `free`d at the end of the lifetime.

For implementation simplicity reasons, locals from outside cannot be used inside `<{:...;>`, and the locals defined inside cannot be used outside. That's why the addresses of the home locations are passed on the data stack to the outside.

If we did not have `<{:...;>`, one would have to write the following code to replace the code above:

```
0 cell+ faligned float+ allocate throw
dup cell+ faligned over
```

So, while `<{:...;>` is more cumbersome than one would like, it is better than nothing; and it is very simple to implement.

`Postpone` locals

Given a local `x`, `postpone x` is equivalent to `x postpone literal`. This is especially convenient in combination with `]]...[[` (see below).

However, the generated code compiles the value that `x` had when the `postpone` runs, not the value `x` has at run-time, so the following example will produce results that some may not expect:

```
: foo
  7 {: a :} postpone a 8 to a ; immediate
: bar foo ;
bar . \ prints 7
```

Therefore we recommend that one should not apply `postpone` and `to` to the same local. It would be relatively easy to warn of this combination, but, for now, our implementation does not.

Allocation

These are variants of existing memory allocation words that fit the stack effect expected by `:}*` and `:}xt`.

`alloch` (size -- addr) A variant of `allocate` with a different stack effect.

`allocd` (size -- addr) A variant of `allot` with a different stack effect.

3 Closure Usage and Alternatives

This section gives some examples for uses of closures. We also show alternatives that do not use these features (sometimes before, sometimes after the usage examples), so you get a better impression of whether closures provide benefits for the example, and what they are.

In stack effect comments, we use `...` to indicate additional data and/or FP stack items. For a stack effect comment `( ... x y -{}- ... z )`, the number of stack items represented by `...` normally does not change.

3.1 Numerical integration

Higher-order words are words that take an xt and call it an arbitrary number of times.

A classical use of words that take an xt (in other languages, a function) as argument is numerical integration (also known as *quadrature*):

```
numint ( a b xt -- r )
\ with xt ( r1 -- r2 )
```

This approximates $\int_a^b \mathrm{xt}(x)dx$.[6] Now consider the case that we want to compute $\int_a^b 1/x^y dx$ for a given a, b, and y, and want to have a word for this:

```
: integrate-1/x^y ( a b y -- r )
  [{: f: y :}l ( r1 -- r2 ) y fnegate f** ;]
  numint ;
```

So the stack element y is consumed (and stored in the local y during closure construction, and then

[6] A practical word would have one or more additional parameters that influence the computational effort necessary and how close the result is to the actual value of the integral.

used during the repeated calls to the closure performed by `numint`.

Another way in which we might express this computation is:

```
: 1/x^y ( y -- xt )
 [{: f: y :}h ( x -- r ) y fnegate f** ;] ;

( a b y ) 1/x^y dup numint >addr free throw
```

`1/x^y` takes y and produces an xt. The xt takes x and produces the result. This technique of splitting a function with multiple arguments into a sequence of functions, each with one argument is called currying. It allows a more uniform treatment of functions, which is useful in conjunction with higher-order functions, and is therefore common in functional programming.[7]

A difference between these variants is that in the latter the local y lives after the definition returns in which it was defined. Therefore, we used `:}l` in the first variant, but `:}h` (and `>addr` and `free`) in the second.

A Forth-specific alternative is to pass y on the (FP) stack rather than through a local. In order to do that, `numint` has to be modified to have the following stack effect:

```
numint ( ... a b xt -- ... r )
\ with xt ( ... r1 -- ... r2 )
```

I.e., `numint` has to ensure that xt can access the values on the stack represented by `...`. Now we can write:

```
: integrate-1/x^y ( a b y -- r )
  frot frot ( y a b )
  [: ( y x -- y r2 )
    fover fnegate f** ;]
  numint fswap fdrop ;
```

The stack handling takes some getting-used-to. For a single level of higher-order execution, as used here, this is manageable.

If we want something like the currying variant, this could look like this:

```
: 1/x^y ( y x -- y r )
  fover fnegate f** ;

( a b y ) frot frot ' 1/x^y numint
fswap fdrop
```

We don't get a properly curried function here, but instead a function that reads the the extra argument from the (FP) stack without consuming it, the same as the quotation in the other pass-on-the-stack variant.

If you need several functions with such extra arguments in one computation (for both pass-on-the stack variants), the functions have to be written specifically for the concrete usage (e.g., one reads the second and third stack item, while another reads the fourth stack item, etc.), not quite in line with the combinatorial nature of currying.

In any case, it is a good practice to design higher-level words such that the called xts have access to the stack below the parameters: Move the internal stuff of the higher-level word elsewhere (return stack or locals) before `execute`ing xts.

3.2 Sum-series

Franck Bensusan posted a number of use cases[8], among them one for writing a word that computes $\sum_{i=1}^{20} 1/i^2$, as an example of computing specific elements of a series.

This can be written as follows, factoring out reusable components, and going all-in with locals:

```
: for ( ... u xt -- ... )
    \ xt ( ... u1 -- ... )
    {: xt: xt :} 1+ 1 ?do i xt loop ;

: sum-series ( ... u xt -- ... r )
    \ xt ( ... u1 -- ... r1 )
    0e {: f^ ra :}
    ra [{: xt: xt ra :}l ( ... u1 -- ... )
          xt ra f@ f+ ra f! ;] for ra f@ ;

20 [: ( u1 -- r )
      dup * 1e s>f f/ ;] sum-series f.
```

In accumulating/reducing words like `sum-series`, we need to update a value in every iteration. In this variant, we update a local. A variant without closures differs in the following definition:

```
: sum-series ( ... u xt -- ... r )
  \ xt ( ... u1 -- ... r1 )
  0e swap [: ( ... xt r1 u1 -- ... xt r2 )
    {: f: r :} swap dup >r execute r> r f+
  ;] for drop ;
```

This puts `r` in a local in the quotation in order to get it out of the way. This is not needed for the particular way we use the word, but it allows to use `sum-series` in other contexts, too. It is the price we pay for being able to use this as a higher-order word without needing closures.

An in-between variant that is better than either variant above is:

[7] Interestingly, working with higher-order and curried functions allows a programming style that avoids local variables; still, general locals are useful in implementing curried functions. There are alternatives, however [Bel87].

[8] news:<8ea09174-ddac-4d5b-b906-df3bd4f07932@googlegroups.com>

```
: sum-series ( ... u xt -- ... r )
  \ xt ( ... u1 -- ... r1 )
  0e [{: xt: xt :}l ( ... u1 r1 -- ... r2 )
       {: f: r :} xt r f+ ;] for ;
```

This passes the xt through the closure mechanism, and the intermediate result on the stack.

3.3 Man or boy?

Knuth's man-or-boy test [Knu64] is an Algol 60 function that has no purpose other than to test whether a compiler implements lexical scoping correctly. In Algol:

```
begin
  real procedure A(k, x1, x2, x3, x4, x5);
  value k; integer k;
  real x1, x2, x3, x4, x5;
  begin
    real procedure B;
    begin k := k - 1;
          B := A := A(k, B, x1, x2, x3, x4)
    end;
    if k <= 0 then A := x4 + x5 else B
  end;
  outreal(A(10, 1, -1, -1, 1, 0))
end;
```

In Forth[9]:

```
: A {: w^ k x1 x2 x3 xt: x4 xt: x5 | w^ B :}
  recursive
  k @ 0<= IF  x4 x5 f+  ELSE
    B k x1 x2 x3 action-of x4
    [{: B k x1 x2 x3 x4 :}L
      -1 k +!
      k @ B @ x1 x2 x3 x4 A ;] dup B !
      execute  THEN ;
10 [: 1e ;] [: -1e ;] 2dup swap [: 0e ;] A
f.
```

This example allocates all locals and all home locations on the locals stack.

Given the purpose of this example, we did not try to find an alternative without closures.

3.4 testr

McCarthy [McC81] presents the following Lisp function (in M-expression syntax) by James R. Slagle, which revealed that the Lisp implementation of the time did not implement lexical scoping:

```
testr[x,p,f,u] <-
  if p[x] then f[x]
  else if atom[x] then u[]
  else testr[cdr[x],p,f,
         lambda:testr[car[x],p,f,u]].
```

[9] Call Gforth with `gforth -l128k`

> The object of the function is to find a subexpression of x satisfying p[x] and return f[x]. If the search is unsuccessful, then the continuation function u[] of no arguments is to be computed and its value returned. ([McC81])

To implement this in Forth, we use the following words for accessing S-Expressions:

atom (s-expr -- f) is the s-expression an atom (true) or a pair (false)?

car (s-expr -- s-expr) the first half of a pair

cdr (s-expr -- s-expr) the second half of a pair

In Forth with closures, the equivalent is:

```
: testr {: x p f u -- s :} recursive
  \ x is an s-expression
  \ p is an xt ( s-expr -- f )
  \ f is an xt ( s-expr1 -- s-expr2 )
  \ u is an xt ( -- s-expr )
  \ s is an s-expression
  x p execute if x f execute exit then
  x atom if u execute exit then
  x cdr p f
  x p f u [{: x p f u :}l
    x car p f u testr ;] testr ;
```

This could also be written using `xt:`, but the number of required `action-of`s would exceed the number of eliminated `execute`s.

The reason for dealing with the unsuccessful search by calling u is that f can return any S-expression, so there is no way to indicate an unsuccessful search through the return value. Of course, in Forth, we have the option of returning such an indication as additional return value, so we can implement `testr` without closures:

```
: testr1 {: x p -- s1 f :} recursive
  x p execute if x true exit then
  x atom if nil false exit then
  x cdr p testr1 dup if exit then
  x car p testr1 ;

: testr {: x p xt: f xt: u -- s :}
  x p testr1 if f exit then
  drop u ;
```

3.5 Defining words

The `create...does>` feature of Forth has a number of problems:

- It does not allow optimizing read-only accesses to the data stored in the word.

- When multiple cells (or other data) are stored in the word, it becomes hard to follow across the `does>` boundary what is what.
- First `create` produces a word with one behavior, then `does>` changes the behaviour (and this can theoretically happen several times). This causes problems in implementations that compile directly to flash memory.

In the following we focus on the first two problems.

`+field`

The first problem is exemplified by:

```
: +field ( u1 u "name" -- u2 )
   create over , +
does> ( addr1 -- addr2 )
   @ + ;

\ example use
1 cells 1 cells +field x ( addr1 -- addr2 )
: foo x @ ;
```

Using `set-does>`, `+field` is written as:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  create over , +
  [: @ + ;] set-does> ;
```

With the built-in `+field`, VFX compiles `foo` into `MOV EBX, [EBX+04]` (3 bytes). However, with the user-defined definition of `+field` above, this is not possible: the user could change the value in `x` later (e.g., with `0 ' x >body !`), and the behaviour of `foo` has to change accordingly. Therefore, VFX produces a an 8-byte two-instruction sequence instead.

With closures, we can write `+field` as follows:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  create over
  [{: u1 :}d drop u1 + ;] set-does>
  + ;
```

The `drop` is there to get rid of the body address of *name*, which the `set-does>` mechanism (like `does>`) pushes automatically.

In this variant, *u1* is transferred to *name* through the closure mechanism; its value does not change (there is no `to u1`), so the compiler can generate efficient code for `foo`. Currently there is no compiler that does that, but a compiler that inlines the closure when *name* is compiled and that is analytical about locals should not find it difficult.

A way to solve this problem without closures is to define the defining word based on `:` instead of `create`:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  over >r : r> ]] literal + ; [[ + ;
```

With this `+field`, VFX produces the same code for `foo` as with the builtin `+field`. This can be made slightly easier to read by using a local, and `postpone`ing it:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  {: u1 u :} : ]] u1 + ; [[ u1 u + ;
```

Another approach for dealing with the read-only problem is to declare the memory as not-going-to-change after initializing it (supported in iForth):

```
: +field ( u1 u "name" -- u2 )
  create over , +
  here cell- 1 cells const-data
does> ( addr1 -- addr2 )
  @ + ;
```

Yet another approach is to change the intelligent `compile,` to compile fields efficiently:

```
: +field ( u1 u "name" -- u2 )
  \ name execution: ( addr1 -- addr2 )
  create over , +
  [: @ + ;] set-does>
  [: >body @ ]] literal + [[ ;]
  set-optimizer ;
```

This works in Gforth (development version), and, with a different syntax, in VFX. `Set-optimizer` changes the last defined word (i.e., the one defined by `+field1`) so that `compile,`ing it calls the quotation; that first fetches the field offset (at compile time, not at run-time), compiles it as a literal and then compiles the `+`. A disadvantage of this approach is that the optimizer has to implement nearly all of the `does>` part again; and such redundancy can make errors hard to find (e.g., the word works fine when interpreted, but acts up when compiled).

We can use closures instead of the body to pass `u1`:

```
: +field ( u1 u "name" -- u2 )
   create
   over [{: u1 :}d drop u1 + ;] set-does>
   over [{: u1 :}d drop ]] u1 + [[ ;]
   set-optimizer
   + ;
```

This demonstrates the redundancy nicely. A disadvantage of this approach is that the redundancy now also costs memory, because two closures are stored in the dictionary.

Finally, there was a proposal for `const-does>` [Ert00], but it did not generate much interest. The code would look as follows:

```
: +field ( u1 u "name" -- u2 )
  over + swap ( u2 u1 )
1 0 const-does> ( addr1 -- addr2 )
  ( addr1 u1 ) + ;
```

The `1 0` tells `const-does>` to take one data stack item and 0 FP stack items from these stacks when `const-does>` is called, and push them on these stacks when the defined word is performed. The body address of the `created` word is not pushed, `addr1` is passed by the caller of *name* (typically the base address of the structure containing the field), `u1` by `const-does>`.

Interface-method

The `+field` example is easy to understand, but the following, larger example is better for demonstrating the effects. It also demonstrates the second problem of passing several values across the `does>` boundary.

The following is a simplified variant of the word for defining interface method selectors in `objects.fs` [Ert97]:

```
\ fields: object-map selector-offset
\         selector-interface
\ structure (constant): selector

: interface-method ( n-sel n-iface -- )
  create here tuck selector allot
  selector-interface ! selector-offset !
does> ( ... object -- ... )
  2dup selector-interface @
  swap object-map @ + @
  swap selector-offset @ + @ execute ;
```

This example exhibits the read-only and the multiple-cells problem. The latter problem is attacked by organising these cells as a struct, storing into it in the `create` part, and reading from it in the `does>` part, but compared to the following closure-using variant, the code is still relatively complicated.

```
: interface-method ( n-sel n-iface -- )
  create [{: n-sel n-iface :}d
    drop dup object-map @ n-iface + @
    n-sel + @ execute ;] set-does> ;
```

This locals-using variant eliminates all the complications of storing the parameters in the `create` part. The `does>` part is also quite a bit simpler, as it avoids having to juggle the address of the `created` word.

The `:`-using definition looks as follows:

```
: interface-method {: n-sel n-iface -- :}
  : ]] dup object-map @
  [[ n-iface ]] literal + @
  [[ n-sel   ]] literal + @
  execute ; [[ ;
```

The resulting code (produced by VFX 4.72) for a call to a word defined with `interface-method` is:

```
 does> version          : version
MOV  EDX, 0 [EBX]      MOV  EDX, 0 [EBX]
ADD  EDX, [080C0BB4]   MOV  EDX, [EDX+04]
MOV  ECX, [080C0BB0]   CALL [EDX+04]
ADD  ECX, 0 [EDX]
CALL 0 [ECX]
```

If we can postpone locals, or, in this case, use them inside `]]...[[`, this can be further shortened into:

```
: interface-method {: n-sel n-iface -- :}
  : ]] dup object-map @ n-iface + @
       n-sel + @ execute ; [[ ;
```

The code between `]]` and `[[` is almost the same as the code in the closure in the closure version.

4 Implementation

This section describes our implementation of the features described in this paper.[10] Other implementations are possible, but are not discussed here, with one exception: Gforth uses a locals stack, and we always mention the locals stack here; but adapting the implementation for a system where the return stack serves as locals stack is not difficult.

4.1 Closures and execution tokens

The execution token for a closure represents not just the code, but also the passed locals. Yet it has to fit into a single cell.

Our implementation deals with that by a variant of the trampolines used by gcc for the same purpose: A block of memory is allocated; the start of this block contains the header of an anonymous word, and the rest contains the values of the locals defined at the start of the closure. The closure is represented by the xt of the anonymous word.

When the closure is performed, it copies the values of the locals to the locals stack. This means that the closure locals can be treated like ordinary locals in the rest of the definition. After this copying, the user-defined code of the closure is performed.

[10] http://git.savannah.gnu.org/cgit/gforth.git/tree/closures.fs

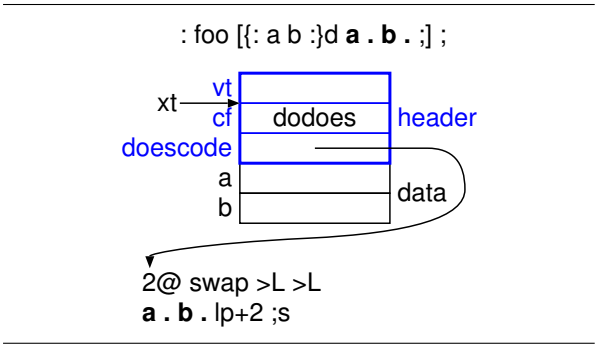

Figure 1: A definition containing a closure, and the memory representation of a closure created by invoking the definition; the part of the doescode before the bold part is generated by the compiler to copy the data of the closure to the locals stack.

In finer detail, a closure is an anonymous `create...does>` defined word (but it can reside not just in the dictionary, but alternatively on the locals stack or on the heap), where the code after the `does>` starts by copying the data from the body of the word to the locals stack, followed by the user-written code. Figure 1 shows an example.

The compile-time part of the closure implementation is deeply intertwined with the pre-existing implementations of locals and quotations in Gforth, and a detailed description will probably be of little benefit to implementors of other systems, but we still mention some interesting aspects: During closure construction, the locals stack pointer points to the memory for the closure (i.e., not always in the locals stack). The closure locals are arranged in the closure memory just as normal locals are arranged on the locals stack, they get the same offsets (using the same code as during ordinary locals definition), and after copying behave just as normal locals.

This part costs 78 source lines of code (SLOC, blank, and comment-only lines not counted).

4.2 Home locations

The home location syntax `<{:...;>` is based on the closure implementation: Creating a home location block differs from creating a closure by not producing a word header, and by letting the locals stack pointer point to the home location block until `;>`.

This part costs 6 SLOC.

4.3 Postpone locals

`Postpone`ing locals is implemented by special-casing locals in `postpone`.

This part costs 25 SLOC. It is so large because each of the nine locals definers needs a special case.

5 Lexical scoping and flat-closure conversion

The closure syntax presented above was originally designed for minimal implementation complexity, even at the expense of programmer inconvenience. However, looking at the examples, we now think that it is very appropriate for stack-based languages like Forth: For languages where the primary data location is the stack(s) rather than locals, it is appropriate to build closures from data on stacks rather than by copying existing locals.

However, this means that when we want to convert code from languages with lexical scoping to Forth, we have to perform flat-closure conversion manually. This section sketches how to do that, in a mechanical way. Alternatively, we can find a way to express the same purpose differently, as in the *testr* example (Section 3.4), but there is no mechanical way to do that, and no guarantee that there is such a way.

This also demonstrates that our closure syntax is as powerful as lexical scoping in Algol-family languages. There is no mechanical process for converting the automatic memory reclamation of, e.g., Scheme to manual memory reclamation. If we stick with mechanical conversion for that part, we either have to live with leaking memory, or use some kind of garbage collection for the closures.

We use the following contrived program with lexical scoping as a running example.

```
: foo ... {: a b :} ...
  [: ... {: c :} ... to a ... b ...
     [: ... to b ... c ... ;] ... ;] ... ;
```

This program only contains definitions and uses of locals, and quotations. Other operations can be inserted in the places marked with `...`, but do not play a role in flat-closure conversion.

The first step is to perform *assignment conversion* [Dyb87, Section 4.5]: For every local that is accessed with `to` and also accessed in a quotation where it is not defined, we convert it into a home location and access it with `@` and `!`:

```
: foo ... <{: w^ a w^ b :}d a b ;>
  drop {: a b :} ...
  [: ... {: c :} ... a ! ... b @ ...
     [: ... b ! ... c ... ;] ... ;] ... ;
```

In this example, we allocated the home locations in the dictionary, and then `drop`ped the address containing the home location block. Note that you need to use `w^` only when defining the home location; in the rest of the code, the addresses are passed around as values, so value-flavoured locals are fine there.

The next step is the actual flat-closure conversion: You have to mention all locals accessed inside the closure in the locals definition at the start of the closure, and pass them to the closure on the stack:

```
: foo ... <{: w^ a w^ b :}d a b ;>
  drop {: a b :} ...
  a b [{: a b :}d ... {: c :}
    ... a ! ... b @ ...
    b c [{: b c :}d ... b ! ... c ... ;]
    ... ;] ... ;
```

5.1 Alternative syntaxes and implementations

What if we tried to go for a syntax that supports lexical scoping directly instead of through manual flat-closure conversion? How much implementation complexity would that cost, and would the benefit be worth the cost? Are there intermediate approaches?

The first step towards lexical scoping is that closures get the values of the closure locals (those defined at the start of the closure) from same-named locals of the enclosing definition or closure, rather than from the stacks. This is relatively easy to implement, but it requires that the value is in a local. As the examples show, this requirement would often result in extra locals definitions, so implementing that is not necessarily an advantage.

The next step would be to completely hide the actual flat-closure conversion: The compiler would have to look at the whole code of the definition, and note which of the locals are used inside which quotation, and then convert the quotations into closures by itself. While that is not particularly hard, it requires looking at the whole definition at once, which would require a major rewrite for most Forth systems. The benefit would be that the code for `foo` shown after the assignment conversion step would work (with some adjustments for manual memory reclamation of closures).

Similarly, the assignment conversion step can be split into two steps:

In the first step, the programmer marks some locals on definition as requiring assignment conversion (with special local definers, e.g., `w!`). The compiler would then allocate a home location for these locals automatically, pass the address around, and automatically convert read accesses to fetches from the address, and `to` accesses to stores to the address. A `to` access to a local that is not marked as requiring assignment conversion produces an error if the local occurs in a quotation where the local was not originally defined. This step would require some work, but no deep changes to the usual compilers. The benefit would be that the programmer would avoid nearly all of the assignment conversion work, and only needs to mark some locals as requiring assignment conversion.

In the second step, the compiler collects the information about the locals requiring assignment conversion by itself, relieving the programmer of that duty. Again, it requires looking at the whole definition at once, but otherwise would not be a lot of work.

6 Performance

This section presents performance results from microbenchmarks on the current implementation in Gforth. Note that microbenchmarks have their pitfalls and in application usage effects may dominate that are not reflected in these microbenchmarks. Moreover, the current implementation has seen only minimal performance work (costing 4 source lines), and some of these benchmarks might see substantial speedups by investing more work in performance.

We have two kinds of microbenchmarks: Creating a closure (or an alternative to a closure), and running a closure (or an alternative). The closure we use is:

```
[{: x :}l x + ;]
```

Running a closure with one or two cells as above profits from the little performance work we have applied, so for the *run closure* benchmark we also measure a three-cell variant that exercises the general case:

```
[{: x y z :}l x + ;]
```

We use the following variants:

closure For creation, we measure the three different allocation methods (locals stack, dictionary, heap), with the heap variant including the `free` overhead.

does Create an anonymous `created` word with `x` in its body, with `[: @ + ;] set-does>`.

:noname create an anonymous `:noname` word which compiles `x` as literal in its body.

stack Use a quotation that uses `x` from the stack (without consuming it): `[: over + ;]`. Benchmarking its creation just means benchmarking pushing the xt.

We run the benchmark on a 4GHz Core i5-6600K (Skylake). We use 50,000,000 iterations for each microbenchmark, but report the cycles and instructions per iteration, subtracting the loop overhead. The results are:

cycles	inst.	per iteration
21.0	99.0	create closure local
62.9	183.5	create closure dictionary
113.6	459.0	create closure heap
735.1	2464.7	create does
5115.4	15159.5	create :noname
8.0	14.0	create stack
7.0	43.0	run closure 1 cell
21.3	85.0	run closure 3 cells
6.0	38.0	run does
6.2	27.0	run :noname
7.1	33.0	run stack

Note that results of 8 cycles or less in these microbenchmarks are usually dominated by dependency chains through instruction or stack pointers, and the relative performance may be different (maybe more like the relations of instructions counts) in applications.[11]

Still, there are some conclusions we can make:

Creating a local closure is relatively cheap, whereas creating heap and dictionary closures is quite a bit more expensive. Dynamically creating `create...does>` words instead is a lot more expensive, and the same with `:noname` is even more expensive. Pushing the xt of a quotation is cheap, as expected.

Running a closure with one cell is slightly more expensive than the other variants; the general case (3 cells) is quite a bit more expensive, but could be optimized, too; there will be very few cases where the number of runs/creation is so high that the does and :noname variants break even. The stack variant is cheap in both creation and run time.

7 Related work

Already Lisp [McC81] and Algol 60 allowed nested functions and accessing outer locals, but with limitations: Lisp initially used dynamic scoping; this was considered a bug by McCarthy (Lisp's creator) [McC81] (see Section 3.4), but that bug had entrenched itself as a feature in the meantime, and the Lisp family took a while to acquire lexical scoping (prominently in Scheme and Common Lisp). A reason for that is that Lisp allows returning functions, which in combination with lexical scoping creates the *upwards funarg* problem: local variables no longer always have lifetimes that allow to use a stack for memory management.

Algol 60 avoided the upwards funarg problem by not allowing to return functions. Still, lexical scoping (in combination with call-by-name) proved a challenge to implement, as can be seen by Knuth's man-or-boy test [Knu64] (see Section 3.3), which revealed that many Algol compilers failed to implement access to outer locals correctly.

The best-known ways to implement the access to outer locals are static link chains and the display [FL88]. They keep each local in only one place, and have relatively complex and sometimes slow ways to access them.

By contrast, in this paper we use the *flat-closure coversion* approach [Dyb87, Section 4.4] in combination with assignment conversion [Dyb87, Section 4.5], which replicates locals (or their addresses) in order to make the access cheap. Moreover, in typical Forth style, we only provide flat closures and home location support, and leave it to the programmer to perform assignment and flat-closure conversion manually. This makes the programmer responsible for optimizations in the conversion process [KHD12], and avoids the need to put values into locals in order to get them into closures.

Concerning memory management, most languages have chosen one of two approaches: 1) restrict function-passing or outer-locals access such that stack management is sufficient; or 2) don't have restrictions, and use garbage collection for the involved data structures when necessary.

After decades of growth in the functional programming community, using higher-order functions and passing functions to them has recently made the jump to mainstream languages like C++ (in C++11), Java (in Java 8), and C#. This feature is typically called *lambda*. The C++ variant[12] is extremely featureful, and, while too complex for Forth, inspires ideas on how such features can be implemented in close-to-the-metal languages.

Moving closer to Forth, Joy [vT01] is a stack-based functional language. It uses the term "quotation" for a nameless word that can be defined inside other words. Joy has no locals, so quotations in it cannot access outer locals.

Factor [PEG10] is a high-level general-purpose language with roots in Forth and Joy; it has quotations and locals, and allows access to outer locals.

Lynas and Stoddart [LS06] added lambda expressions with read-only lexical scoping to RVM-Forth. They implemented accesses to outer variables by compiling them as literals with placeholder values; when generating the xt, the code is copied, and the actual values of the outer variables are plugged into the code instead of the placeholder values. These code copies are not freed in forward execution.

Gerry Jackson implemented quotations with full lexical scoping and explicit deallocation of closures in Forth-94.[13] He managed to implement

[11] You may also wonder about the impossible apparent instructions per cycle (IPC) for some of the benchmarks, but note that you have to add the loop overhead (12 instructions in 6 cycles) to compute the actual IPC.

[12] https://en.cppreference.com/w/cpp/language/lambda

[13] news:<6b5eead4-f809-4dd4-81c6-16e1c2a9f613@q14g2000vbn.googlegroups.com>, http://qlikz.org/forth/archive/lambda.zip

all this functionality (but with some limitations) and workarounds for the limitations of Forth-94 in 312 SLOC (including an object-oriented package).

In contrast to these works, the present work abandons lexical scoping in favour of reducing the implementation effort, putting the onus of assignment and closure conversion on the programmer.

In 2017 the Forth200x committee has accepted a proposal[14] for quotations that does not standardize the access to outer locals, leaving it up to systems whether and how they implement accesses to outer locals.

Of course, in classical Forth fashion, some users explored the idea of what outer-locals accesses can be performed with minimal effort. In particular, Usenet user "humptydumpty" introduced rquotations[15], a simple quotation-like implementation that uses return-address manipulation. The Forth system does not know about these rquotations and therefore treats any locals accessed inside rquotations as if they were accessed outside. In the case of Gforth (as currently implemented) this works as long as the locals stack is not changed in the meantime; e.g., the higher-order word that calls the rquotation must not use locals.

There is no easy way to see whether this restriction has been met; this is also classical Forth style, but definitely not user-friendly. Static analysis could be used to find out in many cases whether the restriction has been met, but that would probably require more effort than implementing the approach presented in this paper, while not providing as much functionality.

8 Conclusion

Locals in standard Forth have a number of restrictions. In this paper we mainly looked at the restriction that, in a quotation, one can only access locals that have been defined in that quotation. But instead of adding the capability to access outer locals, we reduced it to the basic need to initialize locals of a quotation/closure from outside data, and presented syntax and an implementation of stack-initialized flat closures with explicit memory management. In addition, we present conveniences for defining home locations for writable locals, and for `postpone`ing (read-only) locals.

We presented a number of examples where these features allow additional, and sometimes shorter and easier-to-read ways to express the functionality. We also presented alternative code that does not use these features.

In these examples, the features provide some benefits. The implementation of flat closures alone costs 78 source lines in Gforth, or 109 source lines for all the features combined. Whether the benefits are worth this implementation effort will have to be seen.

Acknowledgments

The anonymous referees, Marcel Hendrix, Gerry Jackson, and Bill Stoddart provided valuable feedback on earlier versions of this paper.

References

[Bel87] Johan G.F. Belinfante. S/K/ID: Combinators in Forth. *Journal of Forth Application and Research*, 4(4):555–580, 1987. 7

[Dyb87] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, April 1987. 5, 7

[Ert94] M. Anton Ertl. Automatic scoping of local variables. In *EuroForth '94 Conference Proceedings*, pages 31–37, Winchester, UK, 1994. 2.3

[Ert97] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997. 3.5

[Ert00] M. Anton Ertl. `CONST-DOES>`. In *EuroForth 2000 Conference Proceedings*, Prestbury, UK, 2000. 3.5

[Ert14] M. Anton Ertl. Region-based memory allocation in Forth. In *30th EuroForth Conference*, pages 45–49, 2014. 2.1

[FL88] Charles N. Fischer and Richard J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA, 1988. 7

[KHD12] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. Optimizing closures in O(0) time. In Olivier Danvy, editor, *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, pages 30–35. ACM, 2012. 7

[Knu64] Donald Knuth. Man or boy? *Algol Bulletin*, page 7, July 1964. 3.3, 7

[14] `http://www.forth200x.org/quotations.txt`

[15] news:<f71bfb01-4b8e-49d6-abd5-12bda6dbfcd2@googlegroups.com>

[LS06] Angel Robert Lynas and Bill Stoddart. Adding Lambda expressions to Forth. In *22nd EuroForth Conference*, pages 27–39, 2006. 7

[McC81] John McCarthy. History of LISP. In Richard L. Wexelblatt, editor, *History of Programming Languages*, pages 173–197. Academic Press, 1981. 3.4, 7

[PEG10] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. Factor: a dynamic stack-based programming language. In William D. Clinger, editor, *Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, October 18, 2010, Reno, Nevada, USA*, pages 43–58. ACM, 2010. 7

[vT01] Manfred von Thun. Joy: Forth's functional cousin. In *EuroForth 2001 Conference Proceedings*, 2001. 7

A Sudoku

This appendix shows another example. It demonstrates the use of an xt-passing style in a larger application. The shown code is complex, and we do not expect you to understand it completely. But you can try to follow the stack flow in Fig. 3 and 4 to get an impression of the benefits and drawbacks of these two approaches, and also skim Fig. 5 to get an impression of that alternative.

In 2006, I (Ertl) wrote a Sudoku program.[16] In Sudoku the same constraints apply to rows and columns, and squares have a related constraint, so I tried to find a good factoring.

At one point[17] I factored out horizontal and vertical walks (of the fields in a row/column, or of the columns/rows of the whole Sudoku) into higher-order words `map-row` and `map-col` (see Fig. 2). I passed the extra parameters to the words called by these words through the stack. You can see these higher-order words in action in Fig. 3.

However, I found it hard to track the stack contents, because the words are not called in the order in which they appear in the code. Therefore I also found it hard to write and maintain this code, even though I used locals to make it a little less opaque. Soon after I switched to a different approach.

But before we look into that approach, let's consider how things would look with closures: Fig. 4. The code is shorter, but, what's more, it is much easier to see the data flow: Instead of following how the data items flow through the higher-order words to the `execute`d xts, the xts (produced from closures) have simple stack effects such as `( var -- )`. These xts do not use extra parameters;[18] instead, the data is passed through the closure mechanism. Note that there are two levels of closures, and accesses to data that originally came from one or two levels out.

The approach I actually switched to was quite different, though: Following advice from Andrew Haley, I created macros `do-row loop-row do-col loop-col` for performing the walks, and wrote `gen-row-constraints gen-col-constraints` and other words using these macros (Fig. 5). The result[19] feels more Forth-like and has seven lines less than the stack-using one (once we eliminate two now-unused words), but increases the dictionary size (including threaded code, excluding native code) on 64-bit Gforth 0.7.9_20180830 from 9168/9248 bytes (for xt-passing/closures) to 11544 bytes (the macros generate quite a bit of code each time they are used).

[16] https://github.com/AntonErtl/sudoku

[17] https://github.com/AntonErtl/sudoku/blob/da19285814c49a007dd8d954cf94a29f51fa51a1/sudoku3.fs

[18] The `var` in `( var -- )` is produced by the higher-order words that call the xt.

[19] https://github.com/AntonErtl/sudoku/blob/dc0f80bbbed8a7c488af7aecb5de0b7d5c5662ac/sudoku3.fs

```
\ gen-valconstraint ( var container xt -- )
\ check ( -- )
\ map-row ( ... row xt -- ... ) apply xt ( ... var -- ... ) to all variables of a row
\ map-col ( ... col xt -- ... ) apply xt ( ... var -- ... ) to all variables of a col
\ row-constraint ( var row -- )
\ col-constraint ( var col -- )
```

Figure 2: Helper words for Sudoku

```
: gen-valconstraint1 { xt container var -- xt container }
    var container xt gen-valconstraint
    xt container
    check ;
: gen-contconstraint { xt-map xt-constraint container -- xt-map xt-constraint }
    xt-map xt-constraint container dup [’] gen-valconstraint1 xt-map execute drop ;
: gen-row-constraints ( -- )
    check [’] map-row [’] row-constraint grid @ [’] gen-contconstraint map-col 2drop ;
: gen-col-constraints ( -- )
    check [’] map-col [’] col-constraint grid @ [’] gen-contconstraint map-row 2drop ;
```

Figure 3: Part of Sudoku program with higher-order words using the stack

```
: gen-contconstraint1 ( xt-map xt-constraint -- xt-contconstraint )
  [{: xt: map xt-constraint :}d ( container -- )
    xt-constraint over [{: xt-constraint container :}l ( var -- )
      container xt-constraint gen-valconstraint check ;] map ;] ;
: gen-row-constraints ( -- )
  check grid @ [’] map-row [’] row-constraint gen-contconstraint1 map-col  ;
: gen-col-constraints ( -- )
  check grid @ [’] map-col [’] col-constraint gen-contconstraint1 map-row ;
```

Figure 4: Part of Sudoku program with closures

```
\ replace MAP-ROW and MAP-COL with
\ do-row ( compilation: -- do-sys; run-time: row -- row-elem R: row-elem )
\ loop-row ( compilation: -- do-sys; run-time: R: row-elem -- )
\ do-col ( compilation: -- do-sys; run-time: col -- col-elem R: col-elem )
\ loop-col ( compilation: -- do-sys; run-time: R: col-elem -- )
: gen-row-constraints ( -- )
    check grid @ do-col
        dup do-row
            over [’] row-constraint gen-valconstraint check loop-row
        drop loop-col ;
: gen-col-constraints ( -- )
    check grid @ do-row
        dup do-col
            over [’] col-constraint gen-valconstraint check loop-col
        drop loop-row ;
```

Figure 5: Part of Sudoku program with macros