General Locals

M. Anton Ertl[*]
TU Wien

Abstract

In standard Forth, locals have the following restrictions: they cannot be `postpone`d; and a quotation cannot access a local defined outside it. This paper explores lifting these restrictions. It shows some use cases, how they would look with the restrictions lifted, and how workarounds for the restrictions look. It also presents an implementation approach and its syntax that is based on explicit allocation, closure conversion, and differentiating between read-only and read/write locals.

1 Introduction

The addition of locals, `postpone` and quotations[1] leads to the question of how these features work together. In particular, can a local be `postpone`d, and if yes, what does it mean? Forth-94 chose to not standardize this. And can a quotation access the locals of outer definitions? The Forth200x proposal for quotations chose not to standardize this, because there is too little existing practice in the Forth community. In other programming languages, particularly functional programming languages, however, access to outer locals is a valuable feature that increases the expressive power[2] of these languages.

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.

So this paper explores the topic of access to outer locals: it gives usage examples for these features and examples of alternatives that do not need these features (Section 2), presents an implementation approach (Section 3), and a syntax appropriate for this implementation (Section 4). We also discuss related work (Section 5).

[*]Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

[1]Nameless colon definitions inside other colon definitions

[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.

2 Usage and Alteratives

This section gives some examples for uses of locals beyond the standard, and how these uses can be avoided. For the general locals, we first show the standard syntax, and assume ideal behaviour: Every time a local definition is performed, it produces a new instance of the local that lives as long as it is needed (which may be unlimited); the uses in the definition refer to the instance produced during the same execution of the definition.

For defining floating-point locals, we use the Gforth syntax: Inside the `{:...:}` locals definition, each FP local is preceded by `F:`.

Then we show the syntax described in Section 4. It is probably a good idea to skip these definitions at first, and revisit them after reading Section 4. They are shown here to make comparisons with the other definitions easier. These definitions use the following words:

```
: alloc ( u -- addr ) allocate throw ;
: alloth ( u -- addr )
  align here swap allot ;
```

And we also show alternatives that do not use these features (sometimes before, sometimes after the usage examples).

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.

2.1 Higher-order words

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

Numerical integration

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 $\int_a^b \mathrm{xt}(x)dx$.[3] Now consider the case that we want to compute $\int_a^b 1/y^x dx$ for a given x, b, and y, and want to have a word for this:

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

The quotation computes $1/y^x$, with y coming from the local in the enclosing word `integrate-1/y^x`. In this case, the local is not used after the definition returns in which it was defined. In our syntax:

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

Another way in which we might express this computation is:

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

( a b y ) 1/y^x numint
```

`1/y^x` 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.[4]

A difference between these variants is that in the latter the local y lives after the definition returns in which it was defined.

In our syntax:

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

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

A Forth-specific alternative is to pass y on the 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/y^x ( a b y -- r )
  frot frot ( y a b )
  [: ( y x -- y r2 )
    fover fswap 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/y^x ( y x -- y r )
  fover fswap fnegate f** ;

( a b y ) frot frot ' 1/y^x 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.

Sum-series

Franck Benunsan posted a number of use cases[5], 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 outer locals:

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

[3]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.

[4]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].

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

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

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 this is performed by updating the local. In our syntax the definition of sum-series becomes:

```
: sum-series ( ... u xt -- ... r )
    \ xt ( ... u1 -- ... r1 )
    0e ['] alloca <{: xt f! r :} drop
    ['] alloca <[{: : xt r :} ( . u1 -- . )
           xt execute r f+ to r :] for r ;
```

A variant without outer locals 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 outer-locals access.

Sudoku

In 2006 I wrote a Sudoku program.[6] 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[7] 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. 1). 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. 2.

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

But before we look into that approach, let's consider how things would look with access to outer locals. Figure 3 shows the result in the ideal syntax: The code is shorter, but, what's more, it is much easier to see the data flow. Figure 4 show this approach in our syntax.

Instead of following how the data items flow through the higher-order words to the executed xts, the xts (produced from quotations) have simple stack effects such as (--) and (var --). These xts do not use extra parameters;[8] instead, the data is passed through locals. Note that there are two levels of quotations, and accesses to locals that are 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[9] 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 0.7.3 from 8360 to 10208 bytes (the macros generate quite a bit of code each time the are used).

2.2 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 one is exemplified by:

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

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

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

[9] 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 1: 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 2: Part of Sudoku program with higher-order words using the stack

```
: gen-constraints1 {: xt-map xt-constraint -- :}
  [: {: container -- :}
    container
    [: ( var -- )
      container xt-constraint gen-valconstraint check ;]
    xt-map execute ;] ;

: gen-row-constraints ( -- )
  check ['] map-row ['] row-constraint grid @ gen-constraints1 map-col ;

: gen-col-constraints ( -- )
  check ['] map-col ['] col-constraint grid @ gen-constraints1 map-row ;
```

Figure 3: Part of Sudoku program with higher-order words and locals (untested)

```
: gen-constraints1 {: xt-map xt-constraint -- :}
  ['] alloth <[{: container : xt-map xt-constraint -- :}
    container
    ['] alloca <[{: : container xt-constraint :} ( var -- )
      container xt-constraint gen-valconstraint check ;]
    xt-map execute ;] ;
```

Figure 4: `gen-constraints1` in our syntax

```
\ 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

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

\ example use
1 cells 1 cells +field x
: foo x @ ;
```

With the built-in `+field`, VFX compiles `foo` into `MOV EBX, [EBX+04]`. 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 general locals, we can solve it by writing `+field` as:

```
: +field ( u1 u "name" -- u2 )
   create over {: u1 :} +
does> ( addr1 -- addr2 )
   drop u1 + ;
```

`U1` is stored in a local, and then used at run-time. Because there is no `to u1` while `u1` is visible, the compiler knows that `u1` does not change, and can use the more efficient code for `foo`. The address pushed by `does>` is not needed, and therefore `dropped`.

As an intermediate step before showing our syntax, we switch to a variant of the above that uses a quotation:

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

This uses `set-does>` which changes the `create`d word to perform the code in the quotation, i.e., the quotation corresponds to the code after `does>` in classical Forth. The same in our syntax:

```
: +field ( u1 u "name" -- u2 )
   create over {: u1 :} drop +
   ['] alloth <[{: : u1 :} drop ( a1 -- a2)
     u1 + ;] set-does> ;
```

A standard way to solve this problem is to define the defining word based on `:` instead of `create`:

```
: +field ( u1 u "name" -- u2 )
  over >r : r> postpone literal postpone +
  postpone ; + ;
```

With this `+field`, VFX produces the same code for `foo` as with the builtin `+field`. The downside is that the definition of `+field` is not particularly easy to read. This can be made easier to read by allowing to `postpone` locals:

```
: +field ( u1 u "name" -- u2 )
  over {: u1 :} : postpone u1 postpone +
  postpone ; + ;
```

`Postpone u1` compiles an access to (the current instance of) `u1` into the defined word. In this case, there is no `to u1` in the definition, so we know that `u1` does not change, and we can compile the value of the local as a literal.

Our syntax for this example is the same, but the compiler behaviour is slightly different: A local like `u1` is always `postpone`d as a read-only local. Once it is `postpone`d, it can not be changed with `to`, and vice versa; there is a special syntax if we want both. This avoids the need to see the whole definition when `postpone`ing a local.

We can make this even shorter, using the non-standard `]]`...`[[` syntax:

```
: +field ( u1 u "name" -- u2 )
  over {: u1 :} : ]] u1 + ; [[ + ;
```

We expected the memory size of words defined with the `:`-based `+field` to be bigger, but that turned out not to be true for all systems, at least with the measurement method we applied: We measured the difference of `here` before and after repeated definitions of a field `x`:

	builtin	`does>`	`:`
Gforth 0.7.2 64-bit	48	48	72
VFXForth 4.72 IA32	48	48	32
SwiftForth 3.6.3 IA32		32	32
iForth 5.1 (64-bit)		32	128

For the bigger `interface-method` example below, the sizes are as follows:

	`does>`	`:`
Gforth 0.7.2 64-bit	56	136
VFXForth 4.72 IA32	48	48
SwiftForth 3.6.3 IA32	32	48
iForth 5.1 (64-bit)	32	128

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:

```
: +field1 ( u1 u "name" -- u2 )
  create over , +
does> ( addr1 -- addr2 )
  @ + ;

: +field ( u1 u "name" -- u2 )
  +field1
  [: >body @ postpone literal postpone + ;]
  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).

The same approach with `set-does>`, access to outer locals, and `]]`...`[[`, in our syntax:

```
: +field ( u1 u "name" -- u2 )
   create over {: u1 :} drop +
  ['] alloth <[{: : u1 :} drop ( a1 -- a2)
    u1 + ;] set-does>
  ['] alloth <[{: : u1 :} drop ( -- )
    ]] u1 + [[ ;] set-optimizer ;
```

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

Finally, there was a proposal for `const-does>` [Ert00], but it did not find much resonance. 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 `create`d word is not pushed, `addr1` is passed by the user (typically the base address of the structure containing the field).

Interface-method

The `+field` example is easy to understand, but the following, larger example is better for demonstrating the effects.

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 variant using general locals, the code is still relatively complicated.

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

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. In our syntax (and using `set-does>`):

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

The `:`-using standard definition looks as follows:

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

If you leave away the `postpone`s and the `literal`s, the code between `:` and `;` is the same as the code between the `drop` and the `;` in the general-locals version. We can use the non-standard `]]`...`[[` syntax to make this code easier to read:

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

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

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

As before, the locals are compiled as literals here, resulting in the same code as the version above.

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]
```

3 Implementation

This section describes one way to implement general locals. We think that this way offers a good compromise between ease of implementation and ease of use, as appropriate for Forth. The syntax is designed to make the implementation easy and Section 4 describes it.

3.1 Read-only and read/write locals

Read-only locals are defined with a specific value and that value is then not changed, i.e., `to` is not applied to them. By contrast, read/write locals can be changed with `to`.

The difference is important, because we can just make as many copies of read-only locals as is convenient. By contrast, one instance of a read/write local must be stored in one place (the home location), and we must arrange that various accesses to that local find that place.

That is relevant for access to outer locals, because keeping each value in only one place would be more complex than our actual implementation.

Our implementation implements a read-only local by copying the value to a place where it is easy to access, and a read-write local by copying the address of the home location to a place where it is easy to access; so read-write locals are accessed through one level of indirection.

3.2 Memory

Standard locals live at most until the end of the definition they are defined in, so they can be stored on the return stack, on a locals stack that also releases the locals on return, or in registers (with saving and restoring to, e.g., the return stack around calls); in the rest of this paper we talk about the locals stack, but the other options are equally viable.

General locals can live longer. Therefore, a different way to manage memory is needed. Bernd

Paysan suggested using explicit allocation, and that appears to be the right approach for Forth. There are several ways to allocate memory in Forth: `allot`, `allocate`, and user-defined memory allocators such a the Forth garbage collector[10] or region-based memory allocation [Ert14].

3.3 Execution Tokens

The execution token for a quotation that references outer locals represents not just the code, but also the outer 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 an anonymous word, and the rest contains the values (for read-only locals) or address (for read-write locals) of the outer locals. The trampoline word copies these values and addresses to the locals stack, then jumps to the code of the quotation.

By pushing the values of read-only locals to the locals stack, these locals can be accessed just like locals defined in the quotation. For read-write locals, there is one additional indirection, but for the most part, the same mechanism can be used. The advantages of this scheme are: We do not need a completely separate mechanism for accessing outer locals; the system does not need to keep a pointer to the trampoline around once it has started executing the Forth code of the quotation; and therefore the program can deallocate the trampoline block right after entering the word (unless the quotation is executed again later).

In addition to these trampoline blocks, we have another kind of memory block that we need to manage: blocks that contain the home location of read/write locals that are used as outer locals. Home locations can also be allocated as part of trampoline allocation, but there are also cases (e.g., in `sum-series`) where it is necessary to allocate them separately.

There is no need to keep read-only locals in home location blocks, because their values are copied directly from the locals stack to trampoline blocks.

3.4 `Postpone`

When `postpone`ing a read-only local, the compiler postpones its value as a literal.

When `postpone`ing a read/write local, the compiler postpones its home address as a literal, followed by postponing a fetch (as appropriate for the type of the local). If a system supports `postpone`ing a change to a local (e.g., `postpone ->x` in Gforth), the compiler postpones the home address followed by postponing a store.

[10]`http://www.complang.tuwien.ac.at/forth/garbage-collection.zip`

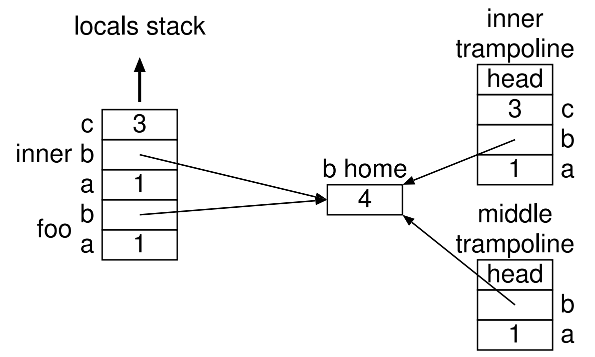

Figure 6: Data structures in our example just before leaving the inner quotation

3.5 Example

Consider the following example, which is contrived, but demonstrates most of the discussed implementation issues in a short space. It does not demonstrate explicit allocation, because that would require delaying the example until after the syntax is discussed. It also does not demonstrate multiple instances of the locals (there is only one call of each definition), because that would overload the picture.

```
: foo ( -- )
  1 2 {: a b :}
  [: \ middle
    3 {: c :}
    [: \ inner
       a . b . 4 to b
       :noname ]] b . c . ; [[
    ;]
  ;] execute execute b . execute ;

foo \ prints "1 2 4 4 3 "
\ the :noname word is equivalent to
\ :noname <b-home> @ . 3 . ;
```

Figure 6 shows the data structures at one point in time. They are built up as follows:

When entering `foo` and encountering the locals definition, the *b home* is allocated (using the method specified in the syntax described in Section 4), because `b` is a read/write local that has accesses from quotations. The value of `a` and the address of `b` are pushed on the locals stack.

Next the middle quotation is encountered, and results in building the middle trampoline and pushing its address as xt on the stack. Because both `a` and `b` are used inside the quotation, they are both copied to the trampoline.

The trampoline head details depend on the Forth system (in particular, how `execute` works). In a native-code system where `execute` just calls the trampoline, the trampoline could start with a call to code that copies the locals from the trampoline to the locals stack, followed by a jump to the code of the quotation.

Next the first `execute` is encountered, which invokes the trampoline of the middle quotation; this pushes `a` and `b` on on the locals stack, then enters the code of the quotation. The locals definition pushes the value of (read-only) `c` on the locals stack, then encounters the inner quotation, builds its trampoline, and pushes the address on the data stack. Given that `a`, `b`, and `c` are used in the inner quotation, they are all included in the trampoline. With that, the middle quotation is finished and its locals are removed from the locals stack.

Execution proceeds to the second `execute`, which calls the inner trampoline. This pushes the three locals on the locals stack, then performs the code of the inner quotation. It accesses `a` directly on the locals stack, and `b` indirectly through the locals stack, first for reading, then for writing. Finally it builds a `:noname` word and compiles (among other things) `b` and `c`, producing the code shown above. Now, right before the end of the inner quotation we have the state that Fig. 6 shows. Upon finishing the middle quotation, its locals are removed from the locals stack.

Execution proceeds by printing `b` which is accessed indirectly through the locals stack. Then the `:noname` definition is executed, which again prints `b`, accessing it through its compiled home address, and it prints the value of `c` which is directly compiled in.

3.6 Variants

One obvious variant is to keep the outer locals in the trampoline instead of copying them to the locals stack. Then we need to manage the pointer to the trampoline, e.g., by storing it in the locals stack, and caching a copy in a register for faster access.

However, given that locals usually don't live long on the locals stack (compared to the time in the trampoline), this saves relatively little memory overall, and it complicates the implementation. It also increases the lifetime of the trampoline.

Extending this idea, one might also think about using a pointer to the next-outer trampoline instead of copying the locals. This results in the classic static link chain. Read-only locals are copied to the first trampoline of a chain that they appear in; otherwise they would not appear in the chain at all, or we would need to create a separate home location for them.

Both variants complicate the implementation, and it is unclear that they give a significant benefit, let alone one justifying this cost.

4 Syntax

Especially in Forth, syntax is a compromise between ease of programming and ease of implementation. In particular, we want to be able to use a single-pass compiler. E.g., we do not want to scan the whole definition in order to determine, e.g., whether a local is read-only or read-write, because we need this information earlier, for compiling every access to the local.

The syntax below leans more towards the ease-of-implementation side, and is intended for gaining experience with the feature. If we find that we want the feature in the long run, and the following syntax is too inconvenient, we can revise it. You find this syntax applied to our usage examples in Section 2.

4.1 Locals definition

The standard locals definition `{:`...`:}` is ok for defining read-only locals, and read/write locals that are not used as outer locals or with `postpone`. A Forth system can initially mark a local defined in a standard locals definition as potentially both read-only and read/write, and then mark it as read-only upon use inside a quotation or with `postpone`, or mark it as read/write when it is used with `to`. And once it is marked as one, the other use is flagged as an error.

For read/write locals that are used as outer locals or with `postpone`, we have to allocate a home location. This is achived with:

```
['] alloc <{: a W! b c | W! d e :}
```

This defines locals `a b c d e`, but allocates a home location for `b` and `d` with the supplied xt (with stack effect `( u -- addr )`). So `b` and `d` can be written to *and* used as outer locals, while `a c e` can be either written to or used as outer local, but not both.

If the system supports defining double and FP locals, the prefixes for read/write locals are `D!` and `F!`, respectively.

After the locals definition, `addr` is left on the stack, so that the memory can be `free`d when appropriate. If that address is not needed (e.g., with `alloth`), the program has to `drop` it explicitly.

4.2 Quotations

For quotations that access outer locals, we use the following syntax:

```
( xt ) <[{: u W! v | W! x y : a b :}
  ... a ... to b ... ;]
```

This starts a quotation, allocating the trampoline with xt `( u -- addr )`. The mention of `a` and `b` explicitly imports these outer locals. That puts them into the trampoline and makes them visible in this quotation; the various attributes (type, writability) are not restated here, they come from the original definition. E.g., `b` must be a read/write local, or the `to b` would be an error. These locals must be visible in the enclosing definition: They must either be defined there, or be imported explicitly from further out.

`U v x y` are newly defined locals, as in a `<{:` definition. `v` and `x` are read/write, and their home location is in the trampoline. So in this case the trampoline must not be deallocated until these locals are no longer used, and until the xt is no longer used, whichever comes later.

So, for this definition, we get a trampoline with the home locations of `v x`, and the value or address of `a b`. When the trampoline's xt is `execute`d, the value/address of `u v x y a b` is pushed on the locals stack, with `u v` initialized from the data stack, and `a b` from the trampoline. On the locals stack, `v x` are pointers to the home locations of these locals, which are in the trampoline.

At least in Gforth, it is possible to do without importing the outer locals explicitly: Whenever a use of an outer local is encountered, the compiler records it as being imported, in all quotations between the definition and the use, and the local is assigned an offset in the locals stack. When a quotation ends, the code for creating the trampoline is compiled, and it puts these implicitly imported locals there. When the trampoline is `execute`d, these locals are pushed on the locals stack, before the newly defined locals. Given that Gforth accesses the locals by using an offset from the locals stack pointer, each outer local encountered gets the next offset.

It is probably also possible to do similar things in other systems, but if not, outer locals can be imported explicitly, as shown above.

4.3 Trampoline address

Some systems (in particular, SwiftForth) do not implement an xt as the start address of the word. Therefore we need a word

```
>addr ( xt -- addr )
```

This word converts the xt of a trampoline to its address, allowing the programmer to `free` the trampoline, if it was `allocate`d.

4.4 `alloca`

In some cases, it would be useful to allocate home locations or trampolines on the locals stack, so that this memory is automatically reclaimed when leaving the definition. In C, `alloca()` provides this feature, inspiring the name for this feature in this paper:

```
alloca ( u -- addr )
```

5 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), 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], 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 *closure coversion* approach (described in a posting[11] by George Neuner), which replicates locals (or their addresses) in order to make the access cheap.

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

[11]`https://compilers.iecc.com/comparch/article/18-03-010`

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

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.

In 2017 the Forth200x committee has accepted a proposal[13] 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[14], 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 a similar effort as implementing the approach presented in this paper, while not providing as much functionality.

6 Conclusion

Locals in standard Forth have a number of restrictions. In this paper we looked at the accesses to outer locals from inside a quotation, and on `postpone`ing locals. We presented a number of examples where lifting the restrictions would allow additional, and sometimes shorter and easier-to-read ways to express the functionality, as well as presenting alternative code that lives with the restrictions.

In these examples, lifting the restrictions provided some benefits. Whether they are worth the implementation effort and complexity is questionable. However, we started with existing Forth code; if there are cases where Forth is not chosen because it misses this feature, it could not show up in our examples. Moreover, there is value in orthogonality, i.e., that you can combine features like locals, `postpone` and quotations.

We also present an implementation approach and corresponding syntax based on explicit memory allocation, closure conversion, and differentiating between read-only and read/write locals. This implementation approach is designed for minimal implementation complexity while offering the full power of outer locals and `postpone`ing locals. Overall, I expect that this implementation takes a few hundred lines of code.

References

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

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

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

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

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

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

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

[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.

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

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

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