Ways to Reduce the Stack Depth

M. Anton Ertl*
TU Wien

Abstract

Having to deal with many different data can lead to problems in Forth: The data stack is the preferred place to store data; on the other hand, dealing with too many data stack items is cumbersome and usually bad style. This paper presents and discusses ways to unburden the data stack; some of them are used widely, others are almost unknown or new.

1 Introduction

The data stack is the primary mechanism for passing data around in Forth. Its advantages include: words that deal only with the stacks are reentrant, i.e., they can be used recursively and in several tasks running at the same time; and straight-line code using the stack can be factored easily (just split any subsequence off into a separate colon definition).

The limitations of the data stack are: It can contain only cell-sized items. And while it may contain many items, accessing more than a few alternatingly requires quite a bit of stack shuffling and is hard to read; idiomatic in Forth usage tries to avoid stack shuffling.

However, some problems inherently have to deal with more than the about three data items that can be managed without too much shuffling.

A commonly-used example problem is drawing a rectangle specified, e.g., by the lower left and upper right point, using line-drawing primitives that take the start point and the end point: If each point is specified by two numbers, the rectangle is represented by four numbers. Moreover, each number is needed after the first line is drawn, so just before the first line primitive we would have the four numbers for the rectangle on the stack, plus the four numbers needed for the line primitive. Many different ways have been suggested for dealing with this example problem and others.

This paper looks at various ways to deal with such problems, and discusses the advantages and disadvantages.

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

2 Grouping data in memory

One approach is to store much of the data in one or few structures in memory, and putting only the address of the structure(s) on the stack. The disadvantage here is that it requires managing the memory for the structures; this includes specifying who is responsible for deallocating the memory.

In our rectangle example, we might represent each point as a structure, both in the input of our rectangle word and in the calls to the line-drawing words, but this would mean that we would have to allocate, fill, and later deallocate at least two such point structures (for the lower right and upper left point of the rectangle), and let the caller deallocate the structures (somewhat contrary to the Forth idiom of consuming stack items):

```
: line-line ( p1 p2 p3 -- )
  \ draw a line between p1 and p2
  \  and one between p2 and p3
  over line line ;

: rect ( ll ur -- )
  over point-x @ over point-y @ make-point
  ( ll ur ul )
  >r 2dup r@ swap line-line r> free-point
  over point-y @ over point-x @ swap make-point
  ( ll ur lr )
  >r 2dup r@ swap line-line r> free-point ;
```

Of course, we could pass in the data to `rect` in a structure and pass it to `line` on the stack, so the memory management would not show up in `rect`, but that would be less instructive, failing to show that memory management overhead occurs.

Given that Forth does not normally have automatic memory management (aka garbage collection), I tend to avoid such solutions where possible.

3 Multiple Stacks

A common way to deal with many stack items is to put some of them on the return stack. The return stack allows no shuffling and only direct access to the top item, and data has to be moved or copied back to the data stack for computations, so this is usually limited to just one or two items. The disadvantage of this strategy is that we lose the nice factoring property of data-stack-only code: there

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth 2011 Conference proceedings, p. 36-41

are some straight-line code sequences in code using the return stack that cannot simply be split of into a separate colon definition.

Floating-point code keeps floating-point numbers on the separate FP stack, so the number of items on the data stack (and the number of items on the FP stack) is usually smaller than in equivalent integer code, and there is usually much less shuffling necessary.

As an example, consider the following word from the integer matrix multiplication benchmark:

```
: innerproduct ( a b -- n )
  \ a points to a column in a matrix
  \ b points to a row    in a matrix
  0 row-size 0 do
    >r over @ over @ * r> + >r
    swap cell+ swap row-byte-size +
    r>
  loop
  >r 2drop r> ;
```

Note that this code already reduces the stack load by passing `row-size` and `row-byte-size` in constants. A floating-point variant of the word looks like:

```
: finnerproduct ( a b -- r )
  0e row-size 0 do
    over f@ over f@ f* f+
    swap float+ swap row-byte-size +
  loop
  2drop ;
```

All the return stack usage went away.

The main disadvantage of the FP stack is the additional implementation cost: managing another memory area for this stack, per task; and having another stack pointer that has to be saved and restored by context switches and in exception handling.

Some people have suggested additional stacks, e.g., an address stack or a string stack. This has not really caught on yet. In addition to the costs mentioned above one often wants to use operations like `-` on addresses and, e.g., string lengths. Keeping these types on separate stacks would require moving these data between the stacks for such operations, which will increase the stack noise in some cases.

4 Locals

Locals offer a way to deal with lots of data. E.g., for our rectangle example a solution using locals would be:

```
: rect {: x1 y1 x2 y2 -- :}
  x1 y1 x1 y2 line
  x1 y2 x2 y2 line
  x2 y2 x2 y1 line
  x2 y1 x1 y1 line ;
```

The result is readable and this approach scales to dealing with many data.

Despite these advantages, using locals has been vilified often by a considerable portion of the Forth community. One disadvantage of this approach is that we lose the nice factoring properties of data-stack-only code, but using the return stack has the same disadvantage without having the same acceptance problems as locals.

A common argument against using locals is that locals discourage proper factoring; they only discourage it in the same way that the return stack does, but maybe the complaint really is that due to the scalability they fail the encourage factoring in the same way that stack-based approaches do; i.e., that locals take away the pressure to factor for less active data at a time, because words that deal with lots of data still remain manageable.

The question then is why we want a more highly factored program. If locals achieve that goal (say, readability) with less factoring, do we need more factoring? If not, maybe we should be aware of and strive for the desired property instead of just avoiding some programming language features.

5 Global/`user` variables

5.1 ... within single definitions

Another related approach is using global variables. As long as you use them inside a single colon definition, the readability and scalability is similar to using locals. And in contrast to locals, in some respects they keep the nice factoring properties of data-stack-only code. The disadvantages are that the result is not reentrant or usable recursively unless special measures are taken.

The most serious problem, though, is: if you make use of the factoring properties, now the data does not just flow through the stack from caller to callee and back, but through an arbitrary set of global variables. This makes the data flow hard to track, and makes programs hard to maintain. If you want to avoid that, you lose the factoring property with globals just as you lose it with locals.

Also, one nice property of normal factors is that they are often useful for other purposes; however, factors involving globals do not have a nice stack-based calling interface, but something more complicated, so they are usually not nice factors.

In conclusion, while globals are in theory less of an obstacle to factoring than locals, it's usually better to avoid this kind of factoring.

If we use `user` variables to make the word reentrant in the presence of multiple tasks or threads, the variables consume space in each task, all the time. With cooperative multi-tasking, we can avoid that as long as the variables live only between task switches (which creates another maintenance problem), but with true concurrency this trick no longer works; and we want to use true concurrency on the increasingly pervasive multi-core CPUs.

5.2 ... across definitions

Global/user variables are sometimes used as additional input or output parameters for words. An example in standard words is `#`, which takes `base` as additional input and produces additional output in the *pictured numeric output buffer*.

This global state complicates the interface, which reduces reusability and causes maintenance problems. An example is trying to debug code between `<#` and `#>` using `.s`.

One programming practice for reducing these kinds of problems is to save the global variable before changing it and restoring it before returning to the caller. An example of this practice is:

```
: hex.-helper ( u -- )
  hex u. ;

: hex. ( u -- )
  base @ >r
  ['] hex.-helper catch
  r> base ! throw ;
```

However, this practice is somewhat cumbersome to program and Charles Moore prefers to just set global state whenever that is needed [Bro04, Page 212]; this is a bad idea for reusability, but Moore does not value reusability of code.

6 Context wrappers

Saving, changing, and restoring a global variable can be factored out into a context wrapper. E.g., Gforth has a word `base-execute` that saves `base`, changes it, executes an xt, then restores `base`. A usage example is:

```
: hex. ( u -- )
  ['] u. $10 base-execute ;
\ base-execute ( xt u -- )
```

Another example is `execute-parsing`, which saves the input stream, sets it to the passed-in string, executes an xt, and restores the input stream. A usage example is `$create`, which takes the name of the created word from a string instead of the input stream:

```
: $create ( c-addr u -- )
  ['] create execute-parsing ;
\ execute-parsing ( c-addr u xt -- )
```

A third example of this pattern is `>string-execute` which redirects the console output (e.g., `type`) into a string. This allows the programmer to construct strings from many or complicated words without having to deal with intermediate strings on the stack.

```
: fe.>string ( r -- c-addr u )
  ['] fe. >string-execute ;
\ >string-execute ( xt -- c-addr u )
```

The general convention used in Gforth (with the exception of `base-execute`, which came before the convention) is to pass the execution token into the context wrapper on the top of stack, because it is usually a literal.

An advantage of context wrappers is that they make it possible to use words that would otherwise be specific to some global resource (e.g., the console output in case of `fe.`) in a more general way; without a word like `>string-execute`, if you want to transform an FP number into engineering notation, you have to reimplement most of `fe.` yourself.

So context wrappers do not only make it possible to reduce stack shuffling in new code, but also to reuse some code in ways for which it has not originally been written.

Gforth also has the words `infile-execute` and `outfile-execute` that allow to use console input (`key`) or output (`type`) words for input from or output to a file. A special advantage of using a context wrapper here is that it restores the old, working setting if an error is thrown; in contrast, if an error occurs during a global redirection of console I/O, the user has problems recovering from the error (especially if input is redirected).

The usual implementation of context uses global/user variables and saves the contents of these variables on the return stack when performing a context wrapper. The disadvantage of this approach is that each context requires space for another user variable in each task.

Hanson and Proebsting [HP01] discuss a related concept: dynamically scoped variables; they also present several implementation techniques that may require less memory (but more run-time) than approach of using global/user variables with saving.

7 Implicit Parameters and Results

A common pattern in reducing the number of items on the data stack is implicit parameters and return values. The context in context wrappers and global/user state like `base` are two examples of this pattern.

Another one is the loop control parameters in `do` loops. The equivalent `begin` loops would often have too many items on the stack to manage easily.

Finally, a number of object-oriented Forth extensions have an implicit *current object*, e.g., `this` in `objects.fs` [Ert97]; in this objects extension, the current object is set automatically from the top-of-stack when entering a method and the old current object is restored when leaving the method. By contrast, in Bernd Paysan's oof.fs model, the current object is set explicitly, but is then used implicitly whenever calling a method. In both objects extensions, the current object is used implicitly when accessing fields of the current object.

8 Registers

ColorForth has the programmer-visible register A, which is used for memory accesses (e.g., Forth's `!` becomes ColorForth's `A! !`); moreover, the top of the return stack R serves a similar function. Virtual machine models with even more registers have been proposed [Pel08].

A value in A does not have to be kept on the stack, reducing stack load. This is supported by `@+` `!+`, a fetch and store that autoincrement the address in A.

These registers are global resources and share many of the disadvantages of globals. However, their usage model is somewhat different from ordinary globals:

- Most globals are specific for one particular purpose, whereas any word that accesses memory will set A in ColorForth. So, the usage of registers is much more temporary, and programmers typically don't expect the contents to survive across calls (unlike, usually, globals). So, they don't reduce the stack load across calls.

- Interrupt handlers and task switchers will preserve the contents of the registers across the interrupt or for the next execution of the task, so the registers can be used in reentrant code.

9 Example: Postscript Graphics Model

The Postscript graphics model demonstrates some of the ideas presented up to now in action. Here is the rectangle example, written in Forth, but with Postscript graphics operators as words:

```
: rectangle ( x y w h -- )
  2swap moveto
  over 0 rlineto
  0 swap rlineto
  negate 0 rlineto
  closepath stroke ;
```

First, we have adapted the parameters of `rectangle` (width and height instead of the coordinates of the other corner), because that requires less stack shuffling in combination with the Postscript graphics operators. Now, to the essential parts:

Postscript has the *current point* as implicit parameter. We start out by setting the current point with `moveto`.

Then we draw the first line with `rlineto`; the current point determines the start of the line, and the basis for our relative operation[1], so we change x by w and y by 0; `rlineto` also sets the current point to the end point of the line.

The next `rlineto` draws the second, vertical line of our rectangle, the third `rlineto` draws the third line.

Actually, these words did not draw lines, they created a path in the (implicit) graphics state (which contains the current point and other information). Now we add a final line to the path with `closepath` that goes back to the start of the path.

Finally, `stroke` draws the lines described by the path onto the canvas (the in-memory representation of the page). It takes a number of additional implicit parameters into account: line width, colour, dash patterns, corner shape, scale and rotation (more generally, a transformation matrix).

As we can see, Postscript makes effective use of implicit parameters through the global graphics state. To avoid some of the problems of global state, Postscript provides `gsave` to save the current graphics state on a dedicated graphics state stack, and `grestore` to change it back to the old value.

10 Staged Execution

Another way to reduce the number of items on the stack at any one time is to divide the computation into different stages. The first stage deals with some

[1] In addition to the relative `rlineto`, which takes a coordinate relative to the current point, Postscript also has `lineto`, which draws a line to an absolute coordinate.

of the data and generates code for the second stage, the second stage deals with more data, and either completes the computation or generates code for a further stage, etc.

As an example, consider `innerproduct` from the matrix multiply benchmark. The single-stage version (already shown in Section 3) looks like this:

```
: innerproduct ( a b -- n )
  0 row-size 0 do
    >r over @ over @ * r> + >r
    swap cell+ swap row-byte-size +
    r>
  loop
  >r 2drop r> ;

a b innerproduct .
```

This version already uses a number of techniques to reduce the stack depth: a `do` loop to get rid of the stack items for loop control; the number of elements in the vectors (`row-size`), and the strides (cell and `row-byte-size`) are not passed in through the stack; and the return stack is used for the intermediate result.

Here is a version that divides the execution into two stages:

```
: gen-innerproduct ( a[row][*] -- xt )
\ xt is of type ( b[*][column] -- n )
 >r :noname r>
 0 ]] literal SWAP
 [[ row-size 0 do ~~ ]]
   dup @
   [[ dup @ ]] literal * under+ cell+
   [[ row-byte-size + loop
 drop ]] drop ;
 [[ ;

a gen-innerproduct b swap execute .
```

This code uses the syntax `]] x y [[`, which is equivalent to `postpone x postpone y`, but more readable. The staged code uses the same stack depth reduction techniques (except the return stack, which becomes unnecessary) as the original code, but it adds staged execution; this results in less stack shuffling, and no need to use the return stack (except to get the parameter a past the `:noname`. The second stage then contains an unrolled loop that contains the values from the vector a; in source code it would look like this (for a three-element vector a containing 5, -3, 2):

```
:noname
  0 swap
  dup @  5 * under+ cell+
  dup @ -3 * under+ cell+
  dup @  2 * under+ cell+
  drop ;
```

Of course, this technique incurs the CPU and memory costs of generating the code for the second stage, and is normally only efficient if the second stage is used several times (if it is used often enough, it can be significantly more efficient than the single-stage code [LL96], if the Forth compiler supports fast code generation); and the programmer may have to deal with recovering the memory for the generated code.

So, this technique is not very general-purpose, but it still is an interesting addition to the arsenal of stack depth reduction techniques.

11 Pipelines

An program can be organized as multiple tasks that are connected in a pipeline. One reason for this organization is that it allows the flexible composition of useful reusable parts; that is the main reason for using pipelines in Unix. Another benefit of pipelines is that the tasks of a pipeline (pipeline stages) can be executed in parallel.

In the context of our topic the benefit is that each task has its own stack; if we have multiple parameters to pass in and multiple data to handle, hopefully each task needs only a part of these parameters and only needs to deal with a part of the data, reducing the stack depth pressure in that task, compared to a program that tries to do it all in a single task.

I am not aware of an implementation of this idea, but it should not be hard to implement on a multitasking Forth system. In any case, the following is just a somewhat elaborate idea, not something you can use as programmer at the moment.

The following code fetches a vector x from memory, multiplies it with an FP number a, and adds the product vector to another vector y in memory. This is a common linear algebra function (called SAXPY, DAXPY, etc. in BLAS, depending on the type). In our pipelined implementation each of these steps (fetching, multiplying, adding) has its own pipeline stage:

```
: v@ ( f-addr nstride ucount -- )
    0 ?do
        over f@ fput
        tuck + loop
    endput 2drop ;

: vf* ( ra -- )
    begin fget? while
            fover f* fput repeat
    fdrop ;
```

```
: v+! ( f-addr nstride -- )
    begin fget? while
            over f@ f+ over f!
            tuck + repeat
    2drop ;

: faxpy ( ra f-addr-x nstride-x
             f-addr-y nstride-y ucount -- )
    rot rot 2>r ['] v@ xxx|
    ['] vf* f|
    2r> v+! ;
```

Here the input parameters are the start address address, stride[2], and size of x, the value of a, and the address and stride of y (the size is the same as for x). So we have five cell-sized parameters and one FP parameter, too many for handling in one function with stack manipulation words only (therefore I present no version without pipelining).

In our pipelined version each pipeline stage only has to handle a few of the parameters, and consequently there is little stack manipulation code. The interface word `faxpy` sees them all, but only has to pass them to the stages, which is relatively simple.

Each pipeline stage passes its FP result with `fput` to the next stage, which receives it with `fget?`.

The connections between the stages are implicit, so we can only have a linear pipeline. Linear pipelines have been good enough for a lot of work in Unix, but one still might want to consider less restricted options (ideally, any data-flow DAG); the main disadvantage would be that we then have to identify which connection an `fput` or `fget?` refers to, and since this identifier would be passed on the data stack, this would increase the stack load. Another disadvantage of data-flow graphs beyond trees is that simple pipeline implementations can lead to deadlocks.

12 Conclusion

In this paper we look at various ways to reduce the stack load. There is no silver bullet, except locals. Yet, using a combination of the other techniques, most of the time it is possible to keep the stack load manageable even if we do not use locals: using the return stack, the counted loop parameters and various implicit parameters present in the Forth system.

The Postscript graphics model shows how a problem that appears hard for stack-based languages can be solved using such techniques.

We also present the more exotic (in Forth) techniques of staged execution and pipelines, which provide additional weapons against high stack item pressure.

[2] The stride parameter allows using the function on vectors that are not consecutive in memory, e.g., a column or diagonal of a matrix.

References

[Bro04] Leo Brodie. *Thinking Forth*. Punchy Publishing, 2004. reprint of the 1984 edition.

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

[HP01] David. R. Hanson and Todd A. Proebsting. Dynamic variables. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 264–273, 2001.

[LL96] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, 1996.

[Pel08] Stephen Pelc. Updating the Forth virtual machine. In *24th EuroForth Conference*, pages 24–30, 2008.