37th EuroForth Conference

September 10-12, 2021

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 37th EuroForth finds us mostly at home, thanks to Covid19, and the conference is being held on the Internet. The two previous EuroForths were held as a remote conference (2020) and in Hamburg, Germany (2019). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there have been two submissions to the refereed track, one of which was accepted (50% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 30 submissions, 21 accepts, 70% acceptance rate. The papers were sent to three program committee members for review, and they all produced reviews. The reviews of all papers are anonymous to the authors: The papers were reviewed and the final decision taken without involving the authors. This year one submission was co-authored by the program chair; Ulrich Hoffmann served as secondary chair and organized the reviewing and the final decision for that paper. I thank the program committee for their service in reviewing the papers. I thank the authors for their papers.

Late papers will be included in the final proceedings (`http://www.euroforth.org/ef20/papers/`).

You can find these proceedings, as well as the individual papers and slides, and links to the presentation videos on `http://www.euroforth.org/ef21/papers/`.

Workshops and social events (yes, even in a remote conference) complement the program. This year's EuroForth is organized by Gerald Wodni.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
Ulrich Hoffmann, FH Wedel University of Applied Sciences (secondary chair)
Matthias Koch, Institute of Quantum Optics, Leibniz University Hannover
Jaanus Pöial, Tallinn University of Technology
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas

Contents

Practical Considerations in a Static Stack Checker

M. Anton Ertl*
TU Wien

Abstract

One difficulty in applying static checking to existing Forth code (rather than accepting only programs written with the checker in mind) is how to deal with words with statically unknown stack effects, such as `execute`. The work described in this paper introduces the concept of an *anchor* to represent the basis of the stack depth for a position in the code. A new anchor is introduced after a word with an unknown stack effect. Two anchors are synchronized (if still unrelated) on control-flow joins (e.g., `then`), without reporting a stack imbalance (which would probably be a false positive). For previously synchronized anchors, such control flow words can compare the stack depth and report a stack imbalance (probably a mistake) if they do not match. The introduction of anchors also allows to perform the analysis in a single pass.

1 Introduction

Static type checking for Forth has been the subject of research for a long time (see Section 7), but has not resulted in type checkers usable for mainstream Forth. Among the reasons for that are:

- In a statically checked language one typically wants to report all programs that may be erroneous and designs the language and type system for that. E.g., PAF [Ert13] (where the stack depth must be statically known) replaces `execute` with the statically checkable `exec.`*tag*.

  By contrast, for checking programs that incorporate significant parts that were developed (and debugged) without checker, the checker should report no or very few violations for the presumably correct legacy parts (false positives), at the potential cost of more false negatives.

- The stack effect comments in existing programs are not quite standardized enough to allow automatic processing, so a type checker cannot check against them, and also cannot use them to fill in holes in stack effect knowledge (e.g., for deferred words).

- It's hard to specify a type system that is practically usable for mainstream Forth [Ert17b].

In the present paper I have chosen to bypass the type system problem by implementing only stack depth checking. I also bypass the stack-effect comment problem by not making use of them. In this paper I explore how to implement a static stack-depth checker for mainstream Forth, and describe the various design decisions along the way.

It treats statically unknown stack effects as blanks to be filled in rather than as errors, reducing the number of false positives.

The main idea is the introduction of *anchors*. An anchor represents a base stack depth for a part of a definition. Compiling a word with a statically unknown stack effect introduces a new anchor; control flow connecting previously disconnected anchors results in synchronizing them, while control flow connecting already-synchronized anchors allows checking.

Section 2 shows a simple example of stack-depth checking. Section 3 discusses how checking can deal with the various time levels in Forth (interpretation, compilation, `postpone`): We decide to check the run-time during compilation, and don't try to do other checking. Section 4 discusses stack-depth checking at a conceptual level, while Section 5 discusses implementation issues, in particular, how to perform the checking in a single pass.

2 Example

This section gives an example of how stack checking could work. Consider the definition:

```
: min ( n1 n2 -- n )
  2dup < if drop else nip then ;
```

This definition contains a stack effect comment. For stack depth checking the relevant information from this comment is that on exit from this definition, the stack depth is one item less than on entry ($s = a-1$), and that the deepest stack item accessed is two items below the entry depth ($d = a - 2$), where a (the anchor) is the depth on entry. Overall:

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

-1/-2. We show the stack effect of words without anchor, and the intermediate results with anchor.

We also know the stack depth effects of the contained words:

word	s	d
`2dup`	$+2$	-2
`<`	-1	-2
`if`	-1	-1
`drop`	-1	-1
`nip`	-1	-2

Let's determine the overall stack depth effect of `min`. We start before the first word, with the stack effect from the start to this place being $a+0/a+0$.

Next we want to combine this stack depth effect s_1/d_1 with the stack effect s_2/d_2 of the first word `2dup`: The combined stack effect is $s_1 + s_2/\min(d_1, s_1 + d_2) = a + 2/a - 2$.

Using the same computation for the next words results in the following stack effects:

sequence	s	d
`2dup`	$a+2$	$a-2$
`2dup <`	$a+1$	$a-2$
`2dup < if`	$a+0$	$a-2$
`2dup < if drop`	$a-1$	$a-2$

After the `else` control flow continues from after the `if`:

sequence	s	d
`2dup < if nip`	$a-1$	$a-2$

The two control flows join at `then`. The s values of the two control flows have to agree, otherwise we will see a statically unknown stack depth (Section 4.4). The minimum of the joining d values is the resulting d value. This leads to $a-1/a-2$ after the `then` and thus at the end of `min`; after removing the anchor we get $-1/-2$.

We can compare this result of static analysis s_a/d_a with the stack depth effect s_c/d_c described by the programmer in the stack effect comment. We check that $s_a = s_c$ and $d_a = d_c$.[1] In the present example this works out.

3 Time levels

3.1 Immediate

In Forth we have immediate stack effects, e.g., when text-interpreting `+` in interpretation state. These stack effects are not interesting for our checker, for two reasons:

- There is usually little information telling us what stack depth the programmer intended.
- Where there is such information, Forth systems tend to check already, using run-time checking: No stack underflow should happen. And the stack depth at the end of a colon definition is the same as at the start.

If more checking is desired, it's easy to add run-time checking:

```
: expect-depth ( u -- )
  depth 1- <> if
    .s true abort" unexpected stack depth"
  then ;

\ usage example:
0 expect-depth
```

I am unaware that Forth programmers use this kind of checking, so maybe the reason checkers are not used more is not related to the easyness or difficulty of designing and implementing them.

3.2 Compilation

When compiling a word, it has a run-time stack effect in addition to its immediate (i.e., compile-time) stack effect. E.g., when compiling `if`, the run-time stack effect is `( f -- )`, while the immediate (compile-time) stack effect is `( -- orig )`.

The primary interest of static stack depth checking is to check whether a colon definition behaves at run-time as intended (with respect to stack depth). In this case, we have the stack effect comment of a colon definition that tells us the intended stack effect.

3.3 Higher levels

Forth allows to write words that compile code, using `postpone`, `compile,`, `literal` etc. Such words have three levels of stack effects: Their immediate stack effect, the stack effect when these words are executed, and the stack effect when the code compiled by these words is executed.

E.g., the parser generator Gray[2] contains the following words:

```
: compile-test ( set -- )
 postpone literal
 test-vector @ compile, ;

: generate-alternative1 ( -- )
 operand1 get-first compile-test
 postpone if
 operand1 generate
 postpone else
 operand2 generate
 postpone endif ;
```

The use of `compile-test` in `generate-alternative1` has the immediate

[1] Or $d_a \le d_c$ to allow having a stack effect comment that reflects the intended interface rather than the implementation.

[2] http://www.complang.tuwien.ac.at/forth/gray.zip

stack effect `( -- )` (compiling it does not change the stack), the stack effect `( set -- )` when `generate-alternative1` runs, and the stack effect `( -- )` when the code generated by running `compile-test` runs. Of these stack effects, only one is documented (and I have seen this also for cases where an undocumented stack effect other than `( -- )` exists).

It is possible to `postpone` a word like `compile-test`, leading to an additional time level with its stack effect. While I don't remember seeing such code in the wild, it still has to be taken into account.

3.4 Checking at which level?

One approach is be to check at all levels (in particular, including more postponed time levels). If possible, the advantage would be that stack mistakes in code involving `postpone` could be pointed out right at the source code level. The difficulty here is that you often have nothing to check against.

A alternative approach is to only check the run-time stack effect during compilation. There you can compare on control-flow joins (which are more frequent than at other levels), and optionally compare with the stack-effect comment (which typically documents the run-time stack effect of a colon definition, but rarely the other levels). In the present paper we only check at this level, and only whether the stack effects agree on control flow joins.

4 Principles

This section outlines general principles of stack-depth checking, without discussing implementation issues.

In order to perform stack depth checking of a colon definition, we need the stack depth effect of the constituent words, and we need something to compare against: we can compare with the stack effect comment, but we can also compare with the result of stack depth checking of other paths on control-flow joins.

4.1 Straight-line code

As outlined above, if we have a straight-line sequence S consisting of two subsequences S_1S_2, we can compute s/d for S with the following rules:

$$\begin{aligned} s &= s_1 + s_2 \\ d &= \min(d_1, s_1 + d_2) \end{aligned}$$

where s_1/d_1 is the effect of S_1, and s_2/d_2 the effect of S_2.

4.2 Control-flow

In this section we discuss the conceptual treatment of control-flow. In the Section 5.2 we discuss practical considerations.

On unconditional branches (`ahead`, `again`), the stack depth computation follows the control flow.

On conditional branches (`if`, `until`), first the stack effect $-1/-1$ of the word itself is appended to the previous sequence. Then the the stack depth computation follows both directions. I.e., for the fall-through path it works as for straight-line code, whereas for the taken branch it works like for the unconditional branch.

On control-flow joins (`then`, `begin`), the current stack depths of the two joining control flows have to be equal (otherwise the stack depth checker should report a stack depth mistake). The deepest stack depth is the deeper of the two joining stack depths.

A `then` or `begin` at a place that is sequentially unreachable (e.g., in `ahead [ 1 cs-roll ] then`[3]) is not a control-flow join; it only continues the control flow on the other path.

While the present section treats control flow as if the direction of control-flow edges was important, we see in Section 5.4 that the checker can follow control-flow edges in any direction (e.g., always downwards).

4.3 Statically unknown stack effects

For some words the static stack effect is unknown, either because of incomplete knowledge, or because the word can have an arbitrary stack effect at run-time, e.g., `execute`. A stack checker that is intended to work for existing Forth programs has to deal with the occurence of such words. In order to avoid false alarms, it has to assume that the actual stack effect of the word with the unknown stack effect is such that the stack effect is balanced, if possible.

Assuming a checker that processes words left-to-right top-to-bottom, we can achieve this by having a new anchor for the stack depth after the unkown-effect word. If there is a control-flow join with code that uses the old anchor, the anchors can by synchronized. E.g. for

```
if execute over else 2drop then
```

the stack effects of the subsequences are:

sequence	s	d
`if`	$a-1$	$a-1$
`if execute`	$b+0$	$b+0$
`if execute over`	$b+1$	$b-2$
`if 2drop`	$a-3$	$a-3$

When processing the `then`, the two anchors can be synchronized to avoid a stack-depth mismatch:

[3] `Else` does this internally

$b = a - 4$. As a result the overall stack effect of this sequence is $-3/-6$.

This approach allows reporting stack-depth errors in known-depth islands isolated from the rest by words with unknown stack effects, e.g.:

```
execute if drop then execute
```

In this example both control flows at the `then` use the same anchor, and the checker can notice and report the depth mismatch.

4.4 Matching and Synchronization

The rest of this paper repeatedly uses terms like matching control flows or synchronizing anchors. This always refers to the same basic operation, which happens when two control flows meet in one place, e.g., a `then`.

If the two control flows have the same anchor or anchors that have been (transitively) synchronized already, we have to compare the stack depths relative to these anchors; e.g., if $b = a+2$ and the stack depth is $s_1 = b+1$ at one control flow and $s_2 = a+3$ at the other, then the stack depths match (because $s_1 = b+1 = a+2+1 = a+3 = s_2$). If they do not match, the checker should warn of a stack depth imbalance, and the currently-defined word should probably be marked as having an unknown stack effect to avoid getting warnings in places where the word is called.

If the two control flows have anchors that have not been synchronized yet, they are synchronized based on the assumption that the two control flows match (we want to avoid reporting false positives). E.g., if a and b are not already synchronized, and we have $s_1 = b+1$ and $s_2 = a+3$, then we synchronize a and b by setting $s_1 = s_2$, i.e., $b+1 = a+3$, i.e., $b = a+2$.

4.5 Multiple Stacks

In the rest of this paper, I write only about the data stack, but we can do the same static checking for the floating-point and return stack as well (and more, if a system has more, e.g., a vector stack [Ert17a]), with the same principles, and appropriately extended data structures.

5 Implementation issues

5.1 Deepest stack access

The deepest stack access d is only used for checks against stack effect comments. However, for existing code there is probably too much variety in stack effect notation to make such checks practical. Nevertheless, I include the maximum depth in the following discussions; it can be useful for code written to a stack comment standard.

5.2 Single-pass implementation

For implementation simplicity, we want to process the words of a colon definition in a single pass from the first to the last word, without requiring to build a control-flow graph and, e.g., performing an iterative analysis until a fixed point is reached [ASU86]. Can we do this for stack-depth checking? Fortunately, we can:

Deepest stack access

The access depth $s_1 + d_2$ at any particular place has no influence on other access depths, so we can just compute the minimum (in our formulation) of all access depths, without needing to track the deepest stack item through control flow.

The existence of multiple anchors is a complication: Access depths are relative to their anchors. So we compute the deepest stack item relative to each anchor. When an anchor is (possibly transitively) synchronized with the word-entry anchor, we can incorporate knowledge about its deepest stack access into the knowledge about the deepest stack access for the word. If there are anchors left at the end that are not synchronized with the word-entry anchor, we are out of luck and cannot guarantee that the deepest stack access for the word-entry anchor is the deepest stack access for the word. This is due to the unknown stack effects and cannot be solved with more sophisticated analysis unless this analysis makes the stack effects less unknown.

Stack depth change

By contrast, computing the current stack depth requires dealing with control flow, not just with the anchors. Fortunately, the direction of a control-flow edge does not play a role: We just want to match the current stack depths at one end of a control-flow edge with that at the other end, and the direction does not play a role. So for a backwards edge (represented by a dest or do-sys) the word pushing the dest/do-sys can put the anchor and current depth in the dest/do-sys; and the word consuming the dest/do-sys them performs the match. Likewise for origs.

This allows us to do the analysis in a single pass.

5.3 Data structures

This means that we need the following data structures:

For each completed word: s and d.

For each word currently analysed[4], we need a set of anchors, a current anchor, a current stack depth relative to the anchor, and an exit anchor and stack depth.

The set of anchors of a word is partitioned into subsets; each anchor starts as a singleton subset, and synchronization unites the subsets of the involved anchors. One way to implement this is as a parent-pointer tree: each anchor may point to a parent anchor, and the common ancestor represents the subset. We also need to store the current-depth difference of an anchor a to its parent b in a. When trying to match depths, follow each anchor to its root and compute the sum of the current-depth differences along this path.

For each orig, dest, or do-sys, we need to store an anchor and a current stack depth relative to that anchor. We also need this information for every `leave`.

In a given Forth system, we can extend existing data structures such as headers and control-flow stack items with these data. This requires changes to core data structures, which has certain costs.

Alternatively, we can keep these data separate. E.g., a separate lookup table xt$\rightarrow$ s, d, an additional control-flow stack, and an additional stack for storing one definition's incomplete anchor and depth data while processing a quotation. This approach is more complex (mainly thanks to memory allocation), but has the advantage of being easier to work as an add-on.

Gforth has used three-cell control-flow stack items for a long time [Ert94]; for the stack checker the control-flow stack items grew a fourth cell, which (if the checker is active) points to a larger anchored stack effect structure that resides in a separate section [Ert16]. Because of the size of the control-flow stack items, Gforth has already employed a separate `leave` stack, and it continues to do so.

The current stack checker stores the stack effect for a colon definition or primitive as `created` word in a `table` (case-sensitive wordlist), using the xt of the colon definition as the name. The entries for colon definitions are in a separate section to avoid any interference with the ordinary memory allocation.

5.4 Control-flow words

This section explains how the control-flow words work with the data structures.

First, let's consider the effect on straight-line code:

For unconditional control-flow words (`ahead`, `again`, `exit`, and non-zero `throw`), the following code is unreachable. One way of dealing with this is to mark the following code as unreachable until a control-flow join (`begin`, `then`) is compiled, but that needs a special handling of unreachable code in control-flow words. A simpler way is to introduce a new anchor right after the unconditional control-flow; if there is ever a join with a control flow coming from reachable code, the end result is the same.

For the other control-flow words (`if`, `then`, `begin`, `until`, `?do`, `do`, `loop`, `+loop`), control flow can flow from before to after the word, so the anchor is the same after the word as before, and the current depth is changed as indicated by the stack effect of the word.

Concerning the effect on the control-flow stack items:

Words that push control-flow stack items (`if`, `ahead`, `begin`, `?do`, `do`) push the current anchor and the current stack depth (after applying the stack effect of the word).

For words that consume control-flow stack items (`then`, `again`, `until`, `loop`, `+loop`) the checker applies the stack effect of the word, then tries to match the current anchor and stack depth with the anchor and stack depth of the incoming control-flow stack item, as outlined above.

`Exit` can be analysed like an unconditional branch to the end of the definition. We use the exit anchor and stack depth for that: Every `exit` is matched with that. At the end of the definition (`;`, `does>`, or `;code`) we match the current stack depth with the exit stack depth. Then we compare the exit stack depth to the entry stack depth: if the anchors have the same ancestor, we can compute the stack effect of the definition, store it for the definition, and possibly compare it to the stack effect comment.

E.g., if the `min` example was instead written as

```
: min ( n1 n2 -- n )
  2dup < if drop exit then nip ;
```

the checker would work much in the same way as before, with the stack effect at the `exit` being matched against the stack effect at the `;`.

`Leave` is basically an unconditional forward branch like `ahead`, but it does not leave an orig on the control-flow stack. So it's not enough to just enhance control-flow stack items or implement another control-flow stack. One way to deal with this is to store the additional information in a lookup table indexed by the address of the branch (for origs) or branch target (for dests); it may be necessary to distinguish between origs and dests in some other way if they can have the same address.

If an additional control-flow stack is used, `cs-roll` and other control-flow stack manipulation words need to be enhanced to deal with it.

[4]Due to quotations, multiple words can be analysed at the same time.

If you miss `else`, `while`, and `repeat` in this discussion, it's because they are composed from the other words [Bad90].

5.5 Recursion

The simplest way to treat `recurse` (and recursive calls by name) is as a word with unknown stack depth. This way is probably good enough, because recursion is significantly rarer than other sources of unknown stack effects, but it is possible to perform better checking in many cases:

One way to do it would be to check again once the stack effect of the word is known (through the base-case path), but this means using a second pass through the word.

Another way computes the stack-depth change s of the recursive call by looking at the stack depth before and after the recursive call once the anchors involved have been synchronized, and checks if it is equal to the s derived from the base-case path.

Concerning the maximum depth d: In the usual case the maxiumum depth is determined from the base case, but if there is a deeper access in the recursive case, the maximum depth depends on the recursion depth and cannot be determined statically. However, the usual case for recursive calls is to access at most as deeply as the base case, so it may be advisable to produce a warning if the recursive call performs a deeper access.

5.6 ?Dup

We have to deal with `?dup`, because we want to process existing code, and existing code uses `?dup` often enough that it would be a significant source of false postives. E.g., in:

```
?dup if . then
```

the `if . then` is unbalanced and a stack checker that does not take the `?dup` into consideration reports this. But this unbalance rebalances the unbalanced stack effect of the `?dup`, so the whole is balanced, and the stack checker ideally should recognize this.

The common cases of correct `?dup` usage are `?dup` followed possibly by `0=` followed by a conditional branch, and this can be dealt with by setting a flag when encountering `?dup`, modifying it on `0=`, and the conditional branch having appropriately different stack depths on the branch-taken and the fall-through paths (and resetting the flag). If any other word encounters the `?dup` flag, it's probably ok to report a stack-depth mismatch.[5]

[5] I have seen only two cases that do not follow this pattern: one was a bug, and one was a usage of `?dup` at the end of a word, resulting in a word with a `?dup`-like stack effect and usage limitations.

6 Status and Further Work

An early stage of a stack checker for Gforth exists. At the moment it can check sequences and control structures where only a single anchor is involved; it does not even support `else` yet. It checks the data, FP, and return stack. The stack effect of most primitives is known, while the stack effect of pre-defined colon definitions is unknown. The stack effect of successfully checked colon definitions is known.

In the future the checker will be able to also work on code with multiple anchors. In the long run the plan is to also (optionally) use stack effect comments for checking and to use the stack effect comments of pre-defined colon definitions to allow more checks.

Finally, the checker needs to be evaluated by applying it to real-world programs. Because these programs are presumably correct, any mismatches are probably false positives, so this kind of evaluation will tell us the false-positive rate. We will also measure how often we match already-synchronized anchors, giving an idea of the number of actual checks we do perform. And we can compare that to the number of total matches and the proportion of code outside control flow, giving a rough idea what proportion of code is not checked; but note that, e.g., a colon definition without control flow is often still checked if it is used inside control flow in another colon definition.

For checking against stack effect comments, we will probably have to update the stack effect syntax in some Forth programs and can then check against the stack effects specified there.

7 Related work

A simpler way to check the stack depth is to do it at run-time. Hoffmann [Hof91] proposes checking the stack depth on entry to a word and on exit from a word against the stack effect comment. The disadvantage of run-time checking is that one needs to run the word with test cases that cover all the code in the word in order to catch errors. No run-time stack-checking scheme has seen wide usage.

Instead, John Hayes' tester framework has seen wide (although by far not universal) usage. The programmer specifies test cases and expected results and tests not only the stack depth, but also the stack contents. The disadvantage is that the bugs are only reported when the tests are run. Still, Forth programmers are used to catch bugs through testing (including less formalized testing methodologies), which may contribute to the lack of popularity of run-time and compile-time stack-depth and type checking. There are, however, programs dealing with complex data structures where a significant amount of code is necessary for performing testing, and a checker can help to find bugs in that testing code, and find bugs early in the application code.

Reasearchers have been working on compile-time checking for a long time, sometimes as a by-product of other goals:

Tevet [Tev89] uses named data stack items (resulting in a feature similar to locals), and accesses them by compiling `pick` for read accesses and `stick` for writes. In order to do this, his compiler keeps track of the stack depth and reports an error when the compiler cannot determine the stack depth (e.g., because of a stack imbalance at a control-flow join). Tevet's work is close to the present work in limiting itself to stack-depth checking, but differs by requiring a statically known stack depth, while the present work can deal with unknown stack effects, and only reports an imbalance on a statically known imbalance.

Similarly, Ertl requires a statically known stack depth in the Forth dialect PAF [Ert13]; this work does not describe how the stack is checked, and, for now, is only a paper design.

The work that focusses on checking generally also requires complete knowledge of the stack depth in order to work and typically assumes complete knowledge of the stack effects of called words. By contrast, the present work assumes that component words with unknown stack effects are used correctly (to avoid false positives), and only warns in cases where the stack effects derived from words with known stack effects do not agree.

Most of the static checking work has been on type checking, but Hoffmann [Hof93] attacks stack depth checking, the same topic as the present paper; he works out the rules for computing the stack effects of Forth code more explicitly than the present work, but without (explicit) anchors.

On the type checking front, Pöial worked out a stack effect calculus with types in a series of papers [Pöi90, Pöi91, Pöi94] and later described [Pöi02, Pöi06] and implemented [Pöi08] a prototype of a type checker for Forth. This type checker does not deal with unknown stack effects, and the work did not make it out of the prototype stage.

Stoddart and Knaggs [SK91] also work out a typed stack algebra, and also discuss considerations such as `@` and `!`, structured data types, immediate words, and `execute`, but, as usual, assume a total knowledge of the types.

Riegler [Rie15] builds on the work of Pöial, Stoddart and Knaggs, and enhances it with configuration options and pluggable types.

Pfitzenmaier sketches his ideas about type checking Forth [Pfi09], but did not follow it up with an implementation.

In addition to the work on type checking (legacy) Forth, which have not resulted in a widely-used checker, there has also been work on creating new, statically type-checked programming languages, and they have sometimes resulted in usable systems:

StrongForth[6] is a system for a statically type-checked dialect of Forth. It does not accept legacy Forth programs, but requires writing programs to conform with its typing rules.

Factor [PEG10] is a Forth-like high-level language with a mixture of static and dynamic type-checking, so it also solves the problem of static stack-depth checking, but again it prefers to err on the side of overreporting rather than underreporting mistakes.

Kleffner [Kle17] attacks the type checking problem by designing a typed concatenative language (including the `execute`-like `call`) and a static type system for it, but this work has not been followed up with an implementation.

8 Conclusion

A practical stack-depth checker for code that contains significant legacy code cannot rely on stack-effect comments and must produce no or very few false positives, even in the presence of words with statically unknown stack effects. To have something to check against, such a checker can check that the stack effects of two joining control flows agree. It can treat words with statically unknown stack effects as blanks by introducing a new stack-depth anchor when processing such words. The use of anchors is also helpful for performing the checking in a single pass.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986. 5.2

[Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990. 5.4

[Ert94] M. Anton Ertl. Automatic scoping of local variables. In *EuroForth '94 Conference Proceedings*, pages 31–37, Winchester, UK, 1994. 5.3

[Ert13] M. Anton Ertl. PAF: A portable assembly language. In *29th EuroForth Conference*, pages 30–38, 2013. 1, 7

[Ert16] M. Anton Ertl. Sections. In *32nd EuroForth Conference*, pages 55–57, 2016. 5.3

[6] https://www.stephan-becher.de/strongforth/

[Ert17a] M. Anton Ertl. SIMD and vectors. In *33rd EuroForth Conference*, pages 25–36, 2017. 4.5

[Ert17b] M. Anton Ertl. Statische Typüberprüfung. Vortrag bei der Forth-Tagung 2017, 2017. 1

[Hof91] U. Hoffmann. Stack checking - A debugging aid. In *euroFORML '91 Conference Proccedings*, 1991. 7

[Hof93] Ulrich Hoffmann. Static stack effect analysis. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993. 7

[Kle17] Robert Kleffner. A foundation for typed concatenative languages. Master's thesis, Northeastern University, 2017. 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

[Pfi09] Jürgen Pfitzenmaier. Forth type checker. In *25th EuroForth Conference*, pages 60–67, 2009. 7

[Pöi90] Jaanus Pöial. Algebraic specification of stack-effects for Forth programs. In *euroFORML'90 Conference Proceedings*, 1990. 7

[Pöi91] Jaanus Pöial. Multiple stack-effects of Forth-programs. In *euroFORML '91 Conference Proccedings*, 1991. 7

[Pöi94] Jaanus Pöial. Forth and formal language theory. In *EuroForth '94 Conference Proceedings*, pages 47–52, Winchester, UK, 1994. 7

[Pöi02] Jaanus Pöial. Stack effect calculus with typed wildcards, polymorphism and inheritance. In M. Anton Ertl, editor, *18th EuroForth Conference*, page 38, 2002. Abstract in hardcopy proceedings. 7

[Pöi06] Jaanus Pöial. Typing tools for typeless stack languages. In *22nd EuroForth Conference*, pages 40–46, 2006. 7

[Pöi08] Jaanus Pöial. Java framework for static analysis of Forth programs. In *24th EuroForth Conference*, pages 20–24, 2008. 7

[Rie15] Gregor Riegler. Evaluation and implementation of an optional, pluggable type system for Forth. Master's thesis, Technische Universität Wien, 2015. 7

[SK91] Bill Stoddart and Peter J. Knaggs. Type inference in stack based languages. In *euroFORML '91 Conference Proccedings*, 1991. 7

[Tev89] Adin Tevet. Symbolic stack addressing. *Journal of Forth Application and Research*, 5(3):365–379, 1989. 7

Using Test Driven Development to build a new Forth interpreter

Peter Knaggs

January 5, 2023

Abstract

.

1 Introduction

We are describe a method for brining up a new Forth interpreter from scratch using a Test Driven development approach and the John Hayes test suite[1].

The minimum requirements are quite basic:

1. A simple Data stack
2. A simple Dictionary
3. A few native definitions of just three standard words
4. A native implementation of the test harness
5. A simple interpret loop
6. The ability to read lines from a file

To demonstrate just how simple this approach is an initial system, written in under 500 lines of C, is provided in the appendix.

2 Data Stack

The core system requires a relatively simple data stack. In annex A we see a simple array of integers and a stack pointer to index into the array.

We define four methods that work on this array:

push Place a new integer value on the top of the stack and increment the stack pointer. This should check for a stack overflow;

pop Return the top most integer value from the stack and decrement the stack pointer. This should check for a stack underflow;

popLong Return a double number from the top of the stack;

nip Remove the item under the top of stack from the stack.

Note that for simplicity of the example, we use a incrementing stack pointer. This stack implementation is very basic and we would expect it to be changed quite significantly during the process.

[1] ftp://ftp.taygeta.com/pub/Forth/Applications/ANS/core.fr

3 Dictionary

This is a simple linked list of word definitions. Each definition has a simple data structure (XT_t), which contains the name of the word, a pointer to a procedure (ptr_func_t) that does not take any arguments and does not return any value. All communication to the word is via the data stack.

This dictionary is not optimised in any way, it has no knowledge of immediate words, compilation semantics, word lists, or even colon definitions. We assume this rather simple dictionary will be replaced with a more complex data structure during the development, adding the missing features as they are required by the test suite.

3.1 A method of adding a new word to the dictionary

A function to add new native code word to the dictionary. This function is not to be invoked directly from the interpreter but is only intended to initialise the dictionary. It will simply map a word name to a native code function, which it does by adding a XT_t to the linked list that makes up the dictionary. For example:

```
AddWord("TESTING", comment);
```

would associate the word "TESTING" with the C function comment.

3.2 A method of finding a word in the dictionary

A function to step though the linked list, looking to match a dictionary item with a given name. If the name is found, the corresponding data structure (XT_t) is returned otherwise a NULL is returned indicating the name was not found in the dictionary.

4 Echo

When debugging the scanning of the input source, it is useful to echo the text as it is scanned. A special variable *echo* is defined to enable this behaviour.

We define two words to allow the test harness to control the echoing of the input.

+ECHO Turn echoing on, white space and words are written to the console as they are processed.

–ECHO Disable the echo display.

5 Scanning

We provide three methods to scan and process the input:

nextChar Read the next character from the input file. If this detects the end of the line, it will automatically read the next line from the input file. It is also responsible for outputting the character if the system in in *echo* mode. It will return the character or the special value EOF if there is no more text in the file.

$\langle$*next char from input*$\rangle \equiv$
  char ← *line*[*position*]
  increment *position*
  if *char* **is** end of *line* **then**
    line ← read line from file
    position ← 0
    increment *line number*
    if *echo* **then**
      println

print *line number*, ": "
end if
char ← *line*[*position*]
increment *position*
end if

if *echo* **then**
if *char* **is not** white space **or** *char* **is** space **or** *char* **is** tab **then**
print *char*
end if
end if

return *char*
■

nextWord Read the next word from the input, ignoring any leading white space. A word is considered to be any non-white space text. Returns a pointer to the start of the word or NULL if there is no more text in the file.

⟨*next word from input*⟩ ≡
(Ignore leading white space)
char ← **space**
while *char* **is** white space **do**
char ← ⟨*next char from input*⟩
end while

if *char* **is** EOF (end of file) **then**
return null (end of file)
end if

(Read name up to next space)
name ← empty string
while *char* **is not** white space **and** *char* **is not** EOF (end of file) **do**
name ← *name* + *char*
char ← ⟨*next char from input*⟩
end while

return *name*
■

parseNumber
Takes two parameters, the text to parse (as returned by the nextWord) and a pointer to an integer where it will put the resultant number. It will attempt to parse the text as a number (using the base value for the radix). If it can parse the number, it will return *true* and place the number in the integer passed as the second parameter, otherwise it will return *false*.

⟨*parse text as number*⟩ ≡
value ← 0
sign ← 1
position ← 0
min ← **ordinal** 'A'
max ← *min* + *base* - 10;

char ← *text*[*position*]
increment *position*
if *char* **is** '-' **then**
sign ← -1
char ← *text*[*position*]

```
    increment position
  end if

  while char is not end of text do
    char ← upper case (char)
    if char is digit do
      char ← ordinal char - ordinal '0'
    else if ordinal char ≥ min and ordinal char < max then
      char ← ordinal char - min + 10
    else
      return invalid value
    end if
    value ← (value × base) + char
    char ← text[position]
    increment position
  end while

  number ← value × sign
  return valid value
■
```

6 Forth Words

The Hayes test suite uses three normal Forth words without testing them first. As we are defining the test harness as native words, we need to provide native definitions of these words:

HEX (6.2.1660) Set the number radix (base) to 16.
Note that all numbers in the test suite are given in hexadecimal;

\ (6.2.2535) End of line comment – Ignore the rest of the line;

((6.1.0080) In-line comment – ignore all text up to the next `)`.

7 Test harness

There are two different test harnesses to be considered depending on suite of test being use. The original John Hayes test harness uses `{`, `->` and `}`. In the Forth200*x* document Anton Etrl extended the test harness to allow for floating point values, this version uses `T{`, `->` and `}T`.

As we are only going to use the core tests provided by the Hayes suite we do not actually need the floating point extension.

{ Start a test case, we clear the data stack at the start of the test, resetting the data stack depth back to zero.

```
⟨start test case⟩ ≡
  test start depth ← 0
■
```

-> Save test case. This must save the current data stack in a test stack, record the depth of the stack and reset the data stack.

```
⟨save test case⟩ ≡
  test stack ← data stack
  test end depth ← data stack depth
  data stack depth ← 0
■
```

} End a test case. This is the most complex definition as it must compare the current data stack with the one saved in the test stack and report any differences.

> ⟨*end test case*⟩ ≡
> *match* ← (*test end depth* **is** *data stack depth*)
> *n* ← *test start depth*
> **while** *match* **is** true **and** *n* < *test end depth* **do**
> *match* ← *data stack*[*n*] **is** *test stack*[*n*]
> **end while**
>
> **if** *match* is false **then**
> **println** "Stack Mismatch"
> **print** "Found: "
> **for** *n* ← *test start depth* **upto** *test end depth* **do**
> **print** *test stack*[*n*]
> **end for**
>
> **println**
> **print** "Expecting: "
> **for** *n* ← *test start depth* **upto** *data stack depth* **do**
> **print** *data stack*[*n*]
> **end for**
> **println**
> **abort**
> **end if**
>
> *data stack depth* ← *test start depth*
> ■

TESTING Ignore the rest of the line. The harness uses a variable *verbose* to control weather the line is sent to the console or not. We have the *echo* option witch will do the same. You could of course provide an implementation that will send the line to the console even when the *echo* option is disabled.

8 Temporary Definitions

The test suite defines a number of words, both colon definitions and constants, before it has tested these features. Our system is so simple, we can not currently process these definitions therefore, as a temporary measure, we need to comment out these definitions and provide our own (native code) versions.

Once we are past the initial stages of the test suite, it moves on to the defining words. This will require changing the way the dictionary is store, but is also means we can uncomment the definitions and remove our temporary ones, allowing the test suite to operate in the manner originally intended. See section 10 (Procedure) for details.

8.1 Colon Definitions

Thankfully the test suite only defines two helper words in the early stages:

BITSSET? This will test the value on the top of the stack to see it it has a value other than 0. Returning either one 0 or two 0's on the stack:

> ⟨*temporary bitsset definition*⟩ ≡
> *top* ← pop()
> push(0)
> **if** *top* **is** 0 **then**

push(0)
end if
■

BITS Counts the number of bits in the value on the top of the stack:

⟨*temporary bits definition*⟩ ≡
top ← pop()
count ← 0
while *top* **is not** 0 **do**
increment *count*
shift *top* right by 1 bit
end if
push(*count*)
■

8.2 Constant Definitions

Similarly we have to comment out the constant definition, replacing them with our own native code versions. Fortunately most languages provide equivalent constant values so the native code versions are relatively simple:

Constant	Forth Definition	Meaning
0S	0	All bits are zero
1S	0 INVERT	All bits are one
<TRUE>	1S	All bits are one
<FALSE>	0S	All bits are zero
MSB	1S 1 RSHIFT INVERT	most significant bit only
MAX-UINT	0 INVERT	maximum unsigned integer
MAX-INT	1S 1 RSHIFT	maximum signed integer
MIN-INT	1S 1 RSHIFT INVERT	minimum signed integer
MID-UINT	1S 1 RSHIFT	mid-point of unsigned integer
MID-UINT+1	1S 1 RSHIFT INVERT	mid-point of unsigned integer plus one

Once we have implemented and tested the CONSTANT definition we can uncomment these constants and remove the temporary definitions.

8.3 Division

The C language does not define whether division is symmetrical or not. So we need to comment out the definition of IFFLOORED and IFSYM, replacing them with our own native versions that simply ignore the rest of the line. Unfortunately that does mean we also have to provide our own version of the subsequent helper words:

IFFLOORED Ignore rest of line

IFSYM Ignore rest of line

T/MOD Native implementation of /MOD (6.1.0240)

T/ T/MOD NIP

TMOD T/MOD DROP

T*/MOD Native implementation of */MOD (6.1.0110)

T*/ T*/ NIP

Again, once we have tested colon-definitions, we can uncomment the IFFLOORD and IFSYM definitions and remove our temporary definitions.

9 Interpret Loop

Like the dictionary and the stack, the interpret loop is very simple. It has no knowledge of the more advanced features, such as state. These will need to be added as development progresses.

The loop will simply read one name at a time from the input file ⟨*scan next word from input*⟩. It will look the name up in the dictionary ⟨*find name in dictionary*⟩, if the name is found it will execute the associated definition, otherwise it attempts to process the name as a number ⟨*parse name as number*⟩. If it is a valid number, the number is placed on the stack, otherwise a "name not found in dictionary" error is reported.

```
⟨interpret loop⟩ ≡
  begin
    name ← ⟨next word from input⟩
  while name is not null (end of file)
    word ← ⟨find name in dictionary⟩
    if word is not null (name found in dictionary)
      execute word.function (execute word)
    else (word not in dictionary)
      value ← ⟨parse name as number⟩
      if value is valid number then
        push(value)
      else (name not in dictionary or a valid number)
        println
        println "Word not found: ", name
        abort
      end if
    end if
  repeat
■
```

10 Procedure

We are now ready to process the Hayes test suite[2]. Any time the test reports a missing word, the word should be defined and the test suite again. This will allow you to run the following sections of the test suite:

1. Basic Assumptions
2. Booleans: `INVERT AND OR XOR`
3. Shifts: `2* 2/ LSHIFT RSHIFT`
4. Comparisons: `0= = 0< < > U< MIN MAX`
5. Stack operations: `2DROP 2DUP 2OVER 2SWAP ?DUP DEPTH DROP DUP OVER ROT SWAP`
6. Return stack operations: *this has been moved to later in the suite*
7. Add/Subtract: `+ - 1+ 1- ABS NEGATE`
8. Multiplication: `S>D * M* UM*`
9. Division: `FM/MOD SM/REM UM/MOD */ */MOD / /MOD MOD`

The rest of the suite requires fully working versions of the `:` and `CONSTANT` defining words. At this point it would be useful to copy the first 12 tests from section 15 (Defining Words) of the test suite to the top of the test file, allowing basic testing of both words.

[2] ftp://ftp.taygeta.com/pub/Forth/Applications/ANS/core.fr

Once CONSTANT is defined and tested, it should be possible to uncomment the constant definitions and remove the corresponding native code definitions (8.2). Allowing the constants to be defined by the test suite.

Similarly, when : has been defined and tested, it should be possible to uncomment the IFFLOORED and IFSYM definitions and remove the dependent native code definitions (8.3). Unfortunately we can not uncomment the BITSSET? and BITS definitions until after section 13 (Flow control).

10. Memory: HERE , @ ! CELL+ CELLS C, C@ C! CHARS 2@ 2! ALIGN ALIGNED +! ALLOT
11. Characters: CHAR [CHAR] [ ] BL S"
12. Dictionary: ' ['] FIND EXECUTE IMMEDIATE COUNT LITERAL POSTPONE STATE
6. Return stack operations: >R R> R@
13. Flow control: IF ELSE THEN BEGIN WHILE REPEAT UNTIL RECURSE

It should now be possible to remove the two temporary colon-definitions (BITSSET? and BITS) in section 8.1 from our system and allow the test suite to define them.

It should also be noted that we have moved section 9 of the test suite (return stack operations) to just after section 12 (Dictionary).

14. Loops: DO LOOP +LOOP I J UNLOOP LEAVE EXIT
15. Defining Words: : ; CONSTANT VARIABLE CREATE DOES> >BODY
16. Evaluate: EVALUATE
17. Parser input: SOURCE >IN WORD
18. Numbers: <# # #S #> HOLD SIGN BASE >NUMBER HEX DECIMAL
19. Memory movement: FILL MOVE
20. Output: . ." CR EMIT SPACE SPACES TYPE U.
21. Input: ACCEPT
22. Dictionary Search Rules

Having completed the Hayes test suite, most of the CORE word set from the ANS Forth standard have been implemented and tested. We are now ready to move on to using the more advanced testing as presented in the Gerry Jackson test suite[3] and/or the Forth200x standard[4].

11 Experience

The Test Driven Development approach to developing a new interpreter outlined here has been used to to successfully develop two compilers, one in Java and one in C#. An example of the base code necessary to start this process is given in the appendix. This demonstrates a small initial code size of just under 500 lines of C[5] (ignore comments).

A Code

```c
#include <stdlib.h>     /* Standard Library: malloc, free, exit */
#include <stdarg.h>     /* Variable argument processing: va_list, va_start, va_end */
#include <stdio.h>      /* Standard Input/Output: fprintf, vfprintf, stderr, puts, fopen, fclose, fgets, EOF */
#include <string.h>     /* String Library: strdup, strchr, strrchr, strcmp, strcpy, strlen, strcat, memset, memcpy */
```

[3] https://github.com/gerryjackson/forth2012-test-suite
[4] https://forth-standard.org/standard/testsuite
[5] https://www.rigwit.co.uk/forth/baseforth.c

```c
#include <ctype.h>      /* Character Library: isgraph, isspace, toupper, isdigit */
#include <limits.h>     /* Constants: INT_MAX, UINT_MAX */

/* Maximum line buffer length */
#define MAXLINE 1024

/* input file name */
static char* filename = NULL;

/* line number within input file */
static int lineNo;

/* file pointer for current input file */
static FILE* fin;

/* input line buffer */
static char* line = NULL;

/* current scanning position within the input line buffer */
static char* pos;

/* current radix (base) for number conversion */
static int base = 10;

/* ========== Error Handling ========== */

/* Forward reference to the freeDict function to free the memory used by the dictionary. */
void freeDict();

/**
 * @brief Report an error message to the standard error and exit the program.
 * The error message may contain parameter place holders with the additional parameters being provide after the
 * message.  This will display the message on the standard error stream, free any allocated memory and exit the
 * program with the exit code.
 * @param code     the exit code.
 * @param format   the error message to be displayed (may contain parameter descriptions).
 * @param ...      any additional parameters required by the format.
 */
void Error(int code, char* format, ...) {
    if (format) {
        fprintf(stderr, "\n%s(%d): ", filename, lineNo);
        va_list vaargs;
        va_start(vaargs, format);
        vfprintf(stderr, format, vaargs);
        va_end(vaargs);
    }

    freeDict();
    free(filename);
    if (line) { free(line); }
    if (fin) { fclose(fin); }
    exit(code);
}
```

```c
/**
 * @brief Remove the directory name from a file path.
 * This will return a pointer to the first character of the last part of the path (or the start of the path, if the path does
 * not have any directories).
 * @param filename pointer to the start of the file path.
 * @return a pointer to the start of the filename within the path.
 */
char* rmDir(char* filename) {
    char* temp = strrchr(filename, '/');    /* Unix directory separator */
    if (!temp) {
        temp = strrchr(filename, '\\');     /* Dos directory separator */
    }
    return temp == NULL ? filename : ++temp;
}

/**
 * @brief Report the program usage, with an error message and a filename that will be displayed after the error
 * message.  Note this does not free memory so may only be used in the initializations, before the dictionary memory
 * has been allocated.
 * @param progname  the program name, may contain a full path name.
 * @param message    the message to be displayed.
 * @param filename   the filename causing the error.
 */
void usage(char* progname, char* message, char* filename) {
    progname = rmDir(progname);
    char* temp = strchr(progname, '.');
    if (temp) {
        *temp = 0;
    }

    printf("Usage: %s <filename>\n", progname);
    if (filename) {
        printf(message, filename);
    } else {
        puts(message);
    }
    exit(EXIT_FAILURE);
}

/* ========== Data Stack ========== */

#define MAXSTACK 10
int stack[MAXSTACK];    /* Data Stack */
int dsp = 0;            /* Data Stack pointer/depth */

/**
 * @brief Push a single cell item on to the data stack.
 * This will report an error if the stack is full.
 * @param data the item to be placed on the stack.
 * @return the data item placed on the stack.
 */
int push(int data) {
    if ( dsp >= MAXSTACK ) {
        Error(EXIT_FAILURE, "Stack Overflow");
    }
    stack[dsp++] = (int) data;
    return data;
}

/**
 * @brief Remove the item at the top of the stack and return it.
 * This will report an error if the stack is empty.
 * @return the data item at the top of the stack.
 */
int pop() {
    if ( dsp == 0 ) {
        Error(EXIT_FAILURE, "Stack Underflow");
    }
    return (int) stack[--dsp];
}
```

```
/**
 * @brief Remove a double cell item from the top of the stack and return it.
 * @return a double cell item.
 */
long long popLong() {
    long long top = (long long) pop() << (sizeof(long) * 8);
    return top | pop();
}

/**
 * @brief Remove the second item on the data stack.
 * @return the data item removed from the stack.
 */
int nip() {
    int data = stack[dsp];
    stack[--dsp] = data;
    return data;
}

/* ========== Dictionary ========== */

/**
 * @brief A pointer to a function that takes no arguments are does not return a value, i.e., void func().
 */
typedef void (*ptr_func_t)();

/**
 * @brief The Execution Token data structure.
 */
struct XT_s {
    char*           /name;
    ptr_func_t      /func;
    struct XT_s*    /next;
};

/**
 * @brief A pointer to an Execution Token data structure.
 */
typedef struct XT_s* XT_t;

/**
 * @brief The head of the dictionary linked list.
 * A pointer to the most recent XT in the dictionary. Each XT contains a pointer to the next XT in the dictionary
 * with the last XT in the list holding the NULL for the next value.
 */
static XT_t dict = NULL;

/**
 * @brief Free all memory used by the dictionary.
 * Loop though the dictionary, one entry at a time, and free the memory used by the word name and the XT_s
 * data structure itself.
 */
void freeDict() {
    XT_t next = dict;
    while ( dict != NULL ) {
        next = dict->next;
        free(dict->name);
        free(dict);
        dict = next;
    }
}

/**
 * @brief Add a word into the dictionary.
 * This will build a new XT data structure which it will place at the head of the dictionary linked list, placing the
 * current head of the list as the next item in the XT data structure.
 * @param name  word name to add.
 * @param func  pointer to c-function to preform the word's action.
 */
```

```c
void AddWord(const char* name, ptr_func_t func) {
    XT_t xt = (XT_t) malloc(sizeof(struct XT_s));
    xt->func = (ptr_func_t) func;
    xt->name = strdup(name);
    xt->next = dict;
    dict = xt;
}

/**
 * @brief Find a word in the dictionary, returning the word's XT data structures or NULL if the word is not found.
 * This will start at the head of the dictionary and follow the links to each XT in the dictionary until it either finds the
 * XT with the given name or comes to the end of the linked list.
 * @param name the word to search for.
 * @return a pointer to the XT of the word or NULL if not found.
 */
XT_t find(char* name) {
    XT_t current = dict;
    while ( current != NULL ) {
        if ( strcmp(current->name, name) == 0 ) {
            break;
        } else {
            current = current->next;
        }
    }
    return current;
}

/* ========== Echo ========== */

static int echo = 1;

void echoOff() { echo = 0; }
void echoOn()  { echo = 1; }

void initEcho() {
    AddWord("+ECHO", echoOn);
    AddWord("-ECHO", echoOff);
}

/* ========== Scanning ========== */

/**
 * @brief Read the next character from the input file.
 * If the character is the end of line marker, read the next line from the file and return the first character of the new
 * line.  If in echo mode write the character to the console, when reading a new line write the line number to the
 * console.
 * @return the character or EOF if at the end of the file.
 */
char nextChar() {
    char c = *pos++;
    if (c == 0) {
        if (fgets(line, MAXLINE, fin)) {
            pos = line;
            lineNo++;
            if (strlen(line) + 1 == MAXLINE) {
                Error(EXIT_FAILURE, "Line too long for buffer of %d characters", MAXLINE);
            }
            if (echo) {
                printf("\n%4d: ", lineNo);
            }
            c = *pos++;
        } else {
            return EOF;
        }
    }
    if (echo && (isgraph(c) || c == ' ' || c == '\t')) {
        putchar(c);
    }
    return c;
}
```

```c

/**
 * @brief Return the next word from the input.
 * This will ignore any leading white space and return a pointer to the next non white-space character in the input
 * line.  It will replace the first white-space character after the word name with an end of text marker to convert that
 * part of the input line into a string.
 * @return a pointer to the first character of the word.
 */
char* nextWord() {
    /* Ignore leading white space */
    char c = ' ';
    while (isspace(c)) {
        c = nextChar();
    }

    if (c == EOF) {
        return NULL;
    }

    /* Read name up to next space */
    char* name = pos - 1;
    while (!isspace(c) && c != EOF) {
        c = nextChar();
    }

    /* Mark end of word */
    *(pos - 1) = 0;

    return name;
}

/**
 * @brief Attempt to convert text into a number using the current base.
 * This will attempt to convert the string in text into a number using the value is base as the radix.  If successful the
 * number will be placed in the integer pointed to by the number parameter and a true value is returned otherwise a
 * false value is returned.
 * @param text       the text to be parsed.
 * @param number  a pointer to a location where the number can be stored.
 * @return true if the text is a number or false if not.
*/
int parseNumber(char* text, int* number) {
    int value = 0;
    int sign = 1;

    char c = *text++;
    if (c == '-') {
        sign = -1;
        c = *text++;
    }

    while (c > 0) {
        c = toupper(c);
        if (isdigit(c)) {
            c = c - '0';
        } else if (c >= 'A' && c < 'A' + base - 10) {
            c = c - 'A' + 10;
        } else {
            return 0;   /* Not a valid number */
        }
        value *= base;
        value += c;
        c = *text++;
    }

    *number = (value * sign);
    return c == 0;
}
```

```c

/* ========== Forth Words ========== */

/**
 * @brief Set BASE to 16
 */
void hex() { base = 16; }

/**
 * @brief Ignore all text up until the end of the line.
 */
void comment() {
    int len = strlen(pos);
    while (len-- > 0) {
        nextChar();
    }
}

/**
 * @brief In-line comment - ignore all text up until the next ).
 */
void parn() {
    char c;
    do {
        c = nextChar();
    } while (c != EOF && c != ')');
}

/**
 * @brief Add the Forth words HEX, \ and ( to the dictionary.
 */
void initForth() {
    AddWord("HEX",  hex);
    AddWord("\\",   comment);
    AddWord("(",    parn);
}

/* ========== Test Harness ========== */

static int testStack[MAXSTACK]; /* Test stack */
static int tend;

/**
 * @brief Start a test, start with a clean data stack.
 */
void testStart() {
    dsp = 0;
}

/**
 * @brief Save the test stack.
 * Copy the current data stack to the test stack.
 */
void testSave() {
    memset(testStack, 0, sizeof(int) * MAXSTACK);
    memcpy(testStack, stack, sizeof(int) * dsp);
    tend = dsp;
    dsp = 0;
}

/**
 * @brief End a test.
 * Compare the current data stack with the test data stack.
 * Report an error if the two stacks to not match exactly.
 */
```

```c
void testEnd() {
    int match = (tend == dsp);
    for (int n = 0; (match && n < tend); n++) {
        match = (stack[n] == testStack[n]);
    }

    if ( !match ) {
        fprintf(stderr, "\n%s(%d): Stack Mismatch\n", filename, lineNo);
        fprintf(stderr, "Found:      ");
        for (int n = 0; n < tend; n++) {
            fprintf(stderr, "%d ", testStack[n]);
        }

        fprintf(stderr, "\nExpecting: ");
        for (int n = 0; n < dsp; n++) {
            fprintf(stderr, "%d ", stack[n]);
        }

        fprintf(stderr, "\n");
        Error(EXIT_FAILURE, NULL);
    }
    dsp = 0;
}

/* ==== Temporary colon definitions to be removed once : is defined and tested */
/* { : BITSSET? IF 0 0 ELSE 0 THEN ; -> } */
void bittest() {
    int top = pop();
    push(0);
    if (top) {
        push(0);
    }
}

/* : BITS ( x -- u ) 0 SWAP BEGIN DUP WHILE MSB AND IF >R 1+ R> THEN 2* REPEAT DROP ; */
void bits() {
    unsigned int top = pop();
    int count = 0;
    while (top) {
        count++;
        top >>= 1;
    }
    push(count);
}

/* ==== Temporary CONSTANT definitions to be removed once CONSTANT is defined and tested */
void zeros()      { push(0); }                  /* 0                        */
void ones()       { push(~0); }                 /* 0 INVERT                 */
void cfalse()     { push(0); }                  /* 0S                       */
void ctrue()      { push(~0); }                 /* 1S                       */
void msb()        { push(~INT_MAX); }           /* 1S 1 RSHIFT INVERT       */

/* The Comparison operators define a number of constants */
/* MSB, MAX-UINT, MAX-INT, MIN-INT, MID-UINT, MID-UINT+1 */
void maxUInt()    { push(UINT_MAX); }           /* 0 INVERT                 */
void maxInt()     { push(INT_MAX); }            /* 0 INVERT 1 RSHIFT        */
void minInt()     { push(INT_MIN); }            /* 0 INVERT 1 RSHIFT INVERT */
void midUInt()    { push(UINT_MAX >> 1); }      /* 0 INVERT 1 RSHIFT        */
void midUI1()     { push(~(UINT_MAX >> 1)); }   /* 0 INVERT 1 RSHIFT INVERT */

/* C can use either Floored or Symmetric division */
/* The following temporary colon definitions can be removed once : is defined and tested */
int isFloored() {
    return (-3 / 2) == -1;
}
```

```
/* IFFLOORED     : T/MOD  >R S>D R> FM/MOD ; */
/* IFSYM         : T/MOD  >R S>D R> SM/REM ; */
void tdm() {
    int top = pop();
    long long nos = popLong();

    long long div;
    int rem;

    if ( isFloored() ) {
        div = nos / top;
        rem = (int) (nos - (div * top));
    } else {
        div = nos / top;
        rem = (int) (nos % top);
    }

    push(rem);
    push((int) div);
}

/* IFFLOORED     : T/    T/MOD SWAP DROP ; */
/* IFSYM         : T/    T/MOD SWAP DROP ; */
void td() { tdm(); nip(); }

/* IFFLOORED     : TMOD   T/MOD DROP ; */
/* IFSYM         : TMOD   T/MOD DROP ; */
void tm() { tdm(); pop(); }

/* IFFLOORED     : T*_/MOD >R M* R> FM/MOD ; */
/* IFSYM         : T*_/MOD >R M* R> SM/REM ; */
void tsdm() {
    int top = pop();
    long long a = pop();
    long long b = pop();
    long long mul = a * b;

    long long div;
    int rem;

    if ( isFloored() ) {
        div = mul / top;
        rem = (top - (int) div * top);
    } else {
        div = mul / top;
        rem = mul % top;
    }

    push(rem);
    push((int) div);
}

/* IFFLOORED     : T*_/ T*_/MOD SWAP DROP ; */
/* IFSYM         : T*_/ T*_/MOD SWAP DROP ; */
void tsd() { tsdm(); nip(); }

/**
 * @brief Initialise the dictionary with the test harness and temporary definitions.
 */
void initTest() {
    /* Hayes test harness { -> } */
    AddWord("{",                  testStart);
    AddWord("->",                 testSave);
    AddWord("}",                  testEnd);
    /* Forth200x test harness T{ -> }T */
    AddWord("T{",                 testStart);
    AddWord("}T",                 testEnd);
```

```
    AddWord("TESTING",          comment);

    /* Temporary colon definitions (Basic assumptions and Memory) */
    AddWord("BITSSET?",         bittest);
    AddWord("BITS",             bits);

    /* Temporary constant definitions */
    AddWord("0S",               zeros);
    AddWord("1S",               ones);
    AddWord("<TRUE>",           ctrue);
    AddWord("<FALSE>",          cfalse);

    /* Comparison operators use the following constants */
    AddWord("MSB",              msb);
    AddWord("MAX-UINT",         maxUInt);
    AddWord("MAX-INT",          maxInt);
    AddWord("MIN-INT",          minInt);
    AddWord("MID-UINT",         midUInt);
    AddWord("MID-UINT+1",       midUI1);

    /* Temporary colon definitions (Division) */
    AddWord("IFFLOORED",        comment); /* Ignore rest of line */
    AddWord("IFSYM",            comment); /* Ignore rest of line */
    AddWord("T/MOD",            tdm);
    AddWord("T/",               td);
    AddWord("TMOD",             tm);
    AddWord("T*/MOD",           tsdm);
    AddWord("T*/",              tsd);
}

/* ========== Main ========== */

/**
 * @brief Open the file given in as the command line argument.
 * This will process all of the command line arguments, checking that there is only one.  It will attempt to open the
 * file, if not able to it will add the .forth extension to the file and try again, if the file is still not found it will try the
 * .fr extension.  If successful the fin and filename global variables configured, otherwise it will report a usage error
 * and exit.
 * @param argc  the number of arguments contained in the argv array.
 * @param argv  an array of strings, one for each command line argument.
 */
void openFile(int argc, char *argv[]) {
    /* Check we have the right number of arguments */
    if ( argc == 1 ) {
        usage(argv[0], "We␣need␣a␣file␣to␣process␣!", NULL);
    } else if ( argc > 2 ) {
        usage(argv[0], "Can␣only␣process␣one␣file␣at␣a␣time", NULL);
    }
    filename = argv[1];

    /* Process the file name (allow for ".forth" extension) */
    int len = strlen(filename) + 10;
    char* name = (char*) malloc(len * sizeof(char));
    strcpy(name, filename);

    /* Does the file exist? */
    fin = fopen(name, "r");
    if ( !fin ) {
        /* File not found, try ".forth" extension */
        strcat(name, ".forth");
        fin = fopen(name, "r");
    }

    if ( !fin ) {
        /* Still not found, try ".fr" extension */
        strcpy(name, filename);
        strcat(name, ".fr");
        fin = fopen(name, "r");
    }

```

```
    if ( !fin ) {
        /* Still not found, give in */
        usage(argv[0], "Can not open input file: \"%s\"", filename);
    }

    filename = strdup(rmDir(name));
    free(name);
}

/**
 * @brief Initialise the system.
 * First it will attempt to process the command line arguments. It will then initialise the dictionary before initialising
 * the line buffer so that the first call to nextChar will load the first line of the file into the buffer.
 * @param argc  the number of command line arguments in the argv array.
 * @param argv  an array of strings, one for each command line argument.
 */
void init(int argc, char* argv[]) {
    /* Open input file */
    openFile(argc, argv);

    /* Initialise dictionary */
    initEcho();
    initTest();
    initForth();

    /* Initialise line buffer */
    line = (char*)malloc(MAXLINE * sizeof(char));
    pos = line;
    *pos = 0;
    lineNo = 0;
}

/**
 * @brief The main interpret loop.
 * This will read the input file, one word at a time, it will look up each word in the dictionary and if found it will
 * preform the action associated with the word, otherwise it will attempt to convert the word into a number (using the
 * current base).  If successful it will place the number on the data stack otherwise it will report a word not found
 * error and abort.
 * @param argc  the number of command line arguments in the argv array.
 * @param argv  an array of strings, one for each command line argument.
 * @return does not return
 */
int main(int argc, char* argv[]) {
    init(argc, argv);

    /* Interpret loop */
    char* name;
    while (name = nextWord()) {
        XT_t word = find(name);     /* Lookup name in dictionary */
        if (word) {                 /* if name found */
            word->func();           /* Execute XT*/
        } else {                    /* Name not found */
            int value;              /* convert to number */
            if (parseNumber(name, &value)) {
                push(value);        /* Push number onto stack */
            } else {                /* Not a word or a number */
                Error(EXIT_FAILURE, "Word not found: %s", name);
            }
        }
    }
    Error(EXIT_SUCCESS, NULL);
}
```

The Linguistics of Forth

Recently, I introduced Forth to a computer science student that is taking linguistics as subsidiary subject. It became clear to me that linguists should love Forth for two reasons that have substantial collateral benefits:

- Its Stack based nature.
- Its simple Parser.

The Stack advantage

Programming an explicitly Stack based machine is considerably different to the vast majority of existing programming languages. It needs a different way of thinking. In the past we could always refer to the "HP-calculators" to familiarise engineers with Forth. (Note: Unfortunately, HP has given up on RPN. It is still there but only as an obscure operating mode for aged engineers.)

From a linguistic point of view this has dramatic advantages for a programming language.

Because input and output arguments are handled by the Stack, Forth words do not need parameter lists. This changes the programming style substantially, because you can put several words on a single line. This could be called "horizontal programming style".

Conventional programming languages clutter the code with parameter lists, which severely hamper the readability of the code. Usually, only one procedure call followed by its parameter list is put on a single line.

First benefit:

Given Forth's horizontal programming capability we can compose phrases and sentences. And we can put our ambition into writing "readable" code that can be understood by system engineers on its highest levels resulting in more reliable code.

Second benefit:

Horizontal programming puts more code on a single screen. Therefore, you do not have to scroll nearly as often as in other languages. That is a clear debugging advantage.

The Parser advantage

Most of the time, Forth's lexical scanner only looks out for whitespace. As a consequence, any special character may be used to compose a name. This opens up a whole new dimension for the signification of names compared to most other programming languages. Those have a rather limited inventory of "valid" characters that may be used.

Therefore, Forth's source code includes many more significant spaces compared to other languages. This makes Forth code more readable, because "reading" Forth is akin to reading a book.

And because of its simplicity, the Forth lexical scanner can do without regular expressions processing.

I asked myself, why most programming languages are so neglectant of the syntactical role of spaces. This came to my mind:

The first programming languages (FORTRAN, COBOL) appeared at a time when the source code had to be punched into cards. Every single character was costly. Omitting "unnecessary" spaces was used as a simple means for compression. Apparently, more recent programming languages upheld this as a tradition, whose justification had withered away decades ago.

Taming the IoT

Forth's Role in the Internet of Things

EuroForth'21 conference 2021-09

Ulrich Hoffmann

uho@  .de

Overview

- The Internet of Things
- MQTT
- Forth Things
- Demo
- Different Kind of Messages
- Domain Specific Languages
- Conclusion

The Internet of Things

source: Wikimedia

- **embedded Systems**
- **interconnected by Internet technology**
- **+ specialised communication protocols**
  - **MQTT (Message Queuing Telemetry Transport) publish and subscribe via a broker**
  - ROS (robot operating system)
  - zeromq, AMQP, DDS

source: Wikimedia

MQTT

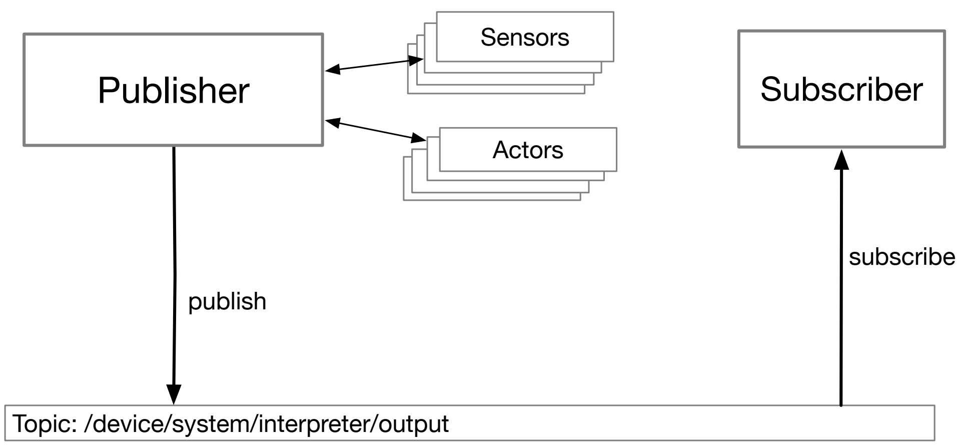
Message Queuing Telemetry Transport

- **lightweigth IoT communication**
- ***publish* and *subscribe* 1:N communication**
- **uses a *broker* (server) usually runs over TCP/IP**
- ***topics* (communication channels)**
  - **a node (thing)**
    - **can publish a message to a topic and**
    - **all subscribers of that topic receive the message**
  - **with hierarchical names such as** `/device/system/interpreter/input`
  - **wild cards in order to subscribe to a set of topics + #**
- quality of service, last will, ...
- wide support by libraries, applications, community
  node red, mqtt explorer, mosquitto broker, ...

source: Wikimedia

MQTT

Message Queuing Telemetry Transport

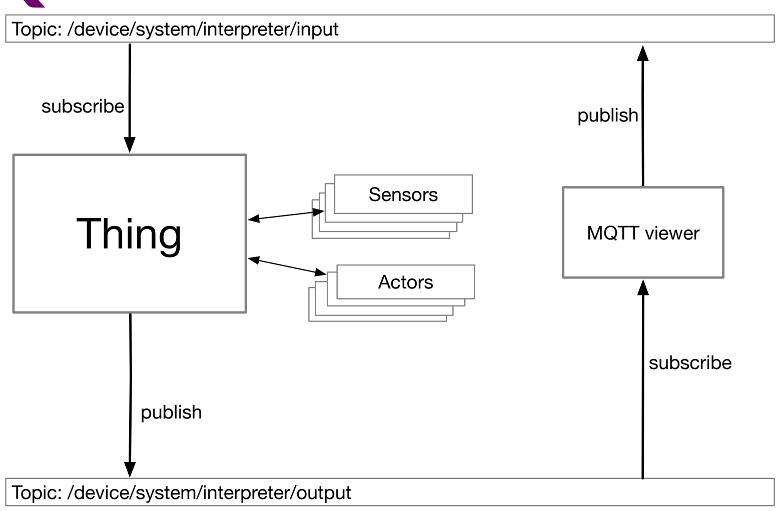


Of course a single thing can be publisher and subscriber at the same time.

source: Wikimedia

MQTT

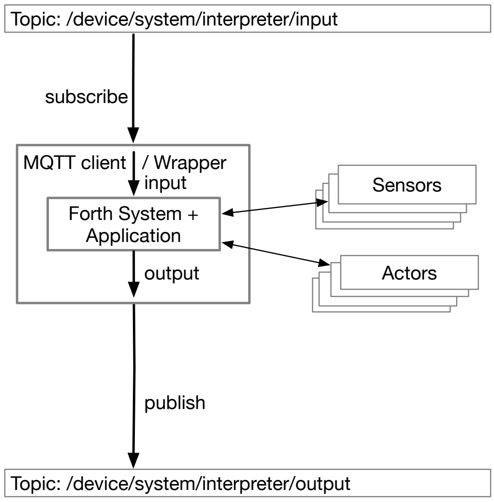
Message Queuing Telemetry Transport



Of course a single thing can be publisher and subscriber at the same time.

Forth things

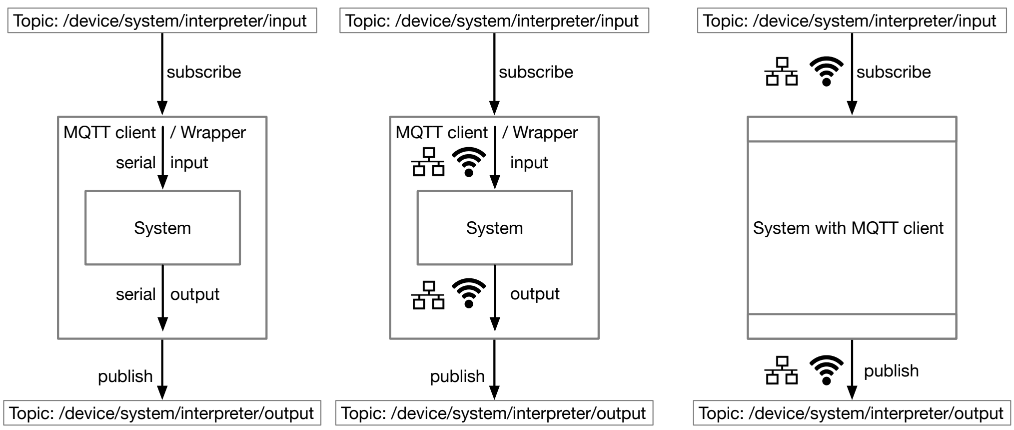
Can we implement things in Forth? Yes



connect Forth's input and output to topics

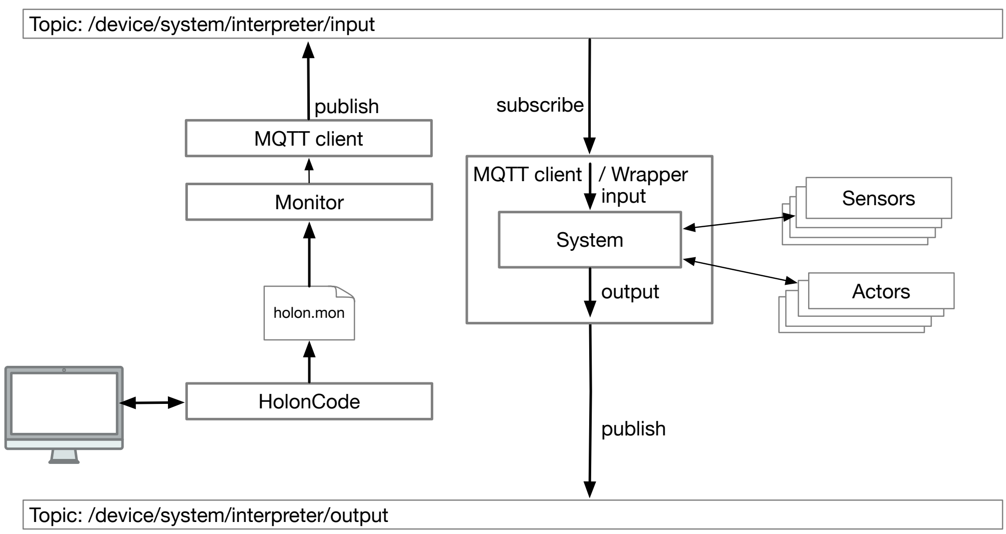
Forth things

Can we implement things in Forth? Yes



connect Forth's input and output to topics

Forth things - Interactive Development



DEMO

- MQTT broker is running
- MQTT explorer is connected to see messages
- seedForth is wrapped so that its
  - input comes from /device/system/seedForth/input
  - output goes to /device/system/seedForth/output
- MQTT explorer can send messages to seedForth
- Command line client `connect` can access seedForth via MQTT

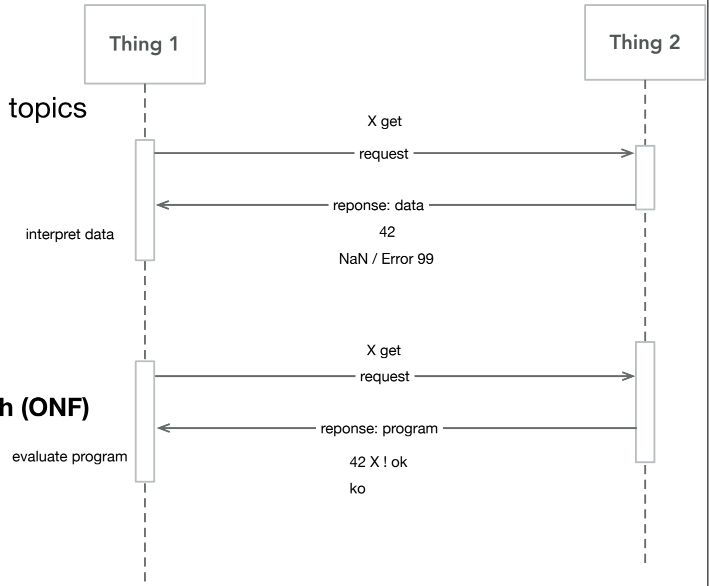
Different kind of messages

- **connect Forth's input and output to topics**

- **Forth's output must be lean. Two words**
  - `verbose` **- make system ready for interactive use**
  - `quiet` **- calm down system to do no echo or superfluous output.**

Different kind of messages

- connect Forth's input and output to topics
- **What messages to exchange?**
  - **requests (commands) and data reponses?**
  - **requests (commands) and program responses?**
- **Heinz Schnitter's Open Network Forth (ONF)**



Domain Specific Languages (DSL)

- "Forth is well suited for DSLs."
- Yes - but how?
  - sealed vocabularies
  - natural langage like syntax
  - best practice for design of Forth DSLs?
  - sandboxes?

Domain Specific Languages (DSL)

Sealed Vocabularies

- **Put all words of your DSL in word lists of their own.**
- **Only search these word lists, i.e. seal these vocabularies**

This might be helpful:

```
: evaluate-in-search-order ( c-addr u i*x wid1 ... widn n -- j*x )
    n>r  get-order
    nr>  set-order
    n>r  ['] evaluate catch
    nr> set-order
    throw ;
```

Domain Specific Languages (DSL)

Natural Language Syntax

- **Design your DSL using different kinds of words**
  - **nouns (-- i*x)**
  - **verbs (i*x --)**
  - **adjectives (i*x -- j*x)**
- **Make your commands phrases with**
  - **subject object1 object2 ... verb**

elbow 30 degrees clockwise turn

See "In Review: FORML 1984 Asilomar Conference", FD, Vol. VI, No. 5, p34ff, 1984

Domain Specific Languages (DSL)

Best practices and sandboxes?

- **Who has a systematic structured aproach to Forth DSLs?**
  - **please contact me ->** 

- **Sandboxing**
  - **We want to evaluate Forth source code.**
  - **How can this be restricted to be save?**
  - **Certainly no unrestricted `@` and `!`** 🙂
  - **Work on best practices to do sandboxes**
  - **Again: please contact me ->** 

Taming the IoT

Forth's Role in the Internet of Things

Conclusion

• The Internet of Things	connected embedded systems
• MQTT	publish and subscribe via broker
• Forth Things	connect input and output to topics
• Demo	we've seen some stuff live
• Different Kind of Messages	data or program responses
• Domain Specific Languages	sealed vocabularies, nouns&verbs, sandboxes
• Conclusion	you are here

Taming the IoT

Forth's Role in the Internet of Things

Conclusion

• The Internet of Things	connected embedded systems
• MQTT	publish and subscribe via broker
• Forth Things	connect input and output to topics
• Demo	we've seen some stuff live
• Different Kind of Messages	data or program responses
• Domain Specific Languages	sealed vocabularies, nouns&verbs, sandboxes
• Conclusion	you are here

Questions?

simulation of the Einstein-Podolsky-Rosen experiment in forth

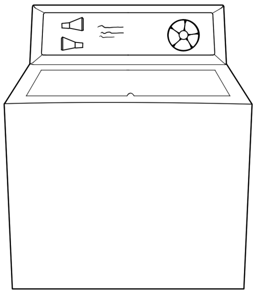
Krishna Myneni

EuroForth 2021

V1.0

back to basics

```
: washer   wash  SPIN  rinse  SPIN  ;
```



Brodie, L. (1987), *Starting Forth*, Prentice-Hall.

- measuring the spin (magnetic moment) of a particle
- simulating spin measurements using forth: `epr-sim`
- quantum theory in a couple of slides
- factoring quantum states and entanglement
- exploring strong correlations in an entangled spin state using `epr-sim`
- EPR argument for incompleteness of QM [using entangled spins]
- exploring hidden variables explanations with `epr-sim`
- correlation coefficient and Bell's inequality for hidden variable theories
- computing Bell's inequality with `epr-sim`
- `epr-sim` design

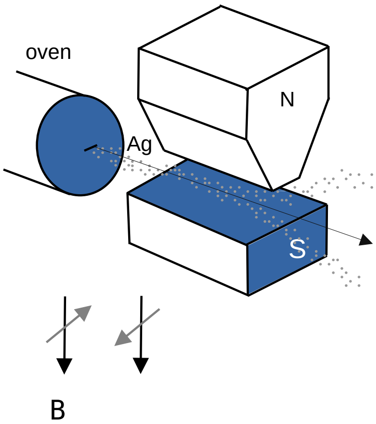
„Wäre es möglich, einen tüchtigen Physiker herbei [nach Frankfurt] zu ziehen, der sich mit dem Chemiker vereinigte und dasjenige heranbrächte, was so manches andere Kapitel der Physik, woran der Chemiker keine Ansprüche macht, enthält und andeutet; setzte man auch diesen in Stand, die zur Versinnlichung des Phänomens nötigen Instrumente anzuschaffen, so wäre in einer großen Stadt für wichtige, insgeheim immer genährte Bedürfnisse und mancher verderblichen Anwendung von Zeit und Kräften eine edlere Richtung gegeben.“

– *Johann Wolfgang Goethe,* 1814: Am Rhein, Main und Neckar.
In: Autobiographische Schriften. Band III, S. 297.

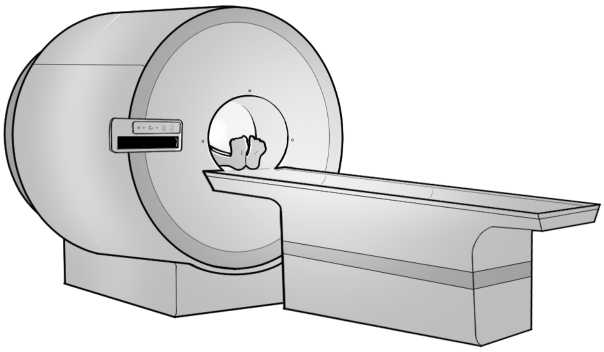
https://de.wikipedia.org/wiki/Physikalischer_Verein
https://www.goethe-university-frankfurt.de/63113635/Physics_of_yesterday

other kinds of spin machines

Stern-Gerlach experiment



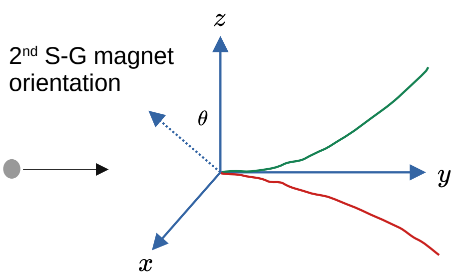
MRI scanner

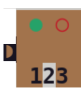


Artwork by Shreya

H. Schmidt-Böcking, et al., arXiv:1609.09311v1 [physics.hist-ph] 29 Sep 2016

single spin-1/2 particle in the “spin-up” quantum state





$P_u(0^\circ) = 1$

$P_u(60^\circ) = ¾$

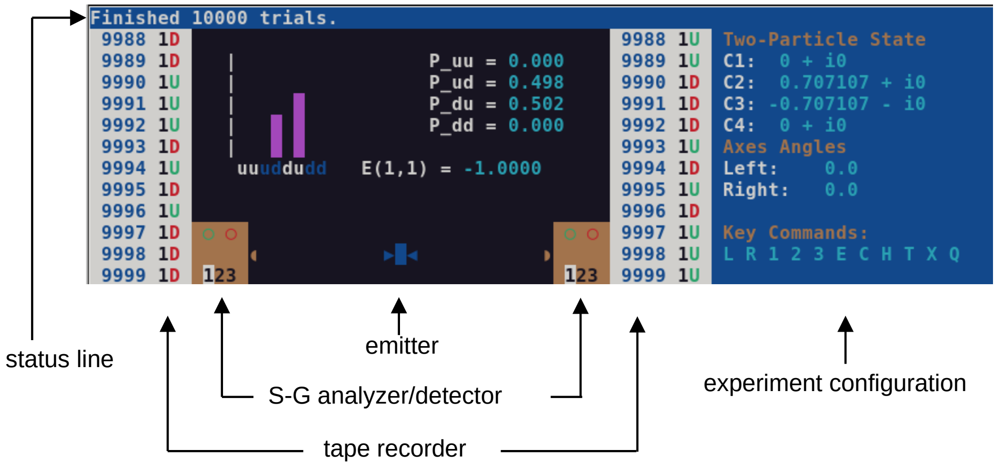
$P_u(120^\circ) = ¼$

Simulation output from `epr-sim` for 0°, 60°, and 120°.

100 1U	100 2U	100 3D
101 1U	101 2D	101 3D
102 1U	102 2D	102 3U
103 1U	103 2D	103 3D
104 1U	104 2U	104 3D
105 1U	105 2U	105 3U
106 1U	106 2U	106 3D
107 1U	107 2U	107 3D
108 1U	108 2U	108 3D
109 1U	109 2U	109 3D
110 1U	110 2D	110 3D
111 1U	111 2U	111 3U

```
Q2p2s new dup
z1 z0 z0 z0 init-2p2s
EM set-qstate
0.0e 60.0e 120.0e rightDet map-angles
draw-experiment go
```

simulating spin measurements using forth: `epr-sim`†



† epr-sim.4th

quantum theory in a couple of slides

for a particle or system of particles in a defined *quantum state*, quantum theory

- predicts probabilities of possible measurement outcomes, *e.g.* $\{P_u, P_d\}$.
- *does not predict*, in general, results of individual measurements.

the above restrictions follow from the axioms and intepretation

- every possible measurement outcome of an *observable* has a *probability amplitude.*
- upon *measurement*, one of the possible outcomes is obtained, *e.g.* $\{+\hbar/2, -\hbar/2\}$.
- probability amplitudes follow a dynamics law (Schrödinger eqn.).
- some observables cannot have precise values simultaneously, *e.g.* $\{x, p_x\}$, $\{s_x, s_z\}$.

quantum states for computer scientists

the *quantum state* is a list of associations between measurement outcomes and probability amplitudes

```
( (mo1 c1) (mo2  c2) … (mo_n c_n) )
```

ex1: single spin-1/2 particle state observed along a specified axis

```
( (up c1) (down c2) )
```

ex2: two spin-1/2 particles state observed along a specified common axis

```
( ((upA upB) c1) ((upA downB) c2) ((downA upB) c3) ((downA downB) c4) )
```

c_i are complex numbers

require $|c_1|^2 + |c_2|^2 + \ldots = 1$

factoring two-particle quantum states

can we factor two-particle states as a product of separate one particle states?

```
(equal ‘( ((uA uB) c1) ((uA dB) c2) ((dA uB) c3) ((dA dB) c4) )

       (product ‘( (uA z1) (dA z2) )  ‘( (uB z3) (dB z4) ) ) )
```

for consistency with probability interpretation, `product` must use the relations

$$
\begin{aligned}
c_1 &= z_1 z_3 \rightarrow |c_1|^2 = |z_1|^2 |z_3|^2 \\
c_2 &= z_1 z_4 \rightarrow |c_2|^2 = |z_1|^2 |z_4|^2 \\
c_3 &= z_2 z_3 \rightarrow |c_3|^2 = |z_2|^2 |z_3|^2 \\
c_4 &= z_2 z_4 \rightarrow |c_4|^2 = |z_2|^2 |z_4|^2
\end{aligned}
$$

then, our Lisp expression evaluates to `T`.

two-particle states can be factored if measurement of one particle is independent of measurement of the other.

unfactorable two-particle quantum states

example of an unfactorable (*entangled*) state:

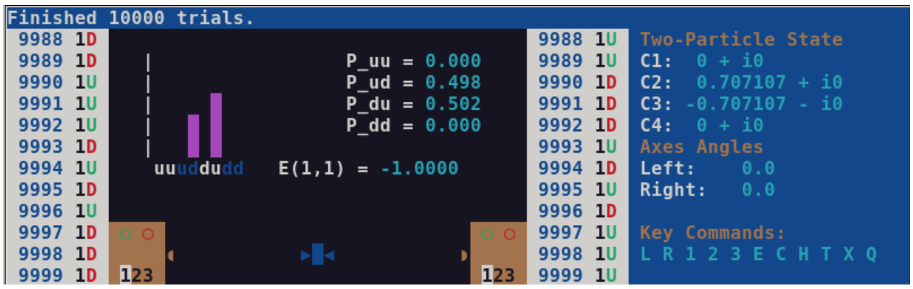
***singlet* two-particle spin state** $c_1 = 0, \; c_2 = 1/\sqrt{2}, \; c_3 = -1/\sqrt{2}, \; c_4 = 0$

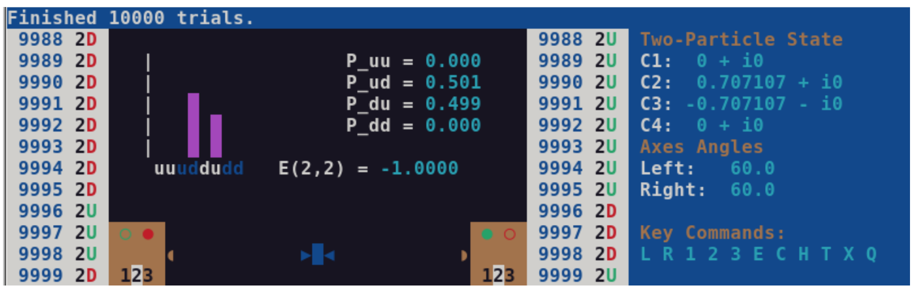
$$
\begin{aligned}
c_1 &= z_1 z_3 = 0 \\
c_2 &= z_1 z_4 = 1/\sqrt{2} \\
c_3 &= z_2 z_3 = -1/\sqrt{2} \\
c_4 &= z_2 z_4 = 0
\end{aligned}
$$

no assignment of z_1, z_2, z_3, z_4 can satisfy the above equations.

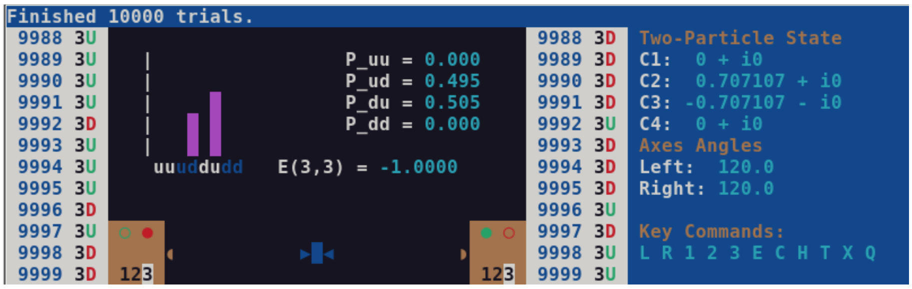
our Lisp expression evaluates to `NIL` for entangled states.

exploring strong correlations in an entangled state using `epr-sim`



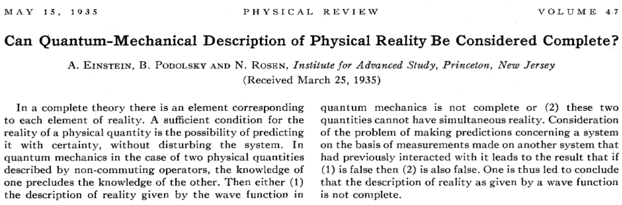


magic of the single state



- each particle, (left and right-going) has equal chance (50%) of spin U or D with respect to any axis.
- measurements for both are perfectly *anti-correlated* when both detectors are set to the same angle – this is the case *for all angles*.

EPR argument for incompleteness of QM [using entangled spins]


MAY 15, 1935 PHYSICAL REVIEW VOLUME 47

Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?

A. Einstein, B. Podolsky and N. Rosen, *Institute for Advanced Study, Princeton, New Jersey*

(Received March 25, 1935)

In a complete theory there is an element corresponding to each element of reality. A sufficient condition for the reality of a physical quantity is the possibility of predicting it with certainty, without disturbing the system. In quantum mechanics in the case of two physical quantities described by non-commuting operators, the knowledge of one precludes the knowledge of the other. Then either (1) the description of reality given by the wave function in quantum mechanics is not complete or (2) these two quantities cannot have simultaneous reality. Consideration of the problem of making predictions concerning a system on the basis of measurements made on another system that had previously interacted with it leads to the result that if (1) is false then (2) is also false. One is thus led to conclude that the description of reality as given by a wave function is not complete.

- left and right detectors can be arbitrarily far apart, and at different distances from the source.
- after a measurement is made on the left, result of measurement on the right, along the same axis, *may be predicted with certainty*.
- measurement on the left cannot in any way disturb the measurement made on the right.
- The axis selection may be random, for example along z-axis (0°) or along x-axis (90°).

therefore, the result of spin measurement on the right exists independently of the measurement on the left, and the quantum state description is incomplete.

A. Einstein, B. Podolsky, and N. Rosen, Physical Review 47, 777 (1935).
D. Bohm and Y. Aharonov, Physical Review 108, 1070 (1957).

hidden variable explanations for spin correlations

assume there exists a *complete* state description with parameter(s) we don't know.

let λ be a random bit (0 or 1) generated at source, and state be specified by

```
λ       s>z zvalue λ1
λ   0=  s>z zvalue λ2
```

```
( ((u u) 0) ((u d) λ1) ((d u) λ2)) ((d d) 0) )
```

which is a factorable (unentangled) state.

λ = 0: `( ((d u) -1) )`

λ = 1: `( ((u d)  1) )`

outcomes are fully determined along 0° when *hidden variable* λ is known:

$A(\lambda{=}0,\ 0^\circ) = $ D, $A(\lambda{=}1,\ 0^\circ) = $ U, $B(\lambda{=}0,\ 0^\circ) = $ U, $B(\lambda{=}1,\ 0^\circ) = $ D

can we find deterministic laws which agree with QM statistics for singlet state?

possible assignments for spin measurements are shown in table

λ	$A(\lambda,\ \theta_i)$			$B(\lambda,\ \theta_j)$		
	1	2	3	1	2	3
0	D	D	D	U	U	U
0	D	D	U	U	U	D
0	D	U	D	U	D	U
0	D	U	U	U	D	D
1	U	D	D	D	U	U
1	U	D	U	D	U	D
1	U	U	D	D	D	U
1	U	U	U	D	D	D

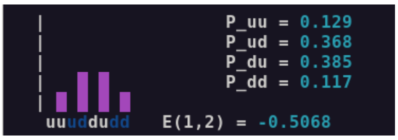
$i,\ j$	P_{uu}	P_{ud}	P_{du}	P_{dd}
1, 1	0	½	½	0
2, 2	0	½	½	0
3, 3	0	½	½	0

exploring hidden variables explanations with `epr-sim`

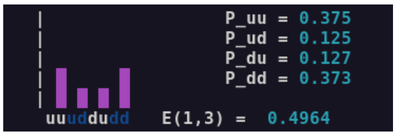
when detector settings are different, QM statistics *do not match* the table statistics.

i*, *j

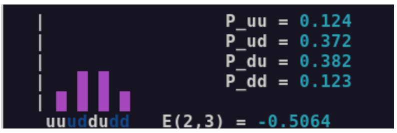
1, 2



1, 3



2, 3



λ	$A(\lambda, \theta_i)$			$B(\lambda, \theta_j)$		
	1	2	3	1	2	3
0	D	D	D	U	U	U
0	D	D	U	U	U	D
0	D	U	D	U	D	U
0	D	U	U	U	D	D
1	U	D	D	D	U	U
1	U	D	U	D	U	D
1	U	U	D	D	D	U
1	U	U	U	D	D	D

i, *j*	P_{uu}	P_{ud}	P_{du}	P_{dd}
1, 2	¼	¼	¼	¼
1, 3	¼	¼	¼	¼
2, 3	¼	¼	¼	¼

correlation coefficient and Bell's inequality for hidden variable theories

E is defined to be the average of the product of the two spin measurements, with **U** ≡ +1 and **D** ≡ −1.

$$E = P_{uu} + P_{dd} - P_{ud} - P_{du}$$

E is also the *correlation coefficient* (reflective correlation coefficient†).

E depends on the two detector angles, θ_L and θ_R (left and right).

J. S. Bell proved‡ that *any* local hidden variable theory must give Es satisfying the following inequality for the singlet state

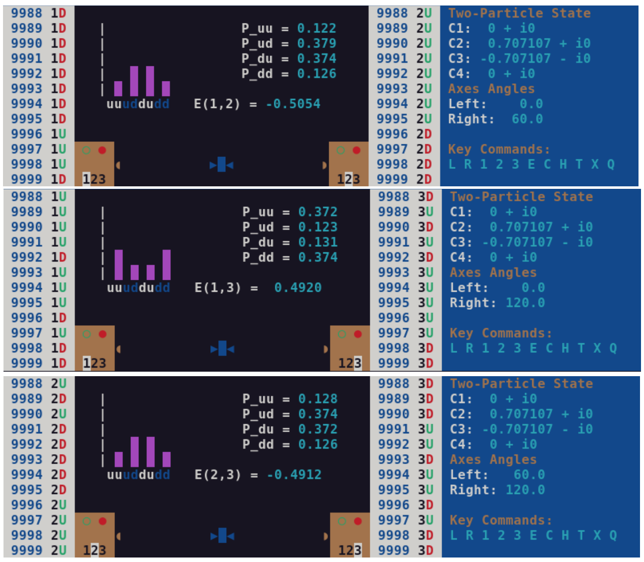
$$|E(1,2) - E(1,3)| - E(2,3) \leq 1$$

where (1,2), (1,3), and (2,3) correspond to left and right detector angle selector settings.

† https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

‡ J. S. Bell, Physics 1, 195 – 200 (1964).

computing Bell's inequality with `epr-sim`




```
10000 value NTRIALS
Singlet EM set-qstate
: f3dup 2 fpick 2 fpick 2 fpick ;

: measure-along ( leftaxis rightaxis -- ) ( F: -- E )
    reset-counts
    rightDet set-axis  leftDet set-axis
    NTRIALS run-fixed-trials drop
    expectation-value ;

: measure-lhs ( -- ) ( F: deg1 deg2 deg3 -- lhs )
    f3dup
    leftDet  map-angles
    rightDet map-angles
    2 3 measure-along
    1 2 measure-along
    1 3 measure-along
    f- fabs fswap f- ;

0.0e 60.0e 120.0e measure-lhs
```

$$|E(1,2) - E(1,3)| - E(2,3) = 1.5 \not\leq 1$$

exercise in using `epr-sim`

Consider the two-particle spin state:

$$c_1 = ½ \quad c_2 = i(½) \quad c_3 = i(½) \quad c_4 = -½$$

Obtain the joint probabilities P_{uu}, P_{ud}, P_{du}, P_{dd} and the correlation, E, for the following pairs of axes:

1, 1 $:=$ $0°,\ 0°$
2, 2 $:=$ $60°,\ 60°$
3, 3 $:=$ $120°,\ 120°$
1, 2 $:=$ $0°,\ 60°$
1, 3 $:=$ $0°,\ 120°$
2, 3 $:=$ $60°,\ 120°$

Setup commands:

```
0.0e 60.0e 120.0e  f3dup
leftDet  map-angles  rightDet map-angles
Q2p2s new constant TestState
z1/2 zdup i* zdup z1/2 znegate TestState init-2p2s
TestState EM set-qstate
draw-experiment go
```

Do the measurements appear to show any correlation for these settings?

Is the two-particle state *entangled*, or is it *factorable* into independent one particle states (Bell's inequality cannot be used for this state)?

`epr-sim` design: forth libraries

forth libraries

`mini-oof.x` compact, object-oriented programming word set by Bernd Paysan†
`ansi.x` ANSI terminal control library‡
`strings.x` simple strings library‡

`forth scientific library`‡
- `fsl-util.x`
- `complex.x` (**#60**)
- `ran4.x` (**#24**)

† Detailed Description of Mini-OOF
‡ kForth-64 forth source examples
††The Forth Scientific Library ; Forth-94 and Forth-2012 compliant Forths may also use kForth versions of FSL modules with the addition of a few compatibility definitions.

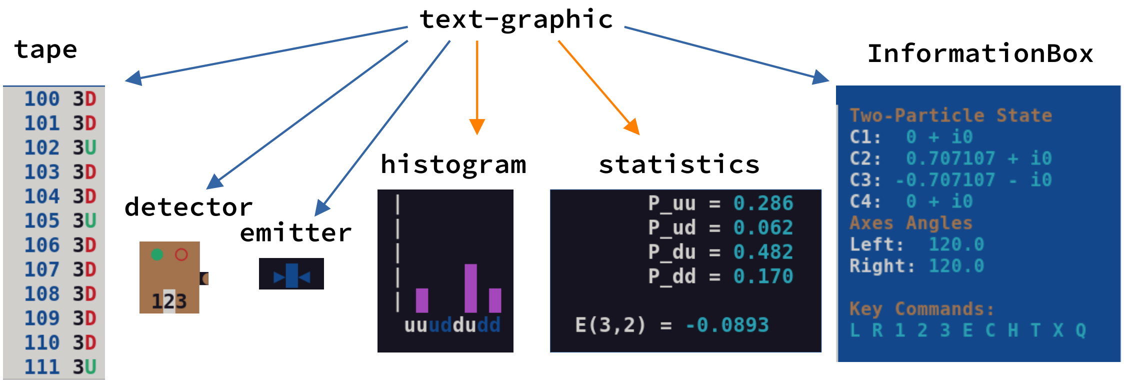
`epr-sim` design: two-particle spin-1/2 state

```
object class
    complex var C1  \ amplitude of |11> component
    complex var C2  \ amplitude of |10> component
    complex var C3  \    "          |01> component
    complex var C4  \    "          |00> component
    method init-2p2s ( o -- ) ( F: z1 z2 z3 z4 -- )
    method normalize ( o -- )
    method exchange  ( o -- )  \ exchange particle labels
    method P_up      ( o -- ) ( F: stheta ctheta -- P_up )
    method M_up      ( o -- ) ( F: stheta ctheta -- C1' C2' C3' C4' )
    method M_down    ( o -- ) ( F: stheta ctheta -- C1' C2' C3' C4' )
end-class Q2p2s  \ two-particle, bipartite quantum state
```

method `normalize` ensures total probability = 1
method `P_up` computes $P_{uu}(\theta_1) + P_{ud}(\theta_1)$

epr-sim design: oop

virtual experiment components are derived from the `text-graphic` class



some visual elements inspired by N. D. Mermin, Physics Today, April 1985, pp 38 -- 47.

dedication

My presentation is dedicated to the memory of professors from whom I learned quantum theory,

Prof. Shi-Yu Wu

Prof. Eugen Merzbacher

appendix: product state of single particles

we have to map $c_i = f_i(\, z_1, z_2, z_3, z_4\,)$ with following constraints

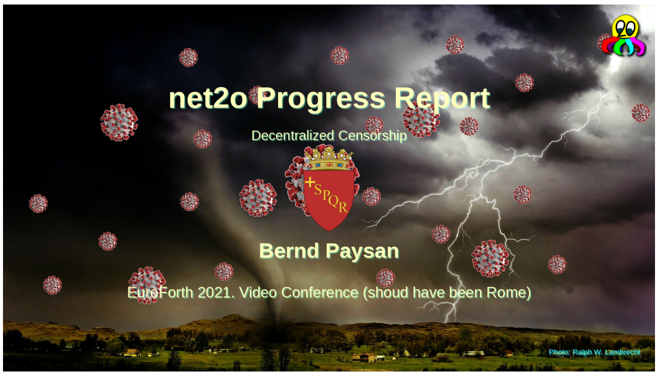
$$P_{uu} + P_{ud} + P_{du} + P_{dd} = |c_1|^2 + |c_2|^2 + |c_3|^2 + |c_4|^2 = 1$$

$$P_u^A = P_{uu} + P_{ud} \rightarrow |z_1|^2 = |c_1|^2 + |c_2|^2$$

$$P_d^A = P_{du} + P_{dd} \rightarrow |z_2|^2 = |c_3|^2 + |c_4|^2$$

$$P_u^B = P_{uu} + P_{du} \rightarrow |z_3|^2 = |c_1|^2 + |c_3|^2$$

$$P_d^B = P_{ud} + P_{dd} \rightarrow |z_4|^2 = |c_2|^2 + |c_4|^2$$



Motivation

Bad Gateway
Internetkurort

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak, symmetric block encryption with Threefish onion routing camouflage with Threefish/Keccak
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files, messages) and distributed metadata (DHT, DVCS)
7. Apps in a sandboxed environment for displaying content

1¾ years into COVID–19 pandemics

Surveillance Capitalism

- Apple wants to scan your pics locally for child porn
- ➡ Had to back down quickly
- Facebook & Twitter "check for facts"
- ➡ Actually still distribute a lot of disinformation
- Telegram became tool of choice of Covidiots
- ➡ Free speech seems to be a problem
- Twitter tests "safe space" feature…
- ➡ The algorithm hides what could hurt you

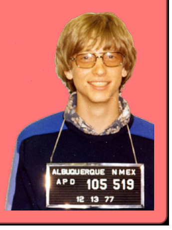
Progress

Little on net2o, more on Bernd 2.0
TCP/IP turns 40

Disinformation

Lessons learned during the pandemics

First Impression — Facts don't change our minds [2]
Ikea Effect — Easy to obtain things have "no value" [3]
Worldview — lets us dismiss facts that don't fit into it
Science — needs to be prudent
Plausibility — This man has done evil things many times
It's just that he doesn't need to chip you
He already has everything he wants to
QAnon — suspected origin: Wu Ming (五名, 5 names)
Wu Ming can be pronounced as 无明, Ignorance



Antisocial Hateworks

Problems with People since Eternal September

Opinions — are not facts, but values people believe in
Beliefs — are not up to discussion, but part of identity
Identity — is viguroulsy defended and used to segregate people
Walls — are in the head, and tearing them down causes aggression

Free Speech — I'm more and more convinced, that "speech" is too generic.
Lies and deception need no protection.
Claims need proof and evidence.
Truth needs protection.

xkcd.com/386

Things done: "Who has What"

Query object origin by hash

- ❓ Original plan: keep hashes in DHT
- ➡ Query reveals who wants what
- ❓ Original solution: Encrypt hashes
- ➡ Query reveals who wants/has the same thing
- ❓ Onion routing within DHT?
- ➡ Complex, slow
- ➡ Better keep "who has what" within the chat log structure
- ➡ "who" is device.pubkey
- ❓ Missing: limit reach of "who has what"

In Progress: Harfbuzz (in MINOΣ2)

- Purpose: Do the more complex part of Unicode rendering
- The interface is actually not that difficult
- But requires restructuring code
- And thinking about right to left scripts
- (and top-to-bottom like Mongolian)
- Challenge: select text by font
- because combiners allow to mix them

Done: Formated chat messages

- Inspired by Mattermost
- Format parsing different from Markdown (simpler)
- Disabled by default
- Sender parsed, so sender parser can change

Mostly done: Voice messages

- Uses Pulseaudio on Linux, OpenSLES on Android
- Encoding in Opus
- OpenSLES recording doesn't work yet
- Android problems with callbacks into Gforth's dynamic code
- The rest is similar to pictures
- Thumbnail is a wavefrom plot with max level/second

1

Decentralized Censorship

Make net2o a better place

- Internal, not external censorship
- Disinfodemic in a peer2peer network similar to pandemic models
- Filtering on incoming content, not your own content
- Sender does not know that content is blocked
- Different settings possible:
  1. Filter hides messages
  2. Filter doesn't transmit messages
  3. Both ("sterile immunity")
- Typical fanout of participants = R_0
- If more than $1–1/R_0$ filter, bad contents doesn't get far
- Requires easy filter sharing

What kind of bad text?

It's not bad words

- Teh spellink is awefull
- SHOUTING ALL THE TIME
- Number (and color) of exclamation marks ‼ ‼
- This sounds like easy to defeat — for smart people
- Smart people are rarely the problem…
- This is porn according to AI:

Disinfodemic

Examples from the Covid pandemic

李文亮 Was gag ordered by Wuhan police when the main news (新闻联播) already had a report. "Would not happen here"

Here? Instead, a hell lot of disinformation spread out in the free west

Evil Govt Yes, the government is evil. But also incompetent. And its bias is pro corporations. Evilness serves a purpose.

Science? Science questions everything. But it conducts experiments to check. Masks work. Lockdowns work. Wuhan lab didn't leak. Vaccines are safe. Ivermectin/Chloroquin/Vitamine D are no miracle cure.

Massacre The failure (willful/incompetent) to contain Covid-19 is a massacre. Democraties can do such atrocities only with massive disinformation.

Filter Algorithms in Real Life

- Algorithm based on words
- Click worker paid per case
- Weird rules
- Want to keep the idiots

  Because they click the ads which are frauds…

Martin Perscheid
† 31. Juli 2021 — RIP

Manual moderation?

Too late, too little

- Delete bad content
- Leave the corrections
- Block the bad actors
- In a P2P network, people can block the moderators
- So a rough consensus is needed
- Manual interaction is too slow
- People don't read rectifications

The non–technical problems

- Get your contacts over to net2o
- How to make a social network a nice place?
- Funding of net2o?

Automatic filter?

Actually the hard problem

Texts: Bad texts (equals PCR test)
- \+ Easy to implement
- – Easy to defeat, easy to be false positive

Images: Fingerprints
- \+ Medium difficulty to implement
- – Easy to defeat, easy to generate pre–image attacks

Audio: Speech to text
- \+ Medium difficulty to implement
- – Defeat is unclear, easy to generate pre–image attacks

Literatur & Links

Bernd Paysan *net2o fossil repository*
https://net2o.de/

The New Yorker *Why Facts don't change our Mind*
https://www.newyorker.com/magazine/2017/02/27/why-facts-dont-change-our-minds

Sascha Lobo *QAnon — Verschwörungsideologie zum Mitmachen*
https://www.spiegel.de/netzwelt/netzpolitik/qanon…

The case for <BUILDS

Brad Rodriguez
EuroForth Conference
12 September 2021

Resident Forth on small embedded microcontrollers
e.g. MaxForth on DSP5680x,
CamelForth on MSP430

Typically code space in Flash ROM, and a small amount of RAM
(e.g. MSP430G2553: 16 K Flash ROM, 0.5K RAM)

Resident Forth compiler needs to compile directly to Flash ROM.

Primary characteristic: each memory location can be written *once*.
Locations cannot be changed once written;
Flash can be erased but only in large blocks.
Usually erased to all ones.

For most compiler actions this is a minor change, e.g.,

```
: IF  [‘] ?BRANCH I, IHERE 0 I, ; IMMEDIATE
```

becomes

```
: IF  [‘] ?BRANCH I, IHERE CELL IALLOT ;
IMMEDIATE
```

(“I” prefix refers to Instruction space, i.e., Flash ROM)

The problem is CREATE ... DOES> . CREATE leaves the run-time action DOCREATE in the newly defined word’s code field. This means that DOES> cannot change that action.

```
: CREATE   HEADER  DOCREATE I, HERE I, ;
```

(Note ROMable variables/data compile a pointer to HERE.)

CREATE is overloaded!

CREATE currently overloads two functions:
1. To define a data strucutre.
2. To define a “defined word” in a CREATE...DOES> construct.

These are conceptually distinct uses, and it is an accident of history that we use CREATE for both (because it has been easy for DOES> to change the action of a CREATEd word).

<u>The solution is to bring back <BUILDS</u> , which performs the function of CREATE but does not store anything in the code field (i.e., leaves that cell unprogrammed).

```
: <BUILDS   HEADER CELL IALLOT ;
```

This does not solve the problem of applying DOES> a *second* time to a defined word's code field, but that is an extremely rare usage.

Standards Compliance:
This becomes a *non-standard-compliant* Forth system, as CREATE...DOES> cannot be used.

But applications are modified easily, by changing CREATE...DOES> to <BUILDS...DOES> wherever used.

Alternatives:
1. Make DOCREATE (the run-time action of a CREATEd word) test a second cell for a DOES> pointer.
2. Make the inner interpreter test for CFA=$FFFF and invoke DOCREATE.
3. Make $FFFF the code address for CREATE. (Rarely viable.)

microCore progress

kschleisiek at freenet.de

Finally, microCore has been published:
https://github.com/microCore-VHDL

gforth_0.6.2 has been wrapped into a docker file (thanks Ulli). It turned out to be completely useless for Windows10, because the serial interface can not be connected.

In order to support the IoT, a 10baseT ethernet interface has been realized.

It just uses a couple of passive components as PHY.

- 10baseT is realised in the FPGA consuming 500 XP2 LUTS.
- (R)ARP and UDP is realised in software consuming 2030 instructions including the multi tasker.

Where does X spend its time?
A small Forth profiler

EuroForth 2021

Philip Zembrod - pzembrod@gmail.com

Motivation

- cc64 compiler written in ITC VolksForth on C64 felt slow …
- … unreasonably slow …
- Solution: optimize hotspots ...
- … which were unknown

Where did cc64 spend its time?

Wish for a profiler that works

- at different levels - module group, module, word group, word
- ideally self-hosted

Prior art & tips

- timing individual words
- let NEXT log all words to stdout
- a C-written VM could be instrumented
- per-word e2e time tracking
  https://sourceforge.net/p/forth-brainless/code/HEAD/tree/trunk/profiler.fs
  - gross time rather than net time

No clear fit for my problem …
… there seemed to be an opportunity^wexcuse for a new tool. :-)

What could I do with NEXT?

- count invocations of a word
- count NEXT cycles within a word
  - # of IP fetches at addresses between : and ;
- count NEXT cycles and sum up time within a word
- count NEXT cycles and sum up time within a range of words
- split cc64 code into N ranges aka buckets
  - count NEXT cycles and sum up time per bucket
- split a bucket into N sub-buckets, rinse & repeat

… this could fly ...

Some details

- NEXT should remain fast
  - only single-interval buckets
  - only 8 buckets -> 3-cmp binary search
  - unrolled loop
- What about Forth core code?
  - core NEXT cycles & time added to calling bucket
- What about non-core code outside buckets?
  - default bucket 0 collects rest of NEXT cycles & time

- Time measurement
  - 2 cascaded 16-bit timers (MOS 6526 CIA) running at CPU clock

```
: compareIp
  IP 1+ lda  >buckets[ ,x cmp  0= ?[ IP lda  <buckets[ ,x cmp ]? ;

: findBucket
  0 # ldx  compareIp  CC ?[
    currentBucket ldx
  ][ inx  compareIp  CC ?[
      dex
    ][
      5 # ldx
      compareIp  0<> ?[  CC ?[ dex dex ][ inx inx ]?
      compareIp  0<> ?[  CC ?[ dex     ][ inx     ]?
      compareIp          CC ?[ dex     ]?             ]? ]?

      IP 1+ lda  >]buckets ,x cmp  0= ?[ IP lda  <]buckets ,x cmp ]?
        CS ?[ 0 # ldx ]?

      txa  .a asl  .a asl  tax
    ]?
    currentBucket stx
  ]? ;
```

```
Label prNext
  timerActrl lda  pha  $fe # and  timerActrl sta
  calcTime
  findBucket
  incCountOfBucket
  addTimeToBucket
  setPrevTime
  incMainCount
  pla  timerActrl sta
  0 # ldx  clc  IP lda  Next $c + jmp


Code install-prNext
 prNext 0 $100 m/mod
     # lda  Next $b + sta
     # lda  Next $a + sta
 $4C # lda  Next $9 + sta
 Next jmp  end-code
```

Buckets defined inline in code

```
\prof profiler-bucket [input]
include input.fth
\prof [input] end-bucket

\prof profiler-bucket [scanner]
include scanner.fth
\prof [scanner] end-bucket

\prof profiler-bucket [symtab]
include symboltable.fth
include preprocessor.fth
\prof [symtab] end-bucket
```

- Easy & straightforward
  - for both start & end of bucket
- Only defined in instrumented mode

Alternative considered:

- define bucket with `' word`
  - end of bucket less intuitive
  - difficult with headerless words

Nested buckets for drilling down

```
\prof profiler-bucket [scanner-nextword]
\prof profiler-bucket [scanner-fetchword]

: fetchword ( -- tokenvalue token )  BEGIN (nextword is-comment? WHILE
     2drop skip-comment REPEAT \ ."  fetchword: " 2dup u. u.  word' 2! ;

: accept ( -- )  1 word# +!  fetchword ;

\prof [scanner-fetchword] end-bucket
\prof profiler-bucket [scanner-thisword]

: thisword ( -- tokenvalue token )  word' 2@ ;

\prof [scanner-thisword] end-bucket
\prof [scanner-nextword] end-bucket
```

Up to 8 active buckets

Buckets grouped in metrics

```
\prof include prof-metrics.fth
```

prof-metrics.fth:

```
profiler-metric:[ profile-cc64
  [strings]
  [memman-etc]
  [file-handling]
  [input]
  [scanner]
  [symtab]
  [parser]
  [pass2]
]profiler-metric
```

```
profiler-metric:[ profile-scanner
  [scanner-alphanum]
  [scanner-identifier]
  [scanner-operator]
  [scanner-char/string]
  [scanner-(nextword]
  [scanner-comment]
  [scanner-nextword]
  [scanner-rest]
]profiler-metric

profiler-metric:[
profile-scanner-nextword
  [scanner-nextword-vars]
  [scanner-fetchword]
  [scanner-thisword]
  [scanner-nextword-mark]
  [scanner-nextword-advanced?]
]profiler-metric
```

Design overview

- Hooks into NEXT routine
- Max 8 buckets active per measurement
  - e.g. per instrumented e2 test run
  - 16 or 32 buckets also feasible
- Define arbitrary # buckets inline
- Metrics: bundles of max 8 buckets
  - defined in separate central file
- A metric, invoked interactively, activates its buckets
  - same compiled turn-key binary can run different metrics

Result #1: Don’t list source during compile

profiler report **PROFILE-CC64-1**
timestamps
919.522.732 **1.078.906.060**

buckets

b#	nextcounts	clockticks	name
0	475419	**52822277**	(etc)
1	1037243	114416784	[MEMMAN]
2	1384162	153154008	[FILE-HDL]
3	797224	**122822197**	[INPUT]
4	2695157	**299076306**	[SCANNER]
5	153639	17403250	[SYMTAB]
6	1826434	197679185	[PARSER]
7	1100491	121509788	[PASS2]
8	0	0	[SHELL]

profiler report PROFILE-CC64-1
timestamps
830.786.988 **989.908.172**

buckets

b#	nextcounts	clockticks	name
0	475419	52854657	(etc)
1	1035458	114210370	[MEMMAN]
2	1384162	153117590	[FILE-HDL]
3	313708	**34373231**	[INPUT]
4	2695157	299094501	[SCANNER]
5	153639	17396511	[SYMTAB]
6	1826434	197594285	[PARSER]
7	1100491	121235385	[PASS2]
8	0	0	[SHELL]

Result #2: Eliminate loops in operator scanning

```
profiler report  PROFILE-SCANNER2
timestamps
831.180.204  990.563.532

buckets
b#  nextcounts   clockticks   name
0     6291642    691659740   (etc)
1      247516     29308255   [ALPHANUM]
2       66366      7040651   [ID]
3     1075233    116814924   [OPERATOR]
4       27320      3487498   [NUMBER]
5      237738     26065205   [CHR/STR]
6      150400     15851841   [NEXTWORD]
7       43874      5189985   [COMMENT]
8      844379     95121698   [REST]
```

```
profiler report PROFILE-SCANNER2
timestamps
725.077.164  884.722.893

buckets
b#  nextcounts   clockticks   name
0     6294398    692597293   (etc)
1      247516     29374038   [ALPHANUM]
2       66366      7035251   [ID]
3       93760     10011918   [OPERATOR]
4       27320      3483433   [NUMBER]
5      237738     26075738   [CHR/STR]
6      150400     15872668   [NEXTWORD]
7       43874      5198813   [COMMENT]
8      844379     95104859   [REST]
```

Result #3: nextword/backword -> thisword/accept

```
profiler report  PROFILE-SCANNER3
timestamps
725.011.628  884.395.213

buckets
b#  nextcounts   clockticks   name
0     6322534    695897987   (etc)
1      247516     29341777   [ALPHANUM]
2       66366      7045787   [ID]
3       93760      9996178   [OPERATOR]
4      237738     26092982   [CHR/STR]
5      150400     15872100
[(NEXTWORD]
6       43874      5190433   [COMMENT]
7      844359     95006489   [NEXTWORD]
8          20         2405   [REST]
```

```
profiler report PROFILE-SCANNER3
timestamps
635.489.452  794.873.037

buckets
b#  nextcounts   clockticks   name
0     6225621    685133179   (etc)
1      247524     29335672   [ALPHANUM]
2       66366      7032118   [ID]
3       93760      9997433   [OPERATOR]
4      237738     26098202   [CHR/STR]
5      150422     15861717   [(NEXTWORD]
6       43885      5193847   [COMMENT]
7      149888     16295019   [NEXTWORD]
8          14         1679   [REST]
```

Result #4: alphanum? in code, len-indexed string search

```
profiler report  PROFILE-CC64-1
timestamps
644.730.028  805.359.052

buckets
b#  nextcounts   clockticks   name
0      216494     22948435   (etc)
1      480396     56645293   [STRINGS]
2      947278    104421341   [MEMMAN]
3     1396465    154541959   [FILE-HDL]
4      316896     34711112   [INPUT]
5      832390     90610649   [SCANNER]
6      155294     17603763   [SYMTAB]
7     1858844    201343115   [PARSER]
8     1109617    122555758   [PASS2]
```

```
profiler report PROFILE-CC64-1
timestamps
579.980.460  740.543.948

buckets
b#  nextcounts   clockticks   name
0      216506     22950222   (etc)
1      114310     12976436   [STRINGS]
2      742175     83029931   [MEMMAN]
3     1396517    154575151   [FILE-HDL]
4      316896     34740882   [INPUT]
5      832390     90869725   [SCANNER]
6      155294     17623890   [SYMTAB]
7     1858844    201608564   [PARSER]
8     1109737    122235615   [PASS2]
```

Result overall: 31% time saved, 45% speed gain

```
profiler report  PROFILE-CC64-1
timestamps
919.522.732 1.078.906.060

buckets
b#  nextcounts  clockticks  name
0       475419    52822277  (etc)

1      1037243   114416784  [MEMMAN]
2      1384162   153154008  [FILE-HDL]
3       797224   122822197  [INPUT]
4      2695157   299076306  [SCANNER]
5       153639    17403250  [SYMTAB]
6      1826434   197679185  [PARSER]
7      1100491   121509788  [PASS2]
8            0           0  [SHELL]
```

```
profiler report PROFILE-CC64-1
timestamps
579.980.460  740.543.948

buckets
b#  nextcounts  clockticks  name
0       216506    22950222  (etc)
1       114310    12976436  [STRINGS]
2       742175    83029931  [MEMMAN]
3      1396517   154575151  [FILE-HDL]
4       316896    34740882  [INPUT]
5       832390    90869725  [SCANNER]
6       155294    17623890  [SYMTAB]
7      1858844   201608564  [PARSER]
8      1109737   122235615  [PASS2]
```

Links

All to be found under https://github.com/pzembrod/cc64:

- profiler: src/common/profiler.fth
- profiler activation: src/cc64/invoke.fth
- instrumented code: src/cc64/cc64.fth & src/cc64/scanner.fth
- metrics definitions: src/cc64/prof-metrics.fth
- profiling results: tests/e2e/profile-register
- cc64 input scripts for different metrics: tests/e2e/*.pfs

Conclusion

- Profiler proved practical & easy to use
- Good overview & drilldown with multiple metrics
- Different metrics within one compiled binary
- Result: Small to moderate optimizations yielded 45% speed increase
- 190 lines of code
- Gross runtime penalty for instrumentation < 4x
- Prerequisite: ITC or DTC

Thank you for your attention!

Questions?

Copying Bytes

M. Anton Ertl, TU Wien

Myths

- Copying bytes efficiently is simple
- `Cmove` is faster than `move`
- Implementing `cmove` efficiently is simple
- Implementing `move` efficiently is more complex

Cycles for 50-byte non-overlapping copy

Skylake		Zen 3			
sf	gforth	vfx32		vfx64	
95	36	34	24	232	move
100	87	32	21	27	cmove
83	90	33	21	224	cmove>

byte loop memmove() cell loop rep movsb

Words and C functions

Forth	C	
`move`	`memmove()`	to-range contains original from-range contents
`cmove`		propagates patterns if to $\in$ [from, from + u)
`cmove>`		propagates patterns if from $\in$ [to, to + u)
	`memcpy()`	undefined behaviour on overlap
`move<`		don't call if to $\in$ [from, from + u)
`move>`		don't call if from $\in$ [to, to + u)

Efficient implementations

```
: move ( from to u -- )
  over 3 pick - 2 pick u< if \ to in [from,from+u)
    move>
  else
    move<
  then ;

: cmove ( afrom ato u -- )
    dup 0= if exit then
    begin ( afrom1 ato1 u1 )
        over 3 pick - 2>r
        2dup 2r@ umin move<
        2r@ 1 rot within while
            2r> /string repeat
    2r> 2drop 2drop ;
```

Extend 2-byte pattern to 1000 bytes with `cmove`

```
  Zen 3 cycles/cmove
   VFX64      VFX32
rep movsb  cell loop
orig  new  orig  new
3360  965  4273  386
```

Conclusion

- ~~Moving bytes efficiently is simple~~
- `Cmove` is faster than `move`? Sometimes
- ~~Implementing `cmove` efficiently is simple~~
- ~~Implementing `move` efficiently is more complex~~

Forth - The New Synthesis

progress report

disaggregating the stacks and memory

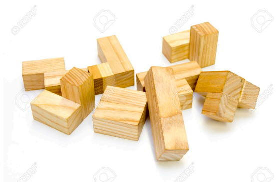
EuroForth'21 conference 2021-09-12

Ulrich Hoffmann

uho@ XLERB .de

Forth the New Synthesis

Forth



disaggregating

synthesizing

Latest work

- investigate in input and output
  - connection between host and target
    - communicating commands between host and target
    - screens
      - do not need to be 1kB BLOCKs   form-feed separated files
      - b n l list load   can work as usual

Current work

- playing with unicode
- disaggregating stacks
- disaggregating memory

playing with Unicode

- Browsing mathematical Unicode symbols, maybe arrows are nice:

```
SYNONYM → TO           5 VALUE x      42 → x    x emit
SYNONYM S→D  S>D       42 S→D D.
SYNONYM ½· 2/          line-width ½·
SYNONYM →BODY >BODY    ' eggs →BODY ...
```

- Greek letters:

```
100 CONSTANT Δt         ...   Δt  ms  ...
```

- Or single symbols where we now have symbol sequences:

```
SYNONYM ≤ <=        ... x 10 ≤ IF ...
SYNONYM ≠ <>        .... x 45 ≠ IF ...
```

But in general I think you have to be careful using symbols as they best need to have a commony accepted meaning.

playing with Unicode

As a counter example, I find symbols for control structures interesting but eventually misleading:

- doubtful

```
SYNONYM ▶ OF
SYNONYM ◀ ENDOF
SYNONYM 『 CASE
SYNONYM 』 ENDCASE

: casetest ( n -- )
   『
     0 ▶ ." no" ◀
     1 ▶ ." one" ◀
     2 ▶ ." two" ◀
     ." many"
   』
   ."  items" ;
```

Disaggregating the Stacks

- data stack and return stack are used for different purposes in different situations.
- disaggregating the stacks means separating these purposes and look at them in isolation.

Disaggregating the Stacks

	Interpreting	Compiling	Executing	comment
Data Stack	parameter passing		parameter passing	
	(unsigned) integers		(unsigned) integers	
	characters		characters	
	floats		floats	
	addresses		addresses	
		control flow		BEGIN IF ...
		compiler security		: ;
		constant folding		
Return Stack	internal return addresses	return addresses	return addresses	
			temporary storage	>R R> R-ALLOT
			loop parameters	DO LOOP
			exception frames	CATCH THROW
			locals	>X X X!

Disaggregating the Stacks

- data stack and return stack are used for different purposes in different situations.

- disaggregating the stacks means separating these purposes and look at them in isolation.

Disaggregating the Stacks

- interferences of the the different purposes lead to restrictions such as:

    - no passing of parameters to definitions at compile time (interference of control flow/compiler security and parameter passing)

    - no use of >R R> across DO-LOOP-boundaries (interference of temporary storage usage and loop parameters)

    - no use of >R R> across definitions (interference of temporary storage and return addressses).

    - specialized stack operators to deal with floating point numbers on the return stack (FDUP, FSWAP, swap cell and float)

Disaggregating the Stacks

Separate stacks for each purpose
Possible disaggregations are

- split data stack into
    - a separate stack for parameter passing that holds (unsigned) integers, characters and also addresses
    - a separate floating point stack for holding floating point numbers (the route Forth-200x went)
    - a separate control flow stack for managing control structures
    - a seperate object stack for handling references to data structures and objects

- split the return stack into
    - a seperate stack for return addreses
    - a seperate stack for temporary data (>R R> R-ALLOT)
    - a seperate stack for loop parameters (DO LOOP)
    - a seperate stack for exception handling (CATCH THROW)
    - a seperate stack for local variables

Disaggregating the Memory

```
: Buffer: ( u -- )
   Create allot ;

: Buffer: ( u -- )
   here swap allot \ RAM  { c0 | ... | cu-1 }
   Create ,        \ ROM  { 'rom }
  Does> ( -- addr ) @
;

: Buffer: ( u -- )
   here swap allot \ RAM
   Constant \ ROM
;
```

<BUILDS

Questions?

DSLs - power & challenge

The underestimated need for design

Lightning talk EuroForth 2021

Philip Zembrod - pzembrod@gmail.com

DSLs are powerful but come at a cost

- With great power comes great responsibility
- "A well-designed language is its own Heaven; a poorly-designed language is its own Hell." *
- Designing a language isn't easy
- Design is a valuable skill
  - … and it takes time and effort

* me, inspired by The Dao of Programming

The underestimated need for design in Forth

- Design is essential in almost every word
  - or else the stack will devour you
  - In Forth I need to think carefully about things I just write down in C or Python
- Impact of design effort:
  - If done well: exceptionally expressive code
  - If not done: write-only code pain
- Beginners should be told this
  - or they will be frustrated
  - for they'll discover the costs anyway

Thoughts on things to acknowledge wrt beginners

- absence of syntax handrails
  - needs design work to mitigate
- absence of type check etc. guard rails
  - needs more testing to mitigate
- freedom and flexibility can be a challenge
- Forth is easy in some ways and hard in others
  - and a great design training field