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 langugaes, 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