Static Stack Effect Analysis

Ulrich Hoffmann

Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität zu Kiel

Preußerstraße 1–9
24105 Kiel
Germany

email: uho@informatik.uni-kiel.de

October 1, 1993

Abstract

It is best to develop Forth programs interactively. Its rigorose incremental bottom up strategy leads to rebust and reliable applications.

Many remaining Forth program errors result from improper stack usage, i. e. mismatch of intended and actual stack behaviour of words. Checking consistency of stack comments and actual stack effects provides a way to further ensure application quality.

Static stack effect analysis determines the actual runtime stack effect of a word at compile time. Starting from a root set of words with known stack effects, the presented algorithm calculates stack effects of newly defined words by using inference rules. The described analysis focuses only on stack depths and not on types of stack items. It is thus only a restricted stack effect analysis but nevertheless has proven to be powerful enough to detect many of the remaining program errors.

The current implementation uses the ANS-Forth core word set as its root set and has the ability to add additional words by simply declaring their stack effects.

1 Introduction

Forth encourages an interactive and incremental development style. This has lots of advantages which we will not discuss here. However at some time the development of an application is more or less stable. This is the time to perform consistency checks on the application (for a classical example see [Har86]) to further ensure its reliability. Many errors in Forth programs are caused by stack mistakes, where the actual stack effect and the intended stack effect do not match. This paper presents an algorithm to calculate stack effects of words by statically examining their definitions at compile time. This opens the opportunity to statically check the consistency of stack comments and the calculated stack effects, and thus enhance application quality. The approach taken here restricts itself to examine *stack depths* only and *not types* of stack items. For more general approaches see [P̈92] or [SK92].

2 Stack Depths

Modelling the depth of the data stack at a given time of execution is the first think we have to cope with. We cannot expect to have enough information to give a precise depth everywhere. This makes it reasonable to distinguish between different levels of exactness for stack depth information:

1. **exact**
   The depth of the data stack is known to be exactly a certain number of stack items. Exact stack depths can be modelled by natural numbers.

2. **range**
   The depth of the data stack is known to lie between two bounds. This can happen if two different pathes to the current point of execution change the stack in different ways. Range stack depths can be modelled by an intervall of natural numbers.

3. **unknown**
   No information about the stack depth can be obtained, when the changes in stack depth depend on solely on the run time data and not on the control flow.

Since a specific natural number can be regarded as a natural interval where both, the lower and the upper bound, are equal, the set of stack depths $\mathcal{D}$

can be defined as:

$$\mathcal{D} =_{def} \{(m,n) \in N \times N \mid m \leq n\} \cup \{?\}$$

? represents the unknown stack depth. Instead of (m,n) we write $m \,..\, n$ to point out the intended meaning, representing an interval by a pair of natural numbers. For exact natural numbers, i. e. intervalls with same upper and lower bound, we write numerals in italics, e.g.: *1* instead of 1 .. 1. If $d \in \mathcal{D}$ then the lower and upper bound of d will be written as d_1 and d_2 respectively.

2.1 Operations on Stack Depths

Next, we need to define some operations on stack depths for later use.

1. **Addition**
   The additon of stack depths describes what happens if we put items on the stack. If at a certain point of execution the stack has the depth $d \in \mathcal{D}$ and the next step will put $e \in \mathcal{D}$ items on the stack, the resulting depth is $d + e$. If both stack depths are exact, then this is equivalent to adding both integers. The definition of $+$, however, will have to take care of intervals and the unknown value ?. This leads to:

   $$\_ + \_ : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$$

   $$d + e =_{def} \begin{cases} d_1 + e_1 \,..\, d_2 + e_2 & \text{if } d, e \in N \times N \\ ? & \text{otherwise} \end{cases}$$

   When we add two intervals, the resulting interval is shifted and is wider than (or as wide as[1]) both of the original intervals. Note, that $\mathcal{D}$ is closed under $+$, so applying $+$ to any two stack depths in $\mathcal{D}$ will result in another stack depth in $\mathcal{D}$.

2. **Subtraction**
   The subtraction of stack depths describes what happens if we remove items from the stack. If at a certain point of execution the stack has the depth $d \in \mathcal{D}$ and the next step will remove $e \in \mathcal{D}$ items from the stack, the resulting depth is $d - e$. Trying to remove more items from the stack than are already on it can be considered an error. Here we decided to leave the resulting stack depth as unknown (?)

[1] We will use *wider* to express *wider or equal* and *strictly wider* if we want to exclude equalness.

for the sake of simplicity. A possible refinement of this analysis can introduce another special stack depth to handle this error situation. For exact stack depths $-$ will be equivalent to the subtraction on natural numbers.

$$\_ - \_ : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$$

$$d - e =_{def} \begin{cases} d_1 - e_2 \mathbin{..} d_2 - e_1 & \text{if } d, e \in N \text{ and } d_1 \geq e_2 \text{ and } d_2 \geq e_1 \\ ? & \text{otherwise} \end{cases}$$

As with addition, if d and e are intervals, the resulting interval is shifted and is wider than both, d and e. $\mathcal{D}$ is closed under subtraction, because all calculation which would result in a value not in $\mathcal{D}$ are mapped to ? by the otherwise clause.

3. **Union**
   The union of stack depths describes what happens if the stack depth at a certain point of execution results from two or more pathes to this point. Since the actual path taken at runtime normally depends on data, all pathes must be considered possible. The resulting depth must be as weak as is necessary to satisfy stack changes on any path to this point. Note that the actual data may ensure that certain pathes will not be taken, but a static analysis cannot predict this behaviour. We define:

   $$\_ \sqcup \_ : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$$

   $$d \sqcup e =_{def} \begin{cases} \min(d_1, e_1) \mathbin{..} \max(d_2, e_2) & \text{if } d, e \in N \times N \\ ? & \text{otherwise} \end{cases}$$

   As with addition and subtraction, $\mathcal{D}$ is close under union. If both arguments are the same exact number than so is the result. If at least one of them specifies a range then the result is a range, which is wider than both of the arguments. Any ? argument causes the result also to be ?.

4. **Comparison** To compare two stack depths we look at their lower bounds. We define:

   $$\_ < \_ \subseteq \mathcal{D} \times \mathcal{D}$$

   $$d < e \Leftrightarrow_{def} d, e \in N \times N \text{ and } d_1 < e_1$$

5. **Equivalence** Two stack depths will be considered equivalent if they have the same lower bound. We define:

$$\_ \equiv \_ \subseteq \mathcal{D} \times \mathcal{D}$$

$$d \equiv e \Leftrightarrow_{def} d, e \in N \times N \text{ and } d_1 = e_1$$

$\equiv$ is an equivalance relation on $\mathcal{D}$.

3 Stack Effects

After knowing how to characterize stack depths, we consider stack effects next. We define stack effects as pairs of stack depths, describing the stack depth before and after execution of (a sequence of) words:

$$\mathcal{E} =_{def} \mathcal{D} \times \mathcal{D}$$

where $(m, n) \in \mathcal{E}$ is written as $(m \; -- \; n)$. This tries to resemble the traditional notation of stack effects.
The inference rules from section 4 describe how to combine stack effects of word compositions from their components. This process starts by knowing the stack effects of primitive (system defined) words. The set of primitive words with known stack effects is called *root set.*
The following table gives some stack effects for primitive words:

DUP	(1 -- 2)
SWAP	(2 -- 2)
DROP	(1 -- 0)
OVER	(2 -- 3)
CELL+	(1 -- 1)
@	(1 -- 1)
+	(2 -- 1)
0=	(1 -- 1)
0	(0 -- 1)

4 Inference Rules

The inference rules will cover most combination of words with the exception of non local control transfers. They will be discussed in section 4.4. The

rules have the form

$$(\texttt{name}) \qquad \frac{\text{conclusion}}{\text{premise}}$$

where both, the premise and the conclusions are either stack effects or compositions of stack effects. Let for all rules be $m, n, o, p \in \mathcal{E}$ and $d \in \mathcal{D}$.

4.1 Sequential Composition

The first three rules handle the case of sequential composition, i. e. what happens to the stack if two (sequences of) words are executed one after the other.
Two of them describe what happens if executing the first word leaves a different number of items on stack than the second word needs as arguments. The stack effect of either the first or the second word is adjusted so their interface match. After that the third rule can be applied.

1. Executing the first word leaves *more* items on the stack than the second word needs as arguments. We adjust the stack effect of the second word:

$$(\texttt{seq}_1) \qquad \frac{(m\ --\ n)(o+d\ --\ p+d)}{(m\ --\ n)(o\ --\ p)} \qquad o < n$$

   where $d = n_1 - o_1\ ..\ n_1 - o_1$.

2. Executing the first word leaves *less* items on the stack than the second word needs as arguments. We adjust the stack effect of the first word:

$$(\texttt{seq}_2) \qquad \frac{(m+d\ --\ n+d)(o\ --\ p)}{(m\ --\ n)(o\ --\ p)} \qquad n < o$$

   where $d = o_1 - n_1\ ..\ o_1 - n_1$.

3. Executing the first word produces a stack depth which is equivalent to the stack depth the second word expects. Both stack effects can be combined then:

$$(\texttt{seq}_3) \qquad \frac{(m\ --\ n+p-o)}{(m\ --\ n)(o\ --\ p)} \qquad n \equiv o \text{ or } n =? \text{ or } o =?$$

4.2 Conditionals

Conditionals have the form `IF` words `THEN` or `IF` words `ELSE` words `THEN`. They are handled in the following rules:
The first rule for conditionals transforms a one way `IF THEN` conditional to a two way `IF ELSE THEN` conditional by inserting an empty `ELSE`-branch.

$$(\texttt{if}_0) \qquad \frac{\texttt{IF}\ (m\ --\ n)\ \texttt{ELSE}\ (0\ --\ 0)\ \texttt{THEN}}{\texttt{IF}\ (m\ --\ n)\ \texttt{THEN}}$$

The next two minor cases handle mismatches of the stack effects in both branches similar to the sequential composition from section 4.1. Here, the aim is to transform the conditional so the initial stack depths of both branches are equivalent. If the words in one branch need more stack items than those in the other branch, the stack effect in the latter is adjusted accordingly.

$$(\texttt{if}_1) \qquad \frac{\texttt{IF}\ (m\ --\ n)\ \texttt{ELSE}\ (o+d\ --\ p+d)\ \texttt{THEN}}{\texttt{IF}\ (m\ --\ n)\ \texttt{ELSE}\ (o\ --\ p)\ \texttt{THEN}} \qquad o < m$$

where $d = m_1 - o_1\ ..\ m_1 - o_1$.

$$(\texttt{if}_2) \qquad \frac{\texttt{IF}\ (m+d\ --\ n+d)\ \texttt{ELSE}\ (o\ --\ p)\ \texttt{THEN}}{\texttt{IF}\ (m\ --\ n)\ \texttt{ELSE}\ (o\ --\ p)\ \texttt{THEN}} \qquad m < o$$

where $d = o_1 - m_1\ ..\ o_1 - m_1$.

Applying these rules will result in a conditional which the next rule can handle. Both branches expect an equivalent stack depth.

$$(\texttt{if}_3) \qquad \frac{(m \sqcup o\ +1\ --\ n \sqcup p)}{\texttt{IF}\ (m\ --\ n)\ \texttt{ELSE}\ (o\ --\ p)\ \texttt{THEN}} \qquad m \equiv o \text{ or } m =? \text{ or } n =?$$

The resulting stack effect is to take this amount of items and to produce a stack depth as weak as necessary to satisfy both branches.
Using the rules seen so far the stack effect of `?DUP` can be infered by its definition:

```
: ?DUP DUP IF DUP THEN ;
```

Inserting the stack effect of `DUP` leads to:

```
: ?DUP (1 -- 2) IF (1 -- 2) THEN ;
```

Rule $\texttt{if}_0$ extends this to:

: ?DUP (*1 -- 2*) IF (*1 -- 2*) ELSE (*0 -- 0*) THEN ;

The stack effect of the ELSE-clause must be adjusted according to rule $\texttt{if}_1$:

: ?DUP (*1 -- 2*) IF (*1 -- 2*) ELSE (*1 -- 1*) THEN ;

Now, rule $\texttt{if}_3$ can be applied:

: ?DUP (*1 -- 2*) (*2 -- 1 .. 2*) ;

Finally, the rule $\texttt{seq}_3$ gives the result:

: ?DUP (*1 -- 1 .. 2*) ;

Thus ?DUP takes one argument and depending on its value leaves one or two items on the stack.

4.3 Repetitions

Repetitions cover the constructs:

- BEGIN words UNTIL
- BEGIN words WHILE words REPEAT
- DO words LOOP
- DO words +LOOP

We will restrict ourself to present only the rules for the BEGIN UNTIL-loop in this paper. The rules for the other repetitions are similar in spirit.
Three minor cases can be distinguished for BEGIN UNTIL-loops:

1. The stack depth is an invariant of the loop. Most of the loops in practise have this property. In this case we will not lose any exactness.

$$(\texttt{until}_1) \qquad \frac{(m\ --\ m)}{\texttt{BEGIN}\ (m\ --\ n)\ \texttt{UNTIL}} \qquad n = m + 1$$

2. Running through the loop puts items on the stack. In this case we can at least keep the number of items that have to be on the stack for the very first loop iteration. Unfortunately we do not know how many items will be on the stack when the loop terminates, so the resulting stack depth is unknown.

   $(\texttt{until}_2)$ $\quad \dfrac{(m\ --\ ?)}{\texttt{BEGIN}\ (m\ --\ n)\ \texttt{UNTIL}} \quad m+1 < n \text{ or } m+1 \equiv n$

3. Running through the loop removes items from the stack, or the stack effect of the loop body is unknown. In this case we neither no how many items have to be on the stack when the loop is entered nor do we know how many there will be on exit.

   $(\texttt{until}_3)$ $\quad \dfrac{(?\ --\ ?)}{\texttt{BEGIN}\ (m\ --\ n)\ \texttt{UNTIL}} \quad n < m+1 \text{ or } m =? \text{ or } n =?$

Using these UNTIL-rules the following loop, which traverses a linked list and sums up its elements, can be analysed:

```
0 SWAP
BEGIN
     SWAP OVER CELL+ @ +
     SWAP @ DUP 0=
UNTIL
DROP
```

First, the stack effects for the primitive words from the root set will be inserted:

```
(0 -- 1)(2 -- 2)
BEGIN
      (2 -- 2)(2 -- 3)(1 -- 1)(1 -- 1)(2 -- 1)
      (2 -- 2)(1 -- 1)(1 -- 2)(1 -- 1)
UNTIL
(2 -- 1)
```

Next, according to the first two rules of sequential composition, the stack effects are adjusted, so their interfaces match:

```
(1 -- 2)(2 -- 2)
BEGIN
      (2 -- 2)(2 -- 3)(3 -- 3)(3 -- 3)(3 -- 2)
      (2 -- 2)(2 -- 2)(2 -- 3)(3 -- 3)
UNTIL
(2 -- 1)
```

Now, rule $\texttt{seq}_2$ can be applied and leads to:

```
(1 -- 2)
BEGIN
      (2 -- 3)
UNTIL
(2 -- 1)
```

The rule $\texttt{until}_1$ simplifies the `BEGIN UNTIL`-loop:

```
(1 -- 2) (2 -- 2) (2 -- 1)
```

Finally $\texttt{seq}_3$ again gives the final result:

```
(1 -- 1).
```

Thus the loop will consume exactly one argument and will leave exactly one result on the stack.

4.4 Non local Control Transfers

The inference rules described in the last section cover many of the constructs that appear in Forth programs. There are however words left, which the inference rules do not cover:

- **Abortions**
  These words terminate the program execution. Examples are `ABORT`, `ABORT"` or `QUIT`. For a stack depth analysis they can be treated as jokers, since the stack effect of a word only describes what happens if the word terminates normally.

- **Escapes**
  These words prematurely leave the current executed word or loop. Examples are `EXIT` or `LEAVE`. For a stack depth analysis they can be handled similar to conditionals. At a conditional the flow of control

is split into several execution pathes. When these pathes join again, their stack effects have to be combined as can be seen in rule $\texttt{if}_3$. Escapes change the current execution path to continue at the end of the definition or enclosing loop. Thus the stack effects leading to these points have to be combined in a similar way for proper handling of escapes. Unfortunately inference rules make it difficult to express non local control transfers, so they are not described in this framework.

5 Conclusion

We presented a way to model stack depths and stack effects in an abstract way. Starting from known stack effects of primitive words we calculated stack effects of defined words by using inference rules. Using the algorithm presented in this paper the author has implemented a static stack effect analysis which is able to obtain stack effects of Forth programs based in the ANS-Forth core word set. This analysis can be extended to become a convenient consistency checker for Forth applications.

References

[Har86] Kim Harris. Analyzing large Forth programs by using the STRUCTURE-TOOL program. In *FORML'85 conferece proceedings*, San Jose, California, 1986. Forth Interest Group.

[Pö92] Jaanus Pöial. Multiple stack effects of FORTH programs. In *euroFORML'91 conference proceedings*, Oakland, California, 1992. Forth Interest Group.

[SK92] Bill Stoddart and Peter Knaggs. Type inference in stack based languages. In *euroFORML'91 conference proceedings*, Oakland, California, 1992. Forth Interest Group.