Stack Effect Algebra

1 Introduction

This document summarizes the mathematical foundations for automatically deriving stack effects in a concatenative stack-oriented programming language. We consider a stack-oriented programming language like *Forth*, where functions are called words and operate on a value stack. Words pop arguments from the top of the stack and push return values back on the stack. Alternatively, one could think of words as functions that take as single argument the whole stack and return an entirely new stack. These two ways of thinking about words are equivalent.

Forth uses a convention to include stack effects as comments in newly defined words. Stack effects describe how a word modifies the stack. They are written as `( <inputs> -- <outputs> )`, where `<inputs>` describes the top of the stack before execution of the word and `<outputs` outputs describes the top of the stack after execution. Both are whitespace delimited lists that assign names to items on the stack and may specify data types with a prefix notation.

Example stack effects:

`( -- )`	identity; no stack modification
`( x -- )`	drop top item from stack
`( -- n )`	push a new signed integer on the stack
`( x -- x x )`	duplicate item on the stack
`( x1 x2 -- x2 x1 )`	swap two items

The language *Factor* takes stack effects a step further. In Factor, they are not mere comments, but are checked by the compiler for consistency with the code. In contrast to Forth, Factor is dynamically typed and can have quotations (anonymous words) on the stack. Combinators are words that operate on quotations. The stack effect of a combinator can depend on the stack effect of the quotations it gets as input, which may not be known in advance. Therefore, Factor extended the syntax for stack effect declarations with row variables and quotation effects. For example, the stack effect of the `if` combinator is declared as follows:

```
( ..a ? true: ( ..a -- ..b ) false: ( ..a -- ..b ) -- ..b )
```

This combinator takes a condition `?` and two quotations `true`, `false` from the stack. Both quotations have the same stack effect `( ..a -- ..b)`. The row variables `..a`

and `..b` declare that the quotations may take any number of items from the stack and push any number of items onto the stack. Finally, the combinator returns `..b`, which is whatever the quotation returned.

Both, Factor and Forth are concatenative languages. That means that words are called in sequence and each word takes the output of the previous word as input. New words are defined as sequences of words. For example, a word that removes the second item from the stack could be defined as

```
: nip   ( x1 x2 -- x2 )   swap drop ;
```

We know the stack effects of `swap ( x1 x2 -- x2 x1 )` and `drop ( x -- )`. The stack effect of `nip` follows directly from combining their effects. The following sections explain how to derive stack effects in a generic way for words, quotations and combinators.

2 Notation

For the purpose of this document, we use mathematical notation for stack effects, where $-$ delimits inputs outputs. Stack items are denoted by name with single lowercase characters $a \ldots z$. The names by themselves are completely meaningless and interchangeable. Their purpose is to uniquely identify items within a single stack effect. Two items with the same name refer to the same object. Furthermore, we add explicit row variables to every definition, which represent the entire remaining stack. For example, $..a\ x$ and $..a\ y$ refer to equal stacks where only the top item is different. Whereas, $..a\ x$ and $..b\ x$ can (but don't need to) be two totally different stacks with the same top item.

Here are a few typical stack effects:

$$\text{identity}: \quad ..a - ..a \tag{1}$$
$$\text{unspecified}: \quad ..a - ..b \tag{2}$$
$$\text{drop}: \quad ..a\ x - ..a \tag{3}$$
$$\text{push}: \quad ..a - ..a\ x \tag{4}$$
$$\text{dup}: \quad ..a\ x - ..a\ x\ x \tag{5}$$
$$\text{swap}: \quad ..a\ x\ y - ..a\ y\ x \tag{6}$$

Quotation effects are written as an item name for the quotation, immediately followed by its stack effect in parentheses. For example, $f(..a - -..b)$ states that the item f is a quotation with an arbitrary stack effect.

Here are a few typical combinator effects:

$$\text{call}: \quad ..a\ f(..a - ..b) - ..b \tag{7}$$
$$\text{if}: \quad ..a\ c\ t(..a - ..b)\ f(..a - ..b) - ..b \tag{8}$$

3 Word Composition

New words are defined by concatenation of existing words. Consider the word `: add-two ( n -- n ) 2 + ;` which is composed of the words `2` and `+`. The former is a literal that pushes the value 2 on the stack, and the latter takes two

values from the stack and pushes their sum. The stack effect of 2 is $(..a\ -\ ..a\ n)$ and the stack effect of + is $(..b\ x\ y\ -\ ..b\ z)$. Their combined effect is $(..c\ m\ -\ ..c\ n)$, which says that an item is popped from the stack and replaced with a different item.

Deriving stack effects of word concatenations seems straight forward: For each word in sequence simply pop inputs from an abstract stack and push back outputs. Popping from an empty abstract stack generates new items. All items popped from the empty stack are the inputs of the derived effect, and the resulting stack represents the output.

$$\frac{\begin{array}{lll} ..a\ x\ y & - & ..a\ y\ x \\ & & ..b\ z \quad - \quad ..b \end{array}}{..a\ x\ y \qquad - \qquad ..a\ y} \tag{9}$$

4 Summary