Forth Semantics for Compiler Verification

Bill Stoddart, Campbell Ritchie, Steve Dunne

September 13, 2012

Abstract

Here we are interested in the semantics of Forth from the point of view of using Forth as a target language for a formally verified compiler for Ruth-R, a reversible sequential programming language we are currently developing. We limit out attention to those Forth operations and constructs which will be targetted by the Ruth-R compiler. To facilitate the comparison of meanings of source and target languages, we represent the semantics of Forth code by translation into a form which can be described using the "prospective value" semantics we use for Ruth-R.

1 Introduction

We are interested in the semantics of Forth from the point of view of using Forth as a target language for a formally verified compiler.

Our source language for this project is Ruth-R, an expressive reversible guarded command language for which we are currently constructing the syntax, semantics, and compiler. The semantics of Ruth-R are expressed in terms of the value some expression E can take after the execution of some program S, which we represent as $S \diamond E$. We call this the prospective value of E after executing S, and refer to the semantics as "prospective value semantics", or PV semantics for short.

The target language is RVM-Forth [11], a reversible version of Forth we have developed to explore the algorithmic possibilities of reversible languages.

Our approach will be to give a translation of each Forth operation into a form to which PV semantics may be applied. That is, we translate it into the form of a sequential programming language, with the stack appearing as a variable.

Our investigations also cover different representations of the same value. For example, RVM-Forth provides a generic set package, but for efficiency reasons we may wish to represent small sets using bit vectors.

A key design question for us will be how to model the Forth parameter stack, and we will compare two possible approaches: first as a sequence of cell values, (an approach previously investigated by P Knaggs[4]) whose entries are subsequently viewed in terms of interpretation functions; and secondly as a

"conceptual stack", a structure which is directly able to hold values of any type.

Our approach is also influenced by the fine grained nature of Forth semantics; the infix language assignment $x := x + 1$ will compile to Forth as `x 1 + to x`; and for compositionality we require separate semantic translations for each of `x, 1, +` and `to x` and the sequential composition of these to obtain the semantics of `x 1 + to x` . This makes it essential that the semantic description of the sequential composition of operations be expressed in a simple way.

Our approach is more detailed than the Pöial algebra of stack effects [7] but is less ambitious that some other previous formulations. We do not seek a technique for the complete description for the Forth virtual machine[6] or the operation of the Forth compiler [5], but we rather extend Forth with the data types and control structures required by Ruth-R, and then give semantic descriptions of these components; thus the work described here can be characterised as a shallow formulation. Its advantages are that is is axiomatic, fully compositional, and minimises the semantic distance between source and target languages.

Our general approach is conditioned by that we have adopted for expressing the semantics of reversible language. We borrow many techniques from the B-Method [1] as described in [12], and to simplify our theory presentation we use Hehner's conception of a "bunch" as the contents of a set.

The rest of the paper is structured as follows: section 2 deals with mathematical background and the axioms of our semantics; section 3 sketches an algebraic treatment of stack semantics; Section 4 considers two different models for the stack, and discusses assignment, literal values, and differing representations of the same data; section 5 defines the semantics of selection and iteration; section 6 discusses local variables; in section 7 we conclude.

2 Mathematical Preliminaries

2.1 Bunch Theory

Following Hehner[2], we give a mathematical meaning to the contents of a set, which we call a bunch: e.g. the contents of the set $\{1, 2\}$ is the bunch $1, 2$. We write $\sim A$ for the contents of set A, thus $\sim\{1, 2\} = 1, 2$.

The comma used in a bunch extension expression such as $1, 2, 5$ is now a mathematical operator, called bunch union. It is associative and commutative, and its precedence is just below than that of the expression connectives. It is associated with set union through the rule

$A \cup B = \{\sim A, \sim B\}$

Binary operations applied to bunches are lifted to apply pairwise, thus adding the bunches $0, 1$ and $2, 4$ yields $0+2, 0+4, 1+2, 1+4$. Note that we cannot write this sum as $0, 1 + 2, 4$ because bunch union (comma) has a lower precedence

than +; nor can we use standard brackets to enforce precedence, as we retain the use of brackets to express tuples. We can however bracket with $\sim\{..\}$, as in $\sim\{0,1\} + \sim\{2,4\} = 2,4,3,5$

A bunch A is a sub-bunch of B if each element of A is an element of B. We write this as $A : B$. Sub-bunches are related to subsets by the rule:

$$A : B \Leftrightarrow \{A\} \subseteq \{B\}$$

The guarded bunch $p \longrightarrow E$ is equal to the empty bunch, *null*, if predicate p is false, and otherwise equal to E. The preconditioned bunch $p \mid E$ is equal to the improper bunch $\perp$ if p is false, and otherwise equal to E. The improper bunch expresses complete ignorance about a value, extending to a suspicion that a computation supposed to produce a value might have crashed.

We use a typed set theory based on the axiomatic approach used in the B-Method. We assume the availability of any required given sets, including the integers, and we are able to produce from these new maximal sets by the use of set product and powerset operations. These maximal sets act as types. We can generate new sets by set comprehensions of the form

$$\{x \mid x \in X \wedge P \bullet E\}$$

where expression E is of a fixed type, so that our set comprehensions can only produce homogeneous sets.

We similarly restrict our attention to bunches which are homogeneous; these have the same type as their elements, thus $1, 2$ is of type integer, which we can write as $1, 2 \in \mathbb{Z}$.

Analogously to set comprehension, we can write the bunch comprehension:

$$\oint x \bullet E$$

Here, although the type of x is not given explicitly it has to be implicit from an examination of the expression E. The result consists of all values that E can take as x ranges over its type. For example the bunch $10, 20$ can be written as the bunch comprehension $\oint x \bullet x : 1, 2 \longrightarrow 10 * x$.

2.2 Prospective Value Semantics

We are formalising a reversible language with a choice construct used to express both non-determinism and provisional choices subject to backtracking. We write $S \diamond E$ for the values expression E could take were program S to be executed. We have the following rules which define $S \diamond E$ over the essential semantic components of the language:

Name	Rule	Condition
skip	$skip \diamond E \;=\; E$	
assignment	$x := F \diamond E \;=\; (\lambda\, x \bullet E)F$	
pre-condition	$P \mid S \diamond E \;=\; P \mid\!\!\mid (S \diamond E)$	
choice	$S \,[]\, T \diamond E \;=\; (S \diamond E)\,,\,(T \diamond E)$	
guard	$P \overset{[]}{\longrightarrow} S \diamond E \;=\; P \longrightarrow (S \diamond E)$	
seq comp	$S \,;\, T \diamond E \;=\; S \diamond T \diamond E$	
local variable	$var\, x \,.\, S \diamond E \;=\; \oint x \bullet S \diamond E$	x not free in E

The precedence of $\diamond$ is below that of programming connectives, whose precedence, in descending order, is $:=$, $\overset{[]}{\longrightarrow}$, $[]$, $;$, $|$ The large equals $=$ has the same meaning as "=" but a very low precedence: we require it when discussing equality in the context of programs as the precedence of the standard equal sign is above that of the program connectives.

Our use of bunches enables us to express the effects of choice and sequential composition in a homogeneous manner, and to describe non-determinism. This case is more fully argued in [8].

The attentive reader will note that these semantic components do not include selection and iteration constructs. These are handled by means of choice and guard. e.g

if g **then** S **else** T **end**

is expressed as:

$g \overset{[]}{\longrightarrow} S \;[]\; \neg\, g \overset{[]}{\longrightarrow} T$

This decomposition was first proposed by Hehner [2, 3] who used predicative semantics. Abrial adapted it to predicate transformer semantics for use in in the B-Method [1]. A description of our approach using prospective value semantics is given in [10]. We can extend this to probabilistic programs [8], and express preference within provisional choice [9].

3 Prospective values and stack algebra

We will consider two different ways a stack may be modelled in terms of an underlying representation within our mathematical world of typed set theory, before choosing our preferred representation. However, we will first deal *algebraically* with some basic stack manipulations. This will avoid some repetition, as the algebraic treatment will be the same for either model.

We represent an empty stack by ε. If s is a stack state, let $s \sqcup x$ be the new stack state obtained by pushing x.

If s is a non-empty stack, $drop(s)$ will be the new stack obtained by dropping the top item. We thus have

$drop(s \sqcup x) \;=\; s$

Let $top(s)$ be the top element of a non-empty stack s, and $next(s)$ the second from top element of a stack s which has at least two elements. Thus:

$top(s \sqcup x) = x$ and $next(s \sqcup x \sqcup y) = x$

The function *swap* takes a stack and returns the new stack obtained by swapping the top two elements. Thus

$swap(s \sqcup x \sqcup y) = s \sqcup y \sqcup x$

We define the semantics of the Forth `SWAP` operation, which unlike the function *swap*, acts on a particular stack (which we just call *stack*):

$$[\![\texttt{SWAP}]\!]^{\mathcal{F}} = \quad depth(stack) \geq 2 \mid stack := swap(stack)$$

Here, the notation $[\![\texttt{SWAP}]\!]^{\mathcal{F}}$ encloses the Forth code being discussed in the semantic brackets $[\![\ldots]\!]^{\mathcal{F}}$. In general, if `S` is Forth code, $[\![\texttt{S}]\!]^{\mathcal{F}}$ will represent its translation into a form whose meaning can be expressed in PV semantics.

With the above semantic definition of `SWAP` we bring a Forth stack manipulation within the scope of PV semantics by treating it as an assignment. The semantics of `SWAP` given above also tells us that its frame (the list of variables it may alter) contains just *stack*. Also, by PV rules for pre-condition and assignment:

$$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond stack = \quad depth(stack) \geq 2 \mid swap(stack)$$

We can apply similar treatments to other stack manipulation operations.

$[\![\texttt{DROP}]\!]^{\mathcal{F}} = \quad depth(stack) \geq 1 \mid stack := drop(stack)$
and hence
$[\![\texttt{DROP}]\!]^{\mathcal{F}} \diamond stack = \quad depth(stack) \geq 1 \mid drop(stack)$

For `NIP` we introduce an auxiliary function *nip* such that for any stack s and items x, y we have $nip(s \sqcup x \sqcup y) = s \sqcup y$, allowing us to define the semantics of the Forth operation `NIP` as:

$$[\![\texttt{NIP}]\!]^{\mathcal{F}} = \quad stack := depth(stack) \geq 2 \mid stack := nip(stack)$$

and hence

$$[\![\texttt{NIP}]\!]^{\mathcal{F}} \diamond stack = \quad depth(stack) \geq 2 \mid nip(stack)$$

In addition to giving the meaning of individual Forth operations, we also need to express the meaning of Forth's program connectives. The first we require is sequential composition:

$$[\![\texttt{S T}]\!]^{\mathcal{F}} = \quad [\![\texttt{S}]\!]^{\mathcal{F}} ; [\![\texttt{T}]\!]^{\mathcal{F}}$$

We are now in a position to prove a semantic equality. We will show that `NIP` is equivalent to `SWAP DROP`. We define two Forth programs to be equal if their semantic representations are equal. i.e. for Forth programs A and B:

$$A = B \mathrel{\widehat{=}} [\![\texttt{A}]\!]^{\mathcal{F}} = [\![\texttt{B}]\!]^{\mathcal{F}}$$

and we define their semantic representation to be equal if they have the same frame and the same PV effect over the variables of that frame (or, equivalently, the same PV effect over an arbitrary expression).

We will show that: $[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \;=\; [\![\texttt{NIP}]\!]^{\mathcal{F}}$

Since the frame of $[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}}$ and the frame of $[\![\texttt{NIP}]\!]^{\mathcal{F}}$ are both *stack*, we can show the required equality by showing

$[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack} \;=\; [\![\texttt{NIP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$

$[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$
$=$ — Forth seq comp
$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \,;\, [\![\texttt{DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$
$=$ — PV seq comp
$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond [\![\texttt{DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$
$=$ — semantics of *DROP*
$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond \mathit{depth}(\mathit{stack}) \geq 1 \mid \mathit{stack} := \mathit{drop}(\mathit{stack})$
$\diamond \mathit{stack}$ $=$ — PV pre-cond
$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond \mathit{depth}(\mathit{stack}) \geq 1 \mid \mathit{stack} := \mathit{drop}(\mathit{stack})$
$\diamond \mathit{stack}$ $=$ — PV assignment
$[\![\texttt{SWAP}]\!]^{\mathcal{F}} \diamond \mathit{depth}(\mathit{stack}) \geq 1 \mid \mathit{drop}(\mathit{stack})$
$=$ — semantics of *SWAP*
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{stack} := \mathit{swap}(\mathit{stack})$
$\diamond \mathit{depth}(\mathit{stack}) \geq 1 \mid \mathit{drop}(\mathit{stack})$
$=$ — PV pre-condition
$\mathit{depth}(\mathit{stack}) \geq 2 \mid (\mathit{stack} := \mathit{swap}(\mathit{stack})$
$\diamond \mathit{depth}(\mathit{stack}) \geq 1 \mid \mathit{drop}(\mathit{stack}))$
$=$ — by assignment
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{depth}(\mathit{swap}(\mathit{stack})) \geq 1$
$\mid \mathit{drop}(\mathit{swap}(\mathit{stack}))$
$=$ — simplifying pre-cond
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{drop}(\mathit{swap}(\mathit{stack}))$

Now for $\mathit{depth}(\mathit{stack}) \geq 2$ there will be some stack s and items x, y such that $\mathit{stack} = s \sqcup x \sqcup y$ and hence:

$[\![\texttt{SWAP DROP}]\!]^{\mathcal{F}} \diamond \mathit{stack} \;=$
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{drop}(\mathit{swap}(s \sqcup x \sqcup y) \;=$
— applying function *swap*
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{drop}(s \sqcup y \sqcup x) \;=$
— applying function *drop*
$\mathit{depth}(\mathit{stack}) \geq 2 \mid s \sqcup y \;=$
— from property of *nip*
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{nip}(s \sqcup x \sqcup y) \;=$
— equality: $\mathit{stack} = s \sqcup x \sqcup y$
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{nip}(\mathit{stack}) \;=$
— assignment introduction
$\mathit{depth}(\mathit{stack}) \geq 2 \mid \mathit{stack} := \mathit{nip}(\mathit{stack})$
$\diamond \mathit{stack} \;=$
— semantics of *NIP*
$[\![\texttt{NIP}]\!]^{\mathcal{F}} \diamond \mathit{stack}$

4 Modelling the stack

Modelling a stack which may contain items of any type is a challenge in our semantics, where we rely on a typed set theory which only admits homogeneous sets. We consider two possible approaches. In the first the stack is modelled as a sequence of raw cell values, and we use interpretation functions to tell us what is represented by these cells. In the second we model a stack state as an n-tuple, taking advantage of the fact that different elements of a tuple need not be homogeneous, and we describe a conceptual stack of possibly non-homogeneous mathematical objects.

4.1 Modelling the stack as a sequence of cells

In this section we introduce a stack modelling technique which we will subsequently reject in favour of a more abstract approach. The section is included mainly to show some difficulties that arise from this approach, and can be omitted without affecting the readers understanding of the rest of the paper.

We might model the stack as a finite sequence of cells, where a cell is a function defined on 32 bit machines as:

$$CELL = 0..31 \pfun BIT$$

A stack is a finite sequence of cells. We introduce a constant, the set of all stacks, which we call $STACK$.

$$STACK = fseq(CELL)$$

and we introduce a variable *stack* to represent the parameter stack, with the invariant property:

$$stack \in STACK$$

With this model, pushing an element onto a stack is simply a matter of appending an element to a sequence:

$$s \in STACK \land c \in CELL \Rightarrow s \backslash c = s \conc \langle c \rangle$$

Functions which perform calculations on a stack will interpret its cell values in a particular way. For example the operation + might interpret them as signed numbers. Let $num \in CELL \to -2^{31} .. (2^{31} - 1)$ be the "interpretation function" giving the signed integer value associated with the cell under a two's complement representation. The inverse of *num* is also a function, such that for any signed integer n in the representable range, $num^{-1}(n)$ will be the cell that represents n.

To describe the semantics of the Forth word +, we first define a function *plus* which acts on any stack of depth ≥ 2 and adds the top two elements: we see in its definition the explicit interpretation of cell values as numbers:

$$
\begin{array}{l}
plus \in STACK \pfun STACK \\
dom(plus) = \{ s \mid s \in STACK \land card(s) > 1 \} \\
s \in STACK \land c_1, c_2 : CELL \Rightarrow \\
\quad plus(s \frown \langle c_1, c_2 \rangle) \;=\; s \frown \langle\, num^{-1}(num(c_1) + num(c_2)) \,\rangle
\end{array}
$$

The attentive reader will not that, in defining *plus*, we have not been careful about what happens if the application of + yields a value outside the expressible range. This is because we avoid any responsibility for the the value provided by such an application by including an appropriate clause in the pre-condition in the semantics of the Forth operation +:

$$
\begin{array}{l}
[\![+]\!]^{\mathcal{F}} \;=\; num(next(stack)) + num(top(stack)) \in ran(num) \land depth(stack) \geq 2 \\
\mid\; stack := plus(stack)
\end{array}
$$

The above example shows an approach we can take when the interpretation functions we are using are constant. However, for some operations the results will be data structures held in the heap and referenced by pointers on the stack. Consider, for example, the following expression in RVM-Forth which represents $\{100, 200\} \cup \{300\}$:

```
INT { 100 , 200 , }  INT { 300 , }  ∪
```

As each set is evaluated, we obtain, on the stack, a pointer to a structure on the heap. Let *set* be the interpretation function that maps the cell values of these pointers to their corresponding sets. We note that *set* is a variable which changes with every new set that is produced. Also, the pointer values are not explicitly defined: we can only say that some suitable pointer is provided, which we must do by introducing it with an existential quantifier. One possible description of $[\![\cup]\!]^{\mathcal{F}}$ is:

$$
\begin{array}{l}
depth(stack) \geq 2 \land top(stack) \in dom(set) \land next(stack) \in dom(set) \Rightarrow \\
\exists\, c \bullet [\![\cup]\!]^{\mathcal{F}} = \\
stack := drop(drop(stack)) \sqcup c \quad \| \\
set := set \cup \{ (c,\, set(next(stack)) \cup set(top(stack)) \}
\end{array}
$$

The reader will note that instead of an explicit definition of the semantics of set union, we have obtained an implicit description in which the term $[\![\cup]\!]^{\mathcal{F}}$ has become a fully fledged mathematical object. This is not a situation we relish: it imposes some additional responsibilities to show the mathematical validity of such terms, and does not provide for equational reasoning.

These concerns, along with the entailment of irrelevant details concerning heap pointers, motivate us to investigate an approach in which we model a non-homogeneous stack of conceptual mathematical objects.

4.2 Modelling the conceptual stack

We use a typed set theory whose axioms are given in [1]. We can define some "given sets" and from these we form new sets by means of the powerset and set product operations. Set comprehension allows us to describe subsets of these

given and constructed sets.

We wish to model a "conceptual stack", which may include different types of mathematical object. The restrictions of our type theory rule out the use of a sequence as the modelling vehicle, so we turn to tuples.

In a formalism which provides n-tuples, we could model a stack containing $a\ b\ c$ with the 3-tuple (a, b, c). However, our formalism provides only ordered pairs, and when we write (a, b, c) we obtain an ordered pair $(\,(a, b),\ c\,)$.

Thus we cannot just use (a, b, c) to model the stack containing a, b and c: since ordered pairs are themselves a possible stack item, there would be no way to distinguish a 3 item stack containing a, b and c from a two item stack containing (a, b) and c (we note that tuple construction is left associative). Furthermore, we would have no way to represent a stack of 0 or 1 items.

We can, however, construct some special ordered pairs which will model n-tuples, or stacks. We require an arbitrary constant value to represent an empty stack, and we call this ε.

For a stack s we define the act of pushing an element s using tuple construction as:

$$s \sqcup x \mathrel{\widehat{=}} (s, x)$$

We make use of the ordered pair notations:

$$\mathit{first}(x, y) \mathrel{\widehat{=}} x, \quad \mathit{second}(x, y) \mathrel{\widehat{=}} y$$

And thus we can define the function that yields the top element of a stack:

$$\mathit{top}(s) \mathrel{\widehat{=}} \mathit{second}(s)$$

If the function *top* is applied to an empty stack we obtain, in our particular theory of partial function application, the empty bunch. However, we will always impose pre-conditions on operations in order to wash our hands of any responsibility for such an application.

We also have:

$$\mathit{drop}(s) \mathrel{\widehat{=}} \mathit{first}(s)$$
$$\mathit{next}(s) \mathrel{\widehat{=}} \mathit{top}(\mathit{drop}(s))$$
$$\mathit{depth}(s) \mathrel{\widehat{=}} s = \varepsilon \longrightarrow 0,\ s \neq \varepsilon \longrightarrow \mathit{depth}(\mathit{drop}(s)) + 1$$

At this point the diligent reader may wish to construct the stack s where $s = \varepsilon \sqcup a \sqcup b \sqcup c$ and evaluate $\mathit{top}(s)$, $\mathit{drop}(s)$, $\mathit{next}(s)$ and $\mathit{depth}(s)$.

Since we cannot characterise s as a stack by saying it is a member of some type (which for us is a maximal set) we introduce a unary predicate *IsStack*, with the following defining properties:

$$\mathit{IsStack}(\varepsilon)$$
$$\mathit{IsStack}(s) \Rightarrow \mathit{IsStack}(s \sqcup x)$$
and nothing else is a stack.

We now return to the semantics of set union which gave us some trouble when

modelling the stack as a sequence of cells. We begin, as usual, by the description of an auxiliary function.

$$IsStack(s) \wedge \exists\ T \bullet x \subseteq T \wedge y \subseteq T \Rightarrow union(s \frown x \frown y) = s \frown x \cup y$$

Then we can give an explicit definition of the semantics of Forth set union:

$$[\![\cup]\!]^{\mathcal{F}} = \exists\ T \bullet top(stack) \subseteq T \wedge next(stack) \subseteq T \mid stack := union(stack)$$

From now on we take the conceptual stack as our model.

4.3 On differing representations of the same data

Within a computer system we may have different representations of the same data. A string could be a counted string or an ascii zero string. A sequence of n elements has a representation in the RVM sets package in terms of its graph, i.e. as a set of ordered pairs, but could also be represented by an n element array. A function could be represented as an operation or, in passive form, by its graph, or, if it is a sequence, as an array.

To show how such variations in representation may be handled using the conceptual stack we consider the example of bitsets, which can represent subsets of 0..31 according the the bit settings in a 32 bit cell.

We define a function $bits2set$ whose domain is the subsets of 0..31 and which maps its argument to the corresponding bitset representation for that set. The semantics of the corresponding Forth operation, `Bits2Set` is:

$$[\![\texttt{Bits2Set}]\!]^{\mathcal{F}} = depth(stack) > 0 \wedge top(stack) \subseteq 0..31 \mid stack := drop(stack) \frown bits2set(top(stack))$$

4.4 Literal values, variables, and assignments

Some RVM-Forth literal expressions are written in non-standard notation. For example the set $\{10, 20\}$ would be written in RVM-Forth as `INT { 10 , 20 , }`. In the following we use $[\![\texttt{L}]\!]^{\mathcal{L}}$ to represent the translation of the RVM-Forth literal expression `L` into standard notation, but we do not give this translation in detail.

We express the semantics of a literal expression `L` as:

$$[\![\texttt{L}]\!]^{\mathcal{F}} = stack := stack \frown [\![\texttt{L}]\!]^{\mathcal{L}}$$

Let `x` be a variable (i.e. a Forth `VALUE`). We assume the existence of a corresponding variable in the mathematical world, which we write, in mathematical typeface, as x. The semantic representations for `x` are:

$$[\![\texttt{x}]\!]^{\mathcal{F}} = stack := stack \frown x$$

$$[\![\texttt{to x}]\!]^{\mathcal{F}} = depth(stack) > 0 \mid stack := drop(stack) \parallel x := top(stack)$$

5 Selection and Iteration

Instead of a condition test which is syntactically connected to the selection statement, the Forth `IF` structure uses the value at the top stack item, assumed to hold the result of a previous test.

$[\![\textbf{ if } \texttt{S} \textbf{ else } \texttt{T} \textbf{ then }]\!]^{\mathcal{F}} \;=$
$var\,\tau \,.\, \tau \;=\; top(stack) \xrightarrow{[\![]} stack := drop(stack)$;
if $\tau \neq 0$ **then** $[\![\texttt{S}]\!]^{\mathcal{F}}$ **else** $[\![\texttt{T}]\!]^{\mathcal{F}}$ **end**

For iteration, we restrict our attention to the structure:

$[\![\texttt{BEGIN G WHILE S REPEAT}]\!]^{\mathcal{F}}$

since this most closely resembles the classic while loop representation used in the formal semantics of sequential programming; to obtain the classical loop interpretation we must impose the following restrictions:

$frame([\![\texttt{G}]\!]^{\mathcal{F}}) \;=\; stack$

$[\![\texttt{G}]\!]^{\mathcal{F}} \diamond stack \;=\; stack \llcorner E$ for some E

$[\![\texttt{S}]\!]^{\mathcal{F}} \diamond stack \;=\; stack$

these assumptions allow us to transcribe the loop into a standard sequential programming representation $[\![\texttt{BEGIN G WHILE S REPEAT}]\!]^{\mathcal{F}} \;=$
while $top([\![\texttt{G}]\!]^{\mathcal{F}} \diamond stack) \neq 0$ **do** $[\![\texttt{S}]\!]^{\mathcal{F}}$ **end**

6 Local variables

Local variables in RVM-Forth operations are used to capture operation arguments and other local variables whose initial values are provided after execution of an operation has begun. An example of local variable syntax in RVM-Forth is:

```
: T (: x y :) 100 VALUE u  200 VALUE v  ...  ;
```

Here `x` and `y` will be initialised with the value of the next and top stack items when `T` is invoked, and `u` and `v` will be initialised with the values 100 and 200.

Our semantics of local variables will be described in terms of initial values which are taken from the Forth parameter stack. However, we wish to accommodate an implementation technique which leaves the values where they are, and accesses them by indexing into the Forth parameter stack. This provides more efficient code on register based architectures, but we need to follow two rules to ensure correct usage.

The first is that an operation whose arguments are instantiated as local variables must not access any stack values held below these arguments. Here is an example of declaration of local parameters that breaks that rule:

```
: SURFA ( a:n b:n c:n -- 2*(a*b + a*c + b*c)) (: b c :) ...
```

here, the value supplied for argument `a` needs to be accessed from the stack, but due to our implementation technique of leaving local values on the parameter stack, it will be hidden below the values supplied for `b` and `c`. To allow `a` to be accessed as a local variable we could begin our definition with:

```
: SURFA ( a:n b:n c:n -- 2*(a*b + a*c + b*c)) (: a b c :) ...
```

The second rule is that the code between the parameter list and any instance of `VALUE` must have a stack signature of the form `( -- x )`. In terms of our Forth semantics this means that, where `S` represents the code between the end of the parameter stack and the relevant occurrence of `VALUE`, then for some expression E:

$$[\![\texttt{S}]\!]^{\mathcal{F}} \diamond stack \;=\; stack \llcorner E$$

Two illustrations will serve as justification: first we modify a previous example so it breaks the rule:

```
 : T (: x y :) 100 200 VALUE u VALUE v  .. ;
```

We now have code between the end of the parameter list and an occurrence of `VALUE` which adds two items to the stack. According to the semantics we will give, this will initialise `u` to 200 and `v` to 100, but in our implementation, which leaves local variables on the stack, it will still initialise `u` to 100 and `v` to 200. This happens as follows: the Forth compilation of `(: x y :)` sets up the top two stack cells to be a stack frame for variables `x` and `y`. The declarations `VALUE u` and `VALUE v` each extend the stack frame by one cell, associating with these cells the names `u` and `v`. Thus when 100 and 200 appear on the stack at run time, they provide the initial values for `u` and `v` respectively. Our rule ensures the synchronisation of compile time and run time activity.

A second way in which the rule can be broken is illustrated by the following code:

```
: T (: x :) DUP VALUE y .. ;
```

Here the programmer is making use of his knowledge of an implementation which leaves local variables on the parameter stack. Where this is the case, this example will initialise `y` with the same value as `x`. However, this is not formally correct as our semantics will insist that the initial value for `x` has been removed from the stack.

6.1 On Scope

The semantics of local variables requires an idea of the following code, indicated by `S` in the rules below, within which the local variable is in scope. This scope is, by default, till the end of the Forth definition in which the local is declared, and otherwise is controlled by the use of scoping brackets.

6.2 Formal semantics of locals

The last local declared in the list of operation parameters is initialised from the top of stack

$$[\![(:\ \ldots\ \texttt{x}\ :)\ \texttt{S}]\!]^{\mathcal{F}} \ =$$
$$var\ x\ .\ x = top(stack) \xrightarrow{[]} stack := drop(stack)\ ;\ [\![(:\ \ldots\ :)\ \texttt{S}]\!]^{\mathcal{F}}$$

This above rule is applied until we obtain an empty parameter list, which has the following empty frame semantics:

$$[\![(:\ :)\ \texttt{S}]\!]^{\mathcal{F}} \ = \ \ [\![\texttt{S}]\!]^{\mathcal{F}}$$

Local variables declared following the operation's parameter list are formally initialised from the stack:

$$[\![\texttt{VALUE x}]\!]^{\mathcal{F}} \ = \ \ var\ x\ .\ x = top(stack) \xrightarrow{[]} stack := drop(stack)$$

7 Conclusions

We have chosen an approach to Forth semantics which is prompted by the needs of our particular research, which is to lay foundations for formal verification of a compiler that uses Forth as its target language. Our source language, Ruth-R, is an expressive reversible language with a prospective values semantics. To allow the most direct comparison of meaning between a Ruth-R program and the corresponding RVM-Forth program produced by the Ruth-R compiler, we provide a semantics that is based on expressing the meaning of Forth operations and programming constructs in terms of PV semantics.

References

[1] J-R Abrial. *The B Book*. Cambridge University Press, 1996.

[2] E C R Hehner. *A Practical Theory of Programming*. Springer Verlag, 1993. Latest version available on-line.

[3] E R Hehner. Retrospective and Prospective for Unifying Theories of Programming. In S. E. Dunne and W Stoddart, editors, *UTP2006 The First International Symposium on Unifying Theories of Programming*, number 4010 in Lecture Notes in Computer Science, 2006.

[4] P. K. Knaggs and W. J. Stoddart. The Cell Type. In *University of Rochester Forth Conference on Automated Instruments, The Journal of Forth Application and Research (JFAR)*, June 1991.

[5] J F Power and D Sinclair. A Formal Model of Forth Control Words in the Pi-Calculus. *Journal of Universal Computer Science*, 10.9, 2004.

[6] W. J. Stoddart. An Event Calculus Model of the Forth Programming System. In *12th EuroForth Conference Proceedings*, October 1996. Note that the downloadable document contains the pages in reverse order.

[7] W. J. Stoddart and P. K. Knaggs. Type Inference in Stack Based Languages. *Formal Aspect of Computing*, 5(4):289–298, 1993.

[8] W J Stoddart and F Zeyda. A Unification of Probabilistic Choice within a Design-based Model of Reversible Computation. *Formal Aspect of Computing*, 2007. Published on line, DOI 10.1007/s00165-007-0048-1.

[9] W J Stoddart, F Zeyda, and S Dunne. Preference and non-deterministic choice. In *ICTAC'10. Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, volume 6255 of *LNCS*, 2010.

[10] W J Stoddart, F Zeyda, and A R Lynas. A Design-based model of reversible computation. In *UTP'06, First International Symposium on Unifying Theories of Programming*, volume 4010 of *LNCS*, June 2006.

[11] W J Stoddart, F Zeyda, and A R Lynas. A virtual machine for supporting reversible probabilistic guarded command languages. *ENTCS*, 253, 2010. Extended version of a paper presented at RC2009.

[12] F Zeyda. *Reversible Computations in B*. PhD thesis, University of Teesside, 2007.