Formal Aspects of Computing (1992) 3: 1–000

Type Inference in Stack Based Languages

Bill Stoddart and Peter J. Knaggs

School of Computing and Mathematics, University of Teesside,
Middlesbrough, Cleveland, U.K.

Keywords: Formal Aspects, Stack Based Languages, Semantic Model Language, Algebras, Type Inference, Forth

Abstract. We consider a language of operations which pass parameters by means of a stack. An algebra over the set of type signatures is introduced, which allows the type signature of a program to be obtained from the type signatures of its constituent operations.
Although the theories apply in principle to any stack based language, they have been evolved with particular regard to the proposed ANSI Standard Forth language, which is currently implemented in a type free manner. We hope this work will stimulate an interest in Forth amongst those applying algebraic techniques in software engineering, and we hope to lay the theoretical foundations for implementing practical type checkers to support Forth.

1. Introduction

Stack based languages are important both as intermediate target languages for compilers and as application languages in their own right (Forth [Bro87], Reverse Polish Lisp [Wic88], Postscript etc).

The formalisms in this paper have been evolved with particular regard to Forth. Forth is an unusual "language" in that it is based on a mapping between words (as text strings) and operations. The meaning and syntactic form of the language is captured in the semantics of the operations, which provide a low level stack based architecture and also perform all system functions including

Correspondence and offprint requests to: Bill Stoddart, School of Computing and Mathematics, University of Teesside, Middlesbrough, Cleveland, TS1 3BA, UK. (Email: bill@scm.tees.ac.uk)

the parsing of input text, text interpretation, compilation, error reporting, and operating system functions such as multi-tasking (albeit in a simple fashion).

Forth also has some importance as an application language. Some chip manufacturers produce high speed micro controllers optimised for running Forth (for example the Harris RTX2000); IBM released its first Forth product, IBM CADFORTH, in 1991; three of the four telescopes carried by the most recent flight of the Columbia space shuttle were controlled by Forth based systems; Hewlett Packard have used Forth as the implementation language for their top end calculators which combine symbolic and numeric processing; all Sun systems have a boot ROM coded in Forth, and Forth now has a draft ANSI standard [ANS91].

Although Forth is always implemented in a type free manner (i.e. with no type checking at all), Forth programmers must keep track of the types they are using and select appropriate operations for these types if their program is to perform correctly. In this paper we show that type inference for a stack based language is not difficult to formalise, and can deal with multi-typed and type free operations as these are used in Forth.

We consider Forth as a "language" of "words" which are associated with operations that obtain input arguments from a stack and return output arguments to the same stack.

The types of these arguments are specified by a type signature. We write $a\ b \xrightarrow{\text{ret}} c$ to represent the signature of an operation that requires two input arguments, of types a and b, and returns one argument of type c. For both input and output arguments the top of the stack is shown to the right.

Suppose an operation with type signature $a\ b \xrightarrow{\text{ret}} c$ is followed by an operation with type signature $d\ c \xrightarrow{\text{ret}} a\ a$. We write

$$(a\ b \xrightarrow{\text{ret}} c)(d\ c \xrightarrow{\text{ret}} a\ a)$$

to denote the combined signature of the first operation followed by the second.

We can reduce this combined signature as follows. First note that the argument left on the top of the stack by the first operation is of type c, and that this is the argument required from the top of the stack by the second operation. The types match and we can cancel them:

$$(a\ b \xrightarrow{\text{ret}} c)(d\ c \xrightarrow{\text{ret}} a\ a) = (a\ b \xrightarrow{\text{ret}})(d \xrightarrow{\text{ret}} a\ a)$$

We now have a form in which the first signature does not supply any arguments to the second, so the argument of type d required by the second operation must be present on the stack before the first operation is executed. Therefore we can write

$$(a\ b \xrightarrow{\text{ret}})(d \xrightarrow{\text{ret}} a\ a) = (d\ a\ b \xrightarrow{\text{ret}} a\ a)$$

We can check these operations by means of a stack trace, which starts with arguments of type $d\ a$ and b on the stack and terminates with arguments of type $a\ a$.

Operation Signature	Stack	
	$d\ a\ b$	
$a\ b \xrightarrow{\text{ret}} c$	$d\ c$	pop types a, b push type c
$d\ c \xrightarrow{\text{ret}} a\ a$	$a\ a$	

The initial formulation of a stack based type signature algebra for Forth was

presented by Jaanus Pöial at Euro Forml 90, the annual European conference of Forth users [Pöi90a]. His algebraic theory draws on work by Nivat & Perrot [NP70]. The current paper adds rules for type variables, and shows how these allow the specification of type signatures for low level type free operations. We also describe the specification of multi-type operations and show how these allow the formulation of type signature inference rules for Forth program control structures (this latter topic also having been formulated, though in a different way, by J. Pöial [Pöi90b] [Pöi91]).

2. Types and Signatures

If a is a type ρa is also a type, and represents a pointer to an item of type a. Application of ρ is right associative.

In addition to signatures of the form $s \xrightarrow{\text{ret}} t$, there is a distinguished signature, denoted by ϕ, which represents a type clash. For example if a, b, c, d are types with $b \neq c$, then

$$(a \xrightarrow{\text{ret}} b)(c \xrightarrow{\text{ret}} d) = \phi$$

since the first signature supplies an argument of type b and the second requires an argument of type c.

We use $\mathcal{T}$ to represent the set of all types and $\mathcal{S}$ to represent the set of all signatures. The relationship between $\mathcal{S}, \mathcal{T}$ and the atomic types from which they are constructed may be expressed in the style of Z's free type notation as

$$\mathcal{T} ::= Atom << ident >> \mid \rho << \mathcal{T} >>$$

$$\mathcal{S} ::= \phi \mid \_ \xrightarrow{\text{ret}} \_ << \text{fseq}\,\mathcal{T} \times \text{fseq}\,\mathcal{T} >>$$

The syntax for type expressions used in this paper, however, is based partly on the notation used by Forth programmers, so that we write, for example $a \xrightarrow{\text{ret}} b\ c$, or when using just the ascii character set `a -- b c` rather than $(< a >, < b, c >)$.

3. Words and Signatures

Δ represents the set of words in the Forth language.

$$sig : \text{fseq}\,\Delta \nrightarrow \mathcal{S}$$

is a partial function mapping a sequence of Forth words to the signature of the operation they define.

We write a sequence of Forth words as words separated by spaces, for example we write `DUP NEGATE MAX` rather than $< \texttt{DUP}, \texttt{NEGATE}, \texttt{MAX} >$.

Given $p, q : \Delta$, we write $sig(p)$ to denote sig applied to the sequence containing just the word p. We use $sig(p) * sig(q)$ or just $sig(p)sig(q)$ to represent the combined type signatures.

We assume that sig is a homomorphism, so that

$$sig(\ p\ q\ ) = sig(p)sig(q)$$

4. The Sequential Composition of Simple Type Signatures

We use finite sequences of types to model the types of data items held on a stack. The last element in the sequence corresponds to the top of the stack.

Given $s_1, s_2, t_1, t_2 : \mathrm{fseq}\,\mathcal{T}$ and $x, y : \mathcal{T}$ we present a set of rules for reducing the sequential composition of two type signatures to a single type signature.

1. $\#s_2 = 0 \Rightarrow (s_1 \overset{\text{ret}}{\rightarrow} s_2)(t_1 \overset{\text{ret}}{\rightarrow} t_2) = t_1 \frown s_1 \overset{\text{ret}}{\rightarrow} t_2$
   Example $sig(p)sig(q) = (a\ b \overset{\text{ret}}{\rightarrow})(c \overset{\text{ret}}{\rightarrow} d) = c\ a\ b \overset{\text{ret}}{\rightarrow} d$
   Here p takes arguments of types a and b from the stack and returns no arguments. The argument of type c required by q must be on the stack before p is executed.
2. $\#t_1 = 0 \Rightarrow (s_1 \overset{\text{ret}}{\rightarrow} s_2)(t_1 \overset{\text{ret}}{\rightarrow} t_2) = s_1 \overset{\text{ret}}{\rightarrow} s_2 \frown t_2$
   Example $sig(p)sig(q) = (a \overset{\text{ret}}{\rightarrow} b)(\overset{\text{ret}}{\rightarrow} c) = (a \overset{\text{ret}}{\rightarrow} b\ c)$
3. $x \neq y \Rightarrow (s_1 \overset{\text{ret}}{\rightarrow} s_2 \frown x)(t_1 \frown y \overset{\text{ret}}{\rightarrow} t_2) = \phi$
   Example $sig(p)sig(q) = (a \overset{\text{ret}}{\rightarrow} a\ b)(b\ c \overset{\text{ret}}{\rightarrow} d) = \phi$
   Here p leaves an item of type b on the top of the stack and q requires an item of type c to be on the top of the stack. This causes a type clash. We take a simple view of types in which a clash occurs unless the types match exactly. Thus for the moment we do not permit sub types, but we will see how these can be handled later.
4. $(s_1 \overset{\text{ret}}{\rightarrow} s_2 \frown x)(t_1 \frown x \overset{\text{ret}}{\rightarrow} t_2) = (s_1 \overset{\text{ret}}{\rightarrow} s_2)(t_1 \overset{\text{ret}}{\rightarrow} t_2)$
   Example $sig(p)sig(q) = (a \overset{\text{ret}}{\rightarrow} b)(a\ b \overset{\text{ret}}{\rightarrow} c) = (a \overset{\text{ret}}{\rightarrow})(a \overset{\text{ret}}{\rightarrow} c)$
   This rule covers the case in which the top stack item supplied by the first operation is the same type as the top stack item required by the second. In this case we can "cancel" the two items.

The composition of a type signature $(s_1 \overset{\text{ret}}{\rightarrow} s_2)(t_1 \overset{\text{ret}}{\rightarrow} t2)$ which does not give a type clash requires n steps where

$$n = 1 + min(\#s_2, \#t_1)$$

To see this note that rule 4 can be used to reduce the length of s_2 and t_1 by one, and whenever one of these sequences becomes empty we can complete the reduction by one application of rule 1 or rule 2.

5. The algebra of type signature composition

Let $(\mathcal{S}, *)$ represent the algebra formed by the set of type signatures together with the operation of type signature composition.

We introduce the following rule for ϕ and any $u : \mathcal{S}$.

$$u\phi = \phi u = \phi$$

Thus ϕ is the zero element for our algebra.

Note also that $(\overset{\text{ret}}{\rightarrow})$ is an identity element since $(\overset{\text{ret}}{\rightarrow})u = u$ by rule 1 and $u(\overset{\text{ret}}{\rightarrow}) = u$ by rule 2.

In the discussion that follows we assume the following identifiers and predicates.

$u, v : \mathcal{S}$

$s_1, s_2, t_1, t_2 : \mathrm{fseq}\, \mathcal{T}$

$u = s_1 \overset{\mathrm{ret}}{\longrightarrow} s_2$

$v = t_1 \overset{\mathrm{ret}}{\longrightarrow} t_2$

Suppose $u * v \neq \phi$. Then there must be some $r : \mathrm{fseq}\, \mathcal{T}$ such that by zero or more applications of rule 4 we obtain

$u * v = (s_1 \overset{\mathrm{ret}}{\longrightarrow})(r \overset{\mathrm{ret}}{\longrightarrow} t_2)$ where $t_1 = r \frown s_2$

or

$u * v = (s_1 \overset{\mathrm{ret}}{\longrightarrow} r)(\overset{\mathrm{ret}}{\longrightarrow} t_2)$ where $s_2 = r \frown t_1$

We can therefore express the rules for type signature composition of u and v as follows:

$$
\begin{array}{l}
(\exists\, r : \mathrm{fseq}\, \mathcal{T} \bullet \\
\quad t_1 = r \frown s_2 \wedge u * v = r \frown s_1 \overset{\mathrm{ret}}{\longrightarrow} t_2 \\
\qquad\qquad \vee \\
\quad t_1 = r \frown t_2 \wedge u * v = s_1 \overset{\mathrm{ret}}{\longrightarrow} r \frown t_2) \\
\vee \\
u * v = \phi
\end{array}
$$

Some results become immediately obvious.

Given $r, s, t : \mathrm{fseq}\, \mathcal{T}$

$(r \overset{\mathrm{ret}}{\longrightarrow} s)(s \overset{\mathrm{ret}}{\longrightarrow} t) = r \overset{\mathrm{ret}}{\longrightarrow} t$

$(s \overset{\mathrm{ret}}{\longrightarrow} s)^n = s \overset{\mathrm{ret}}{\longrightarrow} s$

We can also show associativity, i.e. for any $u, v, w : \mathcal{S}$

$(uv)w = u(vw)$

The proof, by considering cases, is obvious but long and is omitted.

6. The Composition of Alternative Type Signatures

We introduce an operation + such that if s_1 and s_2 are alternative type signatures, $s_1 + s_2$ is interpreted as the type signature of an operation that can have type signature s_1 or type signature s_2.

The + operator is commutative and obeys the distributive laws. i.e.

$s_1 + s_2 = s_2 + s_1$

$s_1(s_2 + s_3) = s_1 s_2 + s_1 s_3$

$(s_1 + s_2)s_3 = s_1 s_3 + s_2 s_3$

The zero element from type signature composition functions an an identity element for $+$

$s + \phi = s$

Also $s + s = s$.

Using the algebra $(\mathcal{S}, \{+, *\})$ we can give results for programs involving conditional statements and iteration, as well as for primitive "type free" operations.

A Forth `IF` structure has the form

`IF` α `ELSE` β `THEN`

The condition test precedes the `IF` and returns an argument of type *flag* which is consumed by the `IF`. Relating this to English syntax we have something akin to the form:

Is it raining?
Ifso visit club
else visit park
then eat sandwiches.

We declare

$\omega : \mathrm{fseq}\, \Delta$

to represent a Forth program. Let

$\omega =$ `IF` α `ELSE` β `THEN`

Then

$sig\, \omega = (flag \xrightarrow{\text{ret}})(sig\, \alpha + sig\, \beta)$

One standard Forth loop structure has the form

`BEGIN` α `WHILE` β `REPEAT`

`BEGIN` marks the beginning of the loop. The sequence of Forth words α provides a flag which is consumed at `WHILE`. If the flag is true, the word sequence β is executed and control is passed back to `BEGIN`. Otherwise execution continues with the word that follows `REPEAT`. Let

$\omega =$ `BEGIN` α `WHILE` β `REPEAT`

Then

$sig\, \omega = (\sum_{i=0}^{\infty}(sig\, \alpha(flag \xrightarrow{\text{ret}})sig\, \beta))^i\, sig\, \alpha(flag \xrightarrow{\text{ret}}))$

Since we have no way of knowing at compile time how many times a loop might execute, the type signature is an infinite sum. However, if the loop is "balanced" in terms of stack arguments the signature will simplify to a single term. The loop is balanced if there exists some $s : \mathrm{fseq}\, \mathcal{T}$ such that

$sig\, \alpha(flag \xrightarrow{\text{ret}})sig\, \beta = s \xrightarrow{\text{ret}} s$

The signature of ω then reduces to

$sig\, \omega = (s \xrightarrow{\text{ret}} s)sig\, \alpha(flag \xrightarrow{\text{ret}})$

7. Type Variables

We use the identiers $w, w', w_1, w_1', \ldots$ to represent "type variables". Type variables are used to represent items of unknown type.

Given w a type variable and $u : \mathcal{S}$ a signature which may depend on w and an identifier x which does not occur in u, we introduce the following rule for variable introduction or elimination.

$$u = \sum_{x \in \mathcal{T}} u[x/w]$$

Type variables can be used to describe the the signatures of type free operations. For example the Forth word `SWAP` exchanges the top two items on the stack regardless of their type. We write its signature as:

$$sig\,\texttt{SWAP} = w_1\ w_2 \overset{\text{ret}}{\longrightarrow} w_2\ w_1$$

This is equivalent to:

$$sig\,\texttt{SWAP} = \sum_{x \in \mathcal{T}} \sum_{y \in \mathcal{T}} (x\ y \overset{\text{ret}}{\longrightarrow} y\ x)$$

We need rules for the reduction of expressions containing type variables. These should allow us to deduce, for example, that:

$$(w_1\ w_2 \overset{\text{ret}}{\longrightarrow} w_2\ w_1)(w_1\ w_2 \overset{\text{ret}}{\longrightarrow} w_2\ w_1) = (w_1\ w_2 \overset{\text{ret}}{\longrightarrow} w_1\ w_2)$$

Type composition rules for signatures containing variable types are derived from the rules for fixed types and the rule for variable introduction and elimination, plus a rule to avoid variable name clashes.

Variable names clashes can occur where type signature reduction causes two variable scopes to be merged. For example consider:

$$(w_1\ w_2 \overset{\text{ret}}{\longrightarrow} w_2\ w_1)(w_1\ w_2 \overset{\text{ret}}{\longrightarrow} w_2\ w_1)$$

here the variables w_1 in the first signature will in general not represent the same type as w_1 in the second signature (in fact it will represent the same type as w_2 represents in the second signature). Thus before performing any type reduction steps we must rename variables as necessary to ensure that no names clash occurs.

We can now introduce the necesary theorems for reducing expressions that contain type variables.

Theorem 1. Given $s, u, v : \text{fseq}\,\mathcal{T}$ where s, u and v may be dependant on type variables:

$$(s \overset{\text{ret}}{\longrightarrow})(u \overset{\text{ret}}{\longrightarrow} v) = u \frown s \overset{\text{ret}}{\longrightarrow} v$$

Proof. Let $w_1 \ldots w_r$ be the only type variables that occur in s, u or v, and let $x_1 \ldots x_r$ be identifier names that do not occur in s, u or v. Then

$$(s \overset{\text{ret}}{\longrightarrow})(u \overset{\text{ret}}{\longrightarrow} v) = \sum_{x_1 \ldots x_n} ((s \overset{\text{ret}}{\longrightarrow})(u \overset{\text{ret}}{\longrightarrow} v))[x_1/w_1 \ldots x_r/w_r]$$

$$= \sum_{x_1 \ldots x_n} (u \frown s \overset{\text{ret}}{\longrightarrow} v)[x_1/w_1 \ldots x_r/w_r]$$

$$= u \frown s \overset{\text{ret}}{\longrightarrow} v$$

□

Theorem 2. Given $s, t, v : \mathrm{fseq}\,\mathcal{T}$ where s, t and v may be dependant on type variables:

$$(s \overset{\text{ret}}{\rightarrow} t)(\overset{\text{ret}}{\rightarrow} v) = s \overset{\text{ret}}{\rightarrow} t \frown v$$

Proof. Similar to theorem 1 □

We now give the theorems needed to match and cancel individual items.

Theorem 3. Given $s, t, u, v : \mathrm{fseq}\,\mathcal{T}$ and w_1, w_2 type variables and $m, n : \mathbb{N} \mid n < m$

$$(s \overset{\text{ret}}{\rightarrow} t \frown \rho^m w_1)(u \frown \rho^n w_2 \overset{\text{ret}}{\rightarrow} v) = (s \overset{\text{ret}}{\rightarrow} t)((u \overset{\text{ret}}{\rightarrow} v)[\rho^{m-n} w_1 / w_2])$$

Proof.

$$LHS = \textstyle\sum_{x \in \mathcal{T}} \sum_{y \in \mathcal{T}} (s \overset{\text{ret}}{\rightarrow} t \frown \rho^m w_1)[x/w_1](u \frown \rho^n w_2 \overset{\text{ret}}{\rightarrow} v)[y/w_2]$$

Non zero terms occur only where $\rho^m x = \rho^n y$ i.e. where $y = \rho^{m-n} x$. Hence LHS

$$= \textstyle\sum_{x \in \mathcal{T}} (s[x/w_1] \overset{\text{ret}}{\rightarrow} t[x/w_1] \frown \rho^m x)(u[\rho^{m-n} x / w_2] \frown \rho^{m-n} x \rho^n x \overset{\text{ret}}{\rightarrow} v[\rho^{m-n} x / w_2])$$

$$= \textstyle\sum_{x \in \mathcal{T}} (s[x/w_1] \overset{\text{ret}}{\rightarrow} t[x/w_1])(u[\rho^{m-n} x / w_2] \overset{\text{ret}}{\rightarrow} v[\rho^{m-n} x / w_2])$$

$$= (s \overset{\text{ret}}{\rightarrow} t)(u[\rho^{m-n} w_1 / w_2] \overset{\text{ret}}{\rightarrow} v[\rho^{m-n} w_1 / w_2])$$

$$= (s \overset{\text{ret}}{\rightarrow} t)((u \overset{\text{ret}}{\rightarrow} v)[\rho^{m-n} w_1 / w_2])$$

□

The following may be similarly proved.

Theorem 4. Given $s, t, u, v : \mathrm{fseq}\,\mathcal{T}$ and w_1, w_2 type variables and $m, n : \mathbb{N} \mid n > m$

$$(s \overset{\text{ret}}{\rightarrow} t \frown \rho^m w_1)(u \frown \rho^n w_2 \overset{\text{ret}}{\rightarrow} v) = ((s \overset{\text{ret}}{\rightarrow} t)[\rho^{n-m} w_2 / w_1])(u \overset{\text{ret}}{\rightarrow} v)$$

Theorem 5. Given $s, t, u, v : \mathrm{fseq}\,\mathcal{T}$ and $a : \mathcal{T}$
w a type variable and $m, n : \mathbb{N}$

$$(s \overset{\text{ret}}{\rightarrow} t \frown \rho^m a)(u \frown \rho^n w \overset{\text{ret}}{\rightarrow} v) =$$
$$\text{if } m < n \text{ then } \phi$$
$$\text{else } (s \overset{\text{ret}}{\rightarrow} t)((u \overset{\text{ret}}{\rightarrow} v)[\rho^{m-n} a / w])$$

Theorem 6. Given $s, t, u, v : \mathrm{fseq}\,\mathcal{T}$ and $a : \mathcal{T}$
w a type variable and $m, n : \mathbb{N}$

$$(s \overset{\text{ret}}{\rightarrow} t \frown \rho^m w)(u \frown \rho^n a \overset{\text{ret}}{\rightarrow} v) =$$
$$\text{if } m > n \text{ then } \phi$$
$$\text{else } ((s \overset{\text{ret}}{\rightarrow} t)[\rho^{n-m} a / w))(u \overset{\text{ret}}{\rightarrow} v)$$

We can illustrate an application of these theorems by reducing the type signature of `SWAP SWAP`

$$(w_1\ w_2 \overset{\text{ret}}{\rightarrow} w_2\ w_1)(w_1\ w_2 \overset{\text{ret}}{\rightarrow} w_2\ w_1)$$

$= (w_1\ w_2 \overset{\text{ret}}{\rightarrow} w_2\ w_1)(w_1'\ w_2' \overset{\text{ret}}{\rightarrow} w_2'\ w_1')$ [renaming]

$= (w_1\ w_2 \overset{\text{ret}}{\rightarrow} w_2)(w_1' \overset{\text{ret}}{\rightarrow} w_1\ w_1')$ [by theorem 3]

$= (w_1\ w_2 \overset{\text{ret}}{\rightarrow})(\overset{\text{ret}}{\rightarrow} w_1\ w_2)$ [by theorem 3]

$= (w_1\ w_2 \overset{\text{ret}}{\rightarrow} w_1\ w_2)$ [by theorem 1]

A typical use of theorems 5 and 6 is in deriving the type signature of programs containing type free memory access operations. Forth can access its memory space using the type free words @ and !. The signatures of these words are

$sig\,@ = (\rho w \overset{\text{ret}}{\rightarrow} w)$

$sig\,! = (w\ \rho w \overset{\text{ret}}{\rightarrow})$

The type variables in these signatures are typically instantiated by composing the signatures with others that provide additional type information. For example:

$(a\ b \overset{\text{ret}}{\rightarrow} \rho c)sig\,@ = a\ b \overset{\text{ret}}{\rightarrow} c$ [by theorem 5 and rule 1 or 2]

$sig\,@(a\ b \overset{\text{ret}}{\rightarrow} c) = a\ \rho b \overset{\text{ret}}{\rightarrow} c$ [by theorem 6 and rule 1]

$(\overset{\text{ret}}{\rightarrow} a\ \rho a)sig\,! = (\overset{\text{ret}}{\rightarrow})$ [by theorem 5, rule 4, and rule 1 or 2]

8. Subtypes

So far we have taken a simple view in which types match exactly. An alternative is to consider atomic types as a partially ordered set, with the ordering indicating a sub-type relation.

For example suppose a_1 and a_2 are atomic types with

$a_1 \leq a_2$

This would indicate that a_1 is a subtype of a_2. Values satisfying a_1 would also satisfy a_2.

Pointers to atomic types derive an ordering from the ordering of the atomic types. Thus for any n and any atomic types a_1, a_2

$a_1 \leq a_2 \Rightarrow \rho^n a_1 \leq \rho^n a_2$

To deal with subtypes we need to reformulate rules 3 and 4 for type signature composition as follows.

3a. $\neg (x \leq y) \Rightarrow (s_1 \overset{\text{ret}}{\rightarrow} s_2 \frown x)(t_1 \frown y \overset{\text{ret}}{\rightarrow} t_2) = \phi$

4a. $x \leq y \Rightarrow (s_1 \overset{\text{ret}}{\rightarrow} s_2 \frown x)(t_1 \frown y \overset{\text{ret}}{\rightarrow} t_2) = (s_1 \overset{\text{ret}}{\rightarrow} s_2)(t_1 \overset{\text{ret}}{\rightarrow} t_2)$

9. Type Correct Programs

One possible way to define the type correctness of a program ω with respect to a specified type signature u is to say that ω is type correct with respect to u if $sig\,\omega = u$.

However, this does not take into account the possibility of using u to help interpret the type of ω.

For example consider the Forth word **AND** which removes two items from the stack, performs a bitwise logical **AND**, and pushes the result back onto the stack. Forth represents the values *true* and *false* with binary values 0 and -1 (noting that in two's complement form a -1 is represented by a binary number with all bits set to zero). So the machine primitive corresponding to **AND** will perform boolean operations on flags as well as operations on bitwise logical entities. We write its type as

$$sig\mathbf{AND} = (\mathit{flag}\ \mathit{flag} \xrightarrow{\text{ret}} \mathit{flag}) + (\mathit{logical}\ \mathit{logical} \xrightarrow{\text{ret}} \mathit{logical})$$

We would like a definition of type correctness which allows **AND** to be type correct with respect, let us say, to the specification $\mathit{flag}\ \mathit{flag} \xrightarrow{\text{ret}} \mathit{flag}$.

We define functions to extract the inputs and outputs of a type signature.

$$\mathit{inputs}(s \xrightarrow{\text{ret}} t) = s$$

$$\mathit{outputs}(s \xrightarrow{\text{ret}} t) = t$$

And we say ω is type correct with respect to a type specification u iff

$$(\xrightarrow{\text{ret}} \mathit{inputs}\ u)\ \mathit{sig}\,\omega(\mathit{outputs}\ u \xrightarrow{\text{ret}}) = (\xrightarrow{\text{ret}})$$

With this definition **AND** is type correct with respect to $\mathit{flag}\ \mathit{flag} \xrightarrow{\text{ret}} \mathit{flag}$.

The definition also helps us to handle subtypes. For example suppose we have

$$a_1, a_2, b_1, b_2 : \mathcal{T}$$

$$a_1 \leq a_2$$

$$b_1 \leq b_2$$

$$u = (a_2 \xrightarrow{\text{ret}} b_2)$$

$$\mathit{sig}\,\omega = (a_1 \xrightarrow{\text{ret}} b_2) + (a_2 \xrightarrow{\text{ret}} b_1)$$

It follows obviously that ω is type correct with respect to u.

References

[ANS91] *ANS ACS X3/X3J14 Programming Languages: Forth, Draft Standard*, 1^{st} Revision, American National Standards Institute, 1991.

[Bro87] Brodie, Leo: *Starting Forth*, 2^{nd} Edition, Prentice Hall International, 1987.

[NP70] Nivat, M. and Perrot, J. F.: Une generalisation du monoid bicyclique, *C.R. Acad Sci.*, Paris, 271A, 1970, pp. 824–827.

[Pöi90a] Pöial, Jaanus: The Algebraic Specification of Stack Effects for Forth Programs, *Proc. EuroForml90 Conf.*, Southampton, UK, October, 1990.

[Pöi90b] Pöial, Jaanus: Letter to the authors, 1990.

[Pöi91] Pöial, Jaanus: Multiple Stack Effects of Forth Programs, *Proc. EuroForml91 Conf.*, Marianbad, Czechoslovakia, October, 1991.

[Wic88] Wickes, W. C.: RPL: A Mathematical Control Language, *Proc. 1988 Rochester Forth Conf.*, Rochester, NY, USA, 1988.