TYPE INFERENCE IN STACK BASED LANGUAGES

Bill Stoddart & Peter Knaggs
School of Computing and Mathematics
Teesside Polytechnic
Middlesbrough TS13BA
UK

Tel: 0642 (International 44642) 218121

Fax: 0642 (International 44642) 226822

Email: NER034 @ TP.AC.UK

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 derived from the type signatures of its individual operations. The algebra is powerful enough to deal with typeless "wildcard" operations, such as swapping the top two elements on the stack, iterative control structures, and "type free" machine level primitives.

Although these theories apply in principle to any stack based language, they have been evolved with particular regard to the language Forth, 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 programming.

Key words

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

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], etc).

We consider 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 given by a type signature.

For example, we write (a b -- 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. In this notation, the top of the stack is shown to the right.

Suppose an operation with type signature (a b -- c) is followed by an operation with type signature (d c -- a a). We write

(a b -- c)(d c -- 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 type of argument required from the top of the stack by the second operation. The types match, and we can cancel them as follows:

(a b -- c)(d c -- a a) = (a b --)(d -- 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 --)(d -- a a) = (d a b -- a a)

We can check these manipulations by means of a stack "trace", which starts with arguments d, a and b on the stack and terminates with arguments a a.

Operation Signature	Stack	
	d a b	
(a b -- c)	d c	(pop types a, b, push type c)
(d c -- a a)	a a	

We also consider primitive type free operations, such as swapping the top two elements of the stack, irrespective of type. The operation has the signature (w_1 w_2 -- w_2 w_1). We call w_1 and w_2 wildcards because they may be of any type. Obviously we will want rules for wildcards which give reductions such as:

(w_1 w_2 -- w_2 w_1)(w_1 w_2 -- w_2 w_1) = (w_1 w_2 -- w_1 w_2)

The initial formulation of the type signature algebra was presented by Jaanus Pöial at Euro Forml 90, the annual European conference of Forth users. [POI90-1]. His algebraic theory draws on work by Nivat & Perrot [NIV70]. The current paper adds rules for wildcards and other multi-types, and gives the type composition rules for some control structures (the latter having also been formulated, though in a different way, by J Pöial [POI90-2]).

NOTATIONS

We assume elementary set theory and first order predicate logic. Other mathematical structures (relations, functions and sequences) are modelled in terms of set theory. Because we hope that this paper will be read by those involved with Forth as well as by those with an interest in formal aspects, a fairly complete review of the notations used is given.

Sets

For an arbitrary set X

#X represents the number of elements in X.

An identifier x representing an element of X is introduced with the declaration:

x:X

Subsequent to its declaration, we say that x is an element of a set A by writing:

$x \in A$.

Set Description

We can describe the elements of a set by direct enumeration, e.g.
$A = a_1 \mid a_2 \mid a_3$
This tells us that $A = \{ a_1, a_2, a_3 \}$ and that a_1, a_2, a_3 are three distinct elements.

Otherwise we use the form:
A = { *declaration* ¦ *predicate* . *shape* }
Where *declaration* declares bound variables used in the set description, *shape* gives the form of the set elements, and *predicate* is a predicate defined over the bound variables which is satisfied iff elements with the given shape are members of the set.

For example the set of all even numbers greater that 10 could be described with:
{ n : N ¦ n > 5 . 2n }

where N represents the set of natural numbers.

An alternative form is
A = { x:X ¦ p(x) }
Which defines the subset of X which contains exactly those elements of X for which p(x) is true.

Power Sets and Cartesian Products

The power set of A (the set of all subsets of A) is denoted by 2^A.

The cartesian product set $A \times B$ of A and B is the set of all ordered pairs (a:A,b:B).

We introduce the following identifiers and predicates:

$p, q : \Delta$
$s_1, s_2, t_1, t_2 : \text{seq } T$
$sig(p) = (s_1, s_2)$
$sig(q) = (t_1, t_2)$

p and q represent two "words" in the language.
s_1 is the sequence of input argument types for p, and s_2 it the sequence of output argument types for p.

We use pq to represent the sequential composition of the operations p and q.

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(pq) = sig(p)sig(q)

For any type sequences s_1, s_2 we introduce the syntactic equivalence:
$(s_1, s_2) == (s_1 \text{ -- } s_2)$

In addition, when writing out the elements of sequences in the context of a type signature, we omit sequence brackets and us spaces as separators. Thus
(< a, b, c > -- < a, a >)
will be written as
(a b c -- a a)
This gives us the "stack notation" used by Forth programmers.

Type Signature Reduction

We present a set of rules for reducing $(s_1\text{--}s_2)(t_1\text{--}t_2)$ to a single type signature.

1. If $\#s_2 = 0$ (i.e. if s_2 is of length 0)
   $(s_1 \text{ -- } s_2)(t_1 \text{ -- } t_2) = (t_1 s_1 \text{ -- } s_2)$

Example:
sig(p)sig(q) = (a b --)(c -- d) = (c a b -- 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. If $\#t_1 = 0$
   $(s_1 \text{ -- } s_2)(t_1 \text{ -- } t_2) = (s_1 \text{ -- } s_2 t_2)$

Example:
sig(p)sig(q) = (a -- b)(-- c) = (a -- b c)
p takes an argument of type a and leaves an argument of type b.

q takes no arguments and leaves one argument of type c.
pq takes an argument of type a and leaves arguments of type b and c.

These first two rules cover the cases in which no parameters are passed between the two operations. No type clash can occur in such circumstances.

The third rule detects a type clash

3. If $\#s_2>0 \wedge \#t_1>0 \wedge last(s_2) \in K \wedge last(t_1) \in K$
$\wedge$
$last(s_2) \neq last(t_1)$
then
$(s_1 -- s_2)(t_1 -- t_2) = 0$

Example
sig(p)sig(q) = (a -- a b)(b c -- d) = 0
Here p leaves an argument of type b on the top of the stack, and q requires an argument of type c to be on the top of the stack.

We call the above rules composition rules. Repeated application of the remaining rules (called reduction rules) reduces the type signatures until one of the composition rules applies.

Rule 4 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 top items.

4. If $\#s_2>0 \wedge \#t_1>0 \wedge last(s_2) \in K \wedge last(t_1) \in K$
$\wedge$
$last(s_2)=last(t_1)$
then
$(s_1 -- s_2)(t_1 -- t_2) = (s_1 -- front(s_2))(front(t_1)--t_2)$

Example:
sig(p)sig(q) = (a -- b)(a b -- c) = (a --)(a -- c)
Here p leaves an argument of type b on top of the stack as required by q.

The remaining rules deal with wildcards.

Rules 5 and 6 cover case in which a wildcard is matched against an argument of known type. In these cases the wildcard is replaced by the known type.

5. If $\#s_2>0 \wedge \#t_1>0 \wedge last(s_2) \in K \wedge last(t_1) \in W$
then
$(s_1 -- s_2)(t_1 -- t_2) =$
$(s_1 -- front(s_2))((front(t_1)--t_2)[last(s_2)/last(t_1)])$

Example:
(a -- b c)(w_1 w_2 -- w_1 w_2 w_1)
= (a -- b)((w_1 -- w_1 w_2 w_1)[c/w_2])
= (a -- b)(w_1 -- w_1 c w_1)

6. If $\#s_2>0 \wedge \#t_1>0 \wedge last(s_2) \in W \wedge last(t_1) \in K$
then
$(s_1 -- s_2)(t_1 -- t_2) =$
$((s_1 -- front(s_2))[last(t_1)/last(s_2)])(front(t_1) -- t_2)$

Example:

(w_1 w_2 -- w_2 w_1)(a b -- c)
= ((w_1 w_2 -- w_2)[b/w_1])(a -- c)
= (b w_2 -- w_2)(a -- c)

Rule 7 covers cases in which a wildcard is matched against a wildcard. In this situation we may also have a names clash, in which case we rename the identifiers in the second signature. The renaming is only done once during a reduction.

For example consider:

(s_1 -- s_2)(t_1 -- t_2) =
(w_1 w_2 -- w_1 w_2 w_1 w_2)(w_1 w_2 w_3 -- w_2 w_3 w_1)

Here we rename the w_1 and w_2 in t_1 and t_2 to w_1' and w_2', giving:

(w_1 w_2 -- w_1 w_2 w_1 w_2)(w_1' w_2' w_3 -- w_2' w_3 w_1')

Now rule 7 allows us to cancel last(s_2) and last(t_1) when these are both wildcards. We must also substitute the name last(s_2) for the name last(t_1) wherever the latter occurs in t_1 or t_2. Applying rule 7 three times and then applying rule 2 gives the following reduction.

(w_1 w_2 -- w_1 w_2 w_1 w_2)(w_1' w_2' w_3 -- w_2' w_3 w_1')
= (w_1 w_2 -- w_1 w_2 w_1)(w_1' w_2' -- w_2' w_2 w_1')
= (w_1 w_2 -- w_1 w_2)(w_1' -- w_1 w_2 w_1')
= (w_1 w_2 -- w_1)(-- w_1 w_2 w_2)
= (w_1 w_2 -- w_1 w_1 w_2 w_2)

Rule 7 is given formally as follows:

7. If $\#s_2 > 0 \wedge \#t_1 > 0 \wedge last(s_2) \in W \wedge last(t_1) \in W$
then
$(s_1 -- s_2)(t_1 -- t_2) =$
$(s_1 -- front(s_2))((front(t_1) -- t_2)[last(s_2)/last(t_1)])$

This completes the rules for type signature reduction.

The complete reduction of a type signature $(s_1 -- s_2)(t_1 -- t_2)$ giving a non zero result requires n steps where:

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

To see this note that all reduction rules reduce the length of s_2 and t_1 by one, and when either of these sequences becomes empty a single composition rule can be used to complete the reduction.

We now give some examples in which complete reductions are performed. The rule being used at each step is noted on the right.

1. (a -- b c d)(w_1 w_2 -- w_2 w_1)
= (a -- b c)(w_1 -- d w_1) Rule 5.
= (a -- b)(-- d c) Rule 5.
= (a -- b d c) Rule 2.

2.	$(w_1\ w_2\ w_3 \text{ -- } w_2\ w_3\ w_1)(a\ b \text{ -- } c)$	
=	$(b\ w_2\ w_3 \text{ -- } w_2\ w_3)(a \text{ -- } c)$	Rule 6.
=	$(b\ w_2\ a \text{ -- } w_2)(\text{ -- } c)$	Rule 6.
=	$(b\ w_2\ a \text{ -- } w_2\ c)$	Rule 2.
=	$(b\ w\ a \text{ -- } w\ c)$	Since the naming of wildcards is arbitrary.

3.	$(w_1\ w_2 \text{ -- } w_1\ w_2\ w_1)(w_1\ w_2 \text{ -- } w_1\ w_2\ w_1)$	
=	$(w_1\ w_2 \text{ -- } w_1\ w_2\ w_1)(w_1'\ w_2' \text{ -- } w_1'\ w_2'\ w_1')$	Renaming
=	$(w_1\ w_2 \text{ -- } w_1\ w_2)(w_1' \text{ -- } w_1'\ w_1\ w_1')$	Rule 7.
=	$(w_1\ w_2 \text{ -- } w_1)(\text{ -- } w_2\ w_1\ w_2)$	Rule 7.
=	$(w_1\ w_2 \text{ -- } w_1\ w_2\ w_1\ w_2)$	Rule 2.

The Algebra of Type Signature Composition

Let $(\Phi, \{ * \})$ represent the algebra formed by the set of type signatures together with the operation of type signature composition. We now investigate the structure of this algebra. We need the following rule for 0 and for any $u : \Phi$.

$u0 = 0u = 0$

Note also that:

$1 == (\text{ -- })$

is an identity element since for an arbitrary signature $u : \Phi$ since:

$u(\text{ -- }) = u$ by rule 2

and

$(\text{ -- })u = u$ by rule 1

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

$u, v, : \Phi$

$s_1, s_2, t_1, t_2, a, b : \text{seq } T$

$u = (s_1 \text{ -- } s_2)$
$v = (t_1 \text{ -- } t_2)$

Consider uv where s_1, s_2, t_1, t_2 contain no wildcards. In this case, either uv=0, or by repeated application of reduction rule 4 we obtain:

$uv = (s_1 \text{ -- })(a \text{ -- } t_2)$ where $t_1 = a \hat{} s_2$

or

$uv = (s_1 \text{ -- } b)(\text{ -- } t_2)$ where $s_2 = b \hat{} t_1$

So in this case we can express the rules for type signature composition as follows:

$$\exists a : seq\ K \cdot t1 = a^\frown s2 \wedge uv = (a^\frown s1\ --\ t2)$$

$$\vee$$

$$\exists b : seq\ K \cdot t1 = b^\frown t2 \wedge uv = (s1\ --\ b^\frown t2)$$

$$\vee$$

$$uv = 0$$

Some results become immediately obvious.

Given r, s, t : K

$$(r\ --\ s)(s\ --\ t) = (r\ --\ t)$$

$$(s\ --\ s)^n = (s\ --\ s)$$

We can also show associativity, i.e. for any u, v, w : Φ.

$$(uv)w = u(vw)$$

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

We are not at present able to prove that the properties of the algebra are unaffected by the introduction of wildcards, but we have a strong intuitive belief that the above results will remain true.

The Composition of Alternative Type Signatures

We introduce an operator, called + say, such that if s_1 and s_2 are type signatures, s_1+s_2 would be interpreted as the type signature of an operationthat can have type signature s_1 or type signature s_2.

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

$s_1+s_2 = s_2+s_1$

$s_1(s_2+s_3) = s_1s_2+s_1s_3$

$(s_1+s_2)s_3 = s_1s_3+s_2s_3$

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

$s+0 = s$

Also $s+s = s$

Using the algebra $(\Phi,\{+,*\})$ we can give results for programs involving conditional statements and iteration, as well as for primitive "type free" operations. Our examples are taken from the Forth language.

We will distinguish Forth words that are enclosed by normal text by printing them in bold.

A Forth **IF** structure has the form:

IF α ELSE β THEN

Where α and β are sequences of Forth words.

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 go to club
else go to park
then eat sandwiches.

We declare:

ω : seq Δ

to represent a Forth program.

Let

ω = IF α ELSE β THEN

Then

sig(ω) = (flag --)(sig(α) + sig(β))

One standard Forth iteration 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 executes and control is passed back to **BEGIN.** Otherwise execution continues with the word that follows **REPEAT.**

Let

ω = BEGIN α WHILE β REPEAT

Then:

$$\text{sig}(\omega) = \left(\sum_{i=0}^{\infty} (\text{sig}(\alpha)(\text{flag -- }) \text{ sig}(\beta))^i \text{ sig}(\alpha)(\text{flag -- }) \right)$$

Since we have no semantic information on the language in this theory we have no way of knowing how many times the loop might execute. The type signature is therefore an infinite sum. However, if the loop is "balanced" in terms of stack arguments this complex signature will simplify to a single term.

The loop is balanced if there exists some s : seq T such that

sig(α)(flag --)sig(β) = (s -- s)

The signature of ω then reduces to:

sig(ω) = (s -- s)sig(α)(flag --)

Multiple Type Signatures and Primitive Operations

Forth has a nucleus of primitive operations that often have multiple type signatures because they operate on primitive binary data which may have more than one interpretation. For example **AND** removes two binary numbers from the stack, performs a bitwise binary *and* operation, and pushes the result back to the stack.

We can interpret this as having type signature (logical logical -- logical), where "logical" is a type representing a sequence of binary bits equal in length to the wordsize of the machine (typically 16 or 32 bits).

Forth represents the Boolean values *true* and *false* with binary values 0 and -1. (noting that in twos complement arithmetic a -1 is represented by a binary number with all bits set to 1). So the same machine primitive function **AND** will also perform boolen operations on flags. We write its type as:
sig(**AND**) = (flag flag -- flag) + (logical logical -- logical)

Memory Access Words

We divide K, the set of known types, into a set of basic types B and a set of pointer types P.
B, P : 2^K
$B \cup P = K$
$P = \{ t:B, n:\mathbb{N} \mid n>0 \cdot *^n t \}$

Here * indicates indirection, so an identifier of type *a would be a pointer to an item of type a (where *a == $*^1$a)

We introduce the Forth word **@**, whose function is to remove an address from the stack and return the contents of that address.

@ : Δ

$sig(@) = \sum_{k \in K} (*k -- k)$

This infinite sum will often collapse to a single term on being composed with a signature (s -- t) which provides additional type information. Here are two examples:

(a b -- *c)sig(@) = (a b -- c)

and

sig(@)(a b -- c) = (a *b -- c)

The most general rules for such reductions can be stated as:

For all
s, t : seq T
x : K

(s -- t^<*x>)sig(@) = (s -- t^<x>)

and sig(@)(s^<x> -- t) = (s^<*x> -- t)

These rules may be easily proved. For example consider the first:

We have
(s -- t^<*x>)sig(@)
= (s -- t^<*x>)$\sum_k \in K$(*k -- k)
= (s -- t^<*x>)(*x -- x)
= (s -- t)(-- x)
= (s -- t^<x>)

The corresponding word to write to a memory location is "store", denoted by the single character !

"Store" removes an item and an address (the address being the top stack item) and stores the item at the address.

$$sig(!) = \sum_{k \in K} (k\ *k\ --\ )$$

Once again this infinite sum will tend to reduce to a single term if the operator is used correctly. The rule for ! composition is:

(s -- t^<x,*x>)sig(!) = (s -- t)

Structured Data Types

Our set of types is an unstructured set of type names. This does not mean the language itself need be without structured data types, but their structure will not form any part of the type theory. In Forth the relationship between a record and one of its constituent fields is generally an operational one. For example an operation could be defined convert a record address to the address of one of the fields in the record. The type signatures of such conversion operations effectively deal with structured types.

Immediate Words

Some Forth words are executed rather than compiled when they are "seen" during the compilation of a word sequence. For example consider the Forth definition:

```
: ABS ( n -- u ) DUP 0 <
    IF  NEGATE  THEN ;
```

This defines a new word **ABS** which returns the absolute value of an integer. It takes an argument of type n (signed integer) and returns an unsigned integer (type u).

The default compilation process for Forth is to compile a sequence of operations for subsequent execution. Thus the compiled "body" of **ABS** will commence with the operations:

DUP	duplicate the top stack item
0	push a zero onto the stack
<	remove top two stack items, compare, return a flag

When compilation reaches **IF** however, we cannot simply compile an **IF** operation, since we do not yet know the relevant branch offset. **IF** is a typical example of a Forth immediate word. It will compile a branch primitive, leave an offset address to be resolved later (by **THEN**) and will leave a token for a subsequent syntax check. **THEN** will check the syntax and resolve the branch offset. Although beyond the scope of the current paper, it is interesting to attempt a formalism that covers both the compile time (immediate) and run time operations of a sequence of words.

Operations as arguments

Forth allows an operation to be passed as an argument and subsequently executed. The Forth word **EXECUTE** removes an "execution token" from the stack, and executes it. The type of **EXECUTE** is therefore:

$$\sum_{s \in seqT} \sum_{t \in seqT} (s \text{ -- } t)$$

This is too complex a form to allow any meaningfull checks, so we will allow the programmer to provide additional type information whenever **EXECUTE** is used. We could also allow all type signatures as types, but at the moment this is not something we had time to investigate.

Type Specifications and Type Correct Programs

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

However, this does not take into account the possibility of using the signature s to help interpret the type of ω. For example we would like **AND** to be type correct with respect to (flag flag -- flag), which it is not according to the above definition, since:

sig(**AND**) = (flag flag -- flag) + (logical logical -- logical)

We modify the definition of type correctness to take this into account.

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

inputs, outputs : seq T x seq T $\rightarrow$ seq T

inputs(s -- t) = s
outputs(s -- t) = t

We say ω is <u>type correct</u> with respect to a type specification u iff

(-- inputs(u))sig(ω)(outputs(u) --) = 1

With this definition, **AND** is type correct with respect to (flag flag -- flag).

Conclusions

Although the Forth language is always implemented in a type free manner, Forth programmers must keep track of the types they are using and select appropriate operations for those types if their program is to perform correctly. Automatic type checkers for the Forth programming environment have not yet developed, and presumably the reasons for this are that Forth has many "type free" operations, and also that type checking was perhaps thought to be dependant on operation semantics. This paper shows that rules for type free opeations can be included in a type algebra, that over a wide range of possible programs the logic required to keep track of types is independant of operation semantics, and that it is possible to use simple rules to derive the type signature of a program from the types of its constituent operations. We conclude from these results that a Forth type checker is a practical possibility.

A number of questions remain to be answered. We have introduced two methods for dealing with "type free" operations. We used special wildcard rules to deal with stack manipulations such as **SWAP**, and introduced the + operator to our algebra to deal with alternative type signatues. Ideally we would like to <u>derive</u> the wildcard rules from a more basic algebra of <u>known</u>

types. This would unify the theory and give a simpler and more tractable algebra. We have also omitted any consideration of type signatures as types when passing operations as arguments.

Acknowledgements

Part of this research was carried out with the assistance of an SERC CASE award for which the industrial partners were Computer Solutions Ltd. Byfleet. Surrey.

Special acknowledgements are due to Jaanus Poial of Tartu University, Estonia, for introducing the idea that type inference for stack based languages could be formulated algebraically.

References

[BRO87] Leo Brodie. Starting Forth, 2nd Edition. Prentice Hall International.

[NIV70] M Nivat & J F Perrot. Une generalisation du monoid bicyclique. C.R. Acad Sci. Paris, 271A, 1970, p 824 - 827.

[POI90-1] Jaanus Pöial. The Algebraic Specification of Stack Effects for Forth Programs.
Proc. of EuroFORML'90 Conference.

[POI90-2] Jaanus Pöial. Letter to the authors.

[WIC88] W C Wickes. RPL: A Mathematical Control Language. Proc of 1988 University of Rochester Forth Conference. Rochester. NY. USA.