Forth and Formal Language Theory

Jaanus Pöial
Department of Computer Science
University of Tartu, Estonia
e-mail: jaanus@cs.ut.ee

May 27, 1994

Abstract

Forth is an excellent programming paradigm, but it is also an interesting object of investigation for formal language theory. With its clear interface based on stacks Forth programs are easily generated by some formal system (e.g. syntax directed translation scheme). Usually it is possible to make such a formal system to generate only "useful" programs, but this subset of programs is often very small and does not cover interesting features of the Forth language itself.

Current paper outlines some problems of formal syntax and semantics of Forth. Forth is often advertised as a language with no syntax at all – programs are treated as sequences of words and all other rules are just an agreement between programmers (standardization). This viewpoint characterizes the language much better than another extreme – Forth is yet another language which can be described by a context free grammar. Extensibility of Forth makes such an attempt hard and doubtful. Probably the truth lies in between of these two extremes. In the following we try to restrict the set of all possible sequences of Forth words (set of "programs" in first sense) using interface descriptions for the words to exclude a lot of meaningless sequences. At the same time this model of syntax will be more general than the context free grammar. This work is far from practical solutions (e.g. describing ANSI standard formally), but it gives a motivation for describing interfaces of Forth words precisely and demonstrates some methods of formal language theory.

1 Calculus for stack effects

Theory of stack effects and type inference for the Forth language has been developed during last five years (see [Pöi90], [Pöi91], [Pöi93], [StK92], [StK93], [Sto93]). The main idea of stack effect calculus is to provide a tool for static type checking of Forth programs and context free grammars which generate Forth programs. Types, subtypes, "wildcard" types and rules for calculating resulting stack effects for different constructs were introduced.

Let us define the basic notation and list some results.

We will consider that stack effects (type signatures) form a polycyclic monoid ([NiP70]). This holds for simple signatures (uniquely defined stack effects without "wildcards" and subtyping [Pöi90]). This also holds for compound signatures (multiple stack effects [Pöi91]) with some restrictions which we will discuss later. Set of multiple stack effects in general is not even an inverse semigroup, but there exists a subset which is a polycyclic monoid. For simplicity we can use the set of simple signatures everywhere, because this models inverse semigroups (see also [ClP67] for the basic theory of semigroups).

Let *types* be denoted by a, b, c ... In real programs these can be `flag`, `char`, `float`, `addr`, `xtoken`, etc. – we do not need any concrete interpretation yet. Let $\mathcal{T}$ be the set of types.

We will use α, β, γ ... for *type lists*. These are finite sequences of types where the rightmost element corresponds to the top of the stack. The set of such lists is $\mathcal{T}^*$.

A *type clash* appears when some word finds an input argument on the stack which has the wrong type. We will use the symbol $\emptyset$ for the type clash.

A *stack effect* (type signature) is a pair of input parameter types and output parameter types. We also consider the type clash as a stack effect. The set of stack effects is defined as follows

$$\mathcal{S} = (\mathcal{T}^* \times \mathcal{T}^*) \cup \{\emptyset\}$$

We use s, t, u ... for stack effects as well as $(\alpha - -\beta)$ for the pair $(\alpha, \beta) \in \mathcal{T}^* \times \mathcal{T}^*$. Sometimes we use indices to express inputs and outputs

$$s = (s_1 -- s_2) \text{ where } s_1, s_2 \in \mathcal{T}^*$$

The *composition* (multiplication) of stack effects is defined as follows

$\forall s \in \mathcal{S} : s\ \emptyset = \emptyset\ s = \emptyset$

$\forall s_1, s_2, t_1, t_2, \alpha, \beta \in \mathcal{T}^*$:

$(s_1 -- s_2)(\alpha s_2 -- t_2) = (\alpha s_1 -- t_2)$

$(s_1 -- \beta t_1)(t_1 -- t_2) = (s_1 -- \beta t_2)$

In all other cases the result will be $\emptyset$.

$\mathbf{1} = (—)$ is a unity for this operation.

We have an algebraic structure now which is isomorphic to the polycyclic monoid. It has a unity $\mathbf{1}$, a null element $\emptyset$ and an associative operation of multiplication.

Example 1

Let us calculate some products using the definitions above.

$(a -- bc)(c -- de) = (a -- bde)$

$(ab -- c)(dc -- e) = (dab -- e)$

$(a -- b)(bc -- d) = \emptyset$

$(-- ab)(ab --) = \mathbf{1}$

$(a -- b)(cb - d) = (ca -- d)$

$(ca -- cb)(cb -- d) = (ca -- d)$

Last two equations serve as a contraexample to theorem 1 in [Sto93] (there is no right cancellation in polycyclic monoid in general).

Let us define the *inverse element* for any $s \in \mathcal{S}$ in the following way

$s = \emptyset \Rightarrow s^{-1} = \emptyset$, i.e. $\emptyset^{-1} = \emptyset$

$s = (s_1 -- s_2) \Rightarrow s^{-1} = (s_2 -- s_1)$, i.e. $(s_1 -- s_2)^{-1} = (s_2 -- s_1)$

This definition introduces a unique inverse element for each stack effect and allows to define the *partial order relation* $\leq$ as follows

$\emptyset \leq s$ for any $s \in \mathcal{S}$ and

$(\gamma\alpha -- \gamma\beta) \leq (\alpha -- \beta)$ for each $\alpha, \beta, \gamma \in \mathcal{T}^*$

It is equivalent to the classical definition

$$s \leq t \Leftrightarrow st^{-1} = ss^{-1}$$

It is interesting to notice that $(-- s_1)(t_1 -- t_2)(s_2 --) = \mathbf{1}$ is equivalent to $(s_1 -- s_2) \leq (t_1 -- t_2)$ ([Pöi90]). In [StK93] the first form is used to define type correct programs.

All *idempotents* of $\mathcal{S}$, i.e. elements u for which $u = uu$, form a commutative subsemigroup of $\mathcal{S}$ with unity and null element. Non-zero idempotents have a form of $(\alpha -- \alpha)$, where $\alpha \in \mathcal{T}^*$.

Having these basic definitions we can return to the question about multiple stack effects (compound signatures). Subset $\mathcal{M} \subset \mathbf{2}^{\mathcal{S}}$, where $\mathbf{2}^{\mathcal{S}}$ denotes the powerset (set of all subsets) of $\mathcal{S}$, is an inverse semigroup, iff

1) $\sigma, \tau \in \mathcal{M} \Rightarrow \sigma\tau \in \mathcal{M}$ ($\mathcal{M}$ is a subsemigroup),
2) $\forall \sigma \in \mathcal{M}\ \exists \tau \in \mathcal{M} :\ \sigma\tau\sigma = \sigma$ (all elements are *regular*),
3) $\sigma, \tau \in \mathcal{M},\ \sigma\sigma = \sigma,\ \tau\tau = \tau\ \Rightarrow \sigma\tau = \tau\sigma$ (all idempotents commutate).

It is not only a question of how to define $\mathcal{M}$, but also how to define the multiplication (and addition). Author still does not have a good (non-trivial) example for that problem.

2 Stack effects and Forth programs

Let us have a set of stack operations (Forth words) Δ. We can build "programs" by writing sequences of the words. The set of all "programs" is Δ^*. Each word $p \in \Delta$ has a given stack effect $sig(p) \in \mathcal{S}$. Mapping $sig : \Delta^* \rightarrow \mathcal{S}$ is defined as homomorphism

$sig() = \mathbf{1}$

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

Now it is possible to calculate the stack effect of a given program simply by multiplying stack effects of its parts (notice that we need associativity and homomorphism to do this).

The set Δ and homomorphism sig determine a language of valid programs (programs without type clash)

$$Valid(\Delta, sig) = \{\omega \in \Delta^* : sig(\omega) \neq \emptyset\}$$

In some cases a subset of valid programs without input and output parameters is considered

$$Closed(\Delta, sig) = \{\omega \in \Delta^* : sig(\omega) = \mathbf{1}\}$$

Obviously

$$empty\ program \in Closed \subset Valid \subset \Delta^*$$

There are different ways to treat the correctness of Forth programs with regard to the stack effects.

- Exact matching when the calculated effect has to be equal to the desired one. If the desired effect is $\mathbf{1}$ (no inputs, no outputs) we have a *closed program.*
- Operational matching when the desired effect may be less (in sense of our partial order relation) than the calculated one. We see from calculations that the program does the job, but does not use all (bottom) elements on the stack.
- We do not have any idea about desired effects, we only want to have valid programs.

Example 2

If the calculation gives $sig(p) = (bc -- bd)$ then

p is type correct w.r.t. $(bc -- bd)$ in sense of all definitions.

p is not type correct w.r.t. $(abc -- abd)$ in sense of first definition, but satisfies the second one, because $(abc -- abd) \leq (bc -- bd)$. p simply does not use the element a .

p is not type correct w.r.t. $(c -- \ d)$, but is still valid. The reason here is that we cannot put p into context where for example ac is on the top of the stack. It leads to the type clash between a (in context) and b (actually used by program p). At the same time $(c -- \ d)$ works well in this context and we will not notice the error.

If the programs are generated by a context free grammar then it is possible to guarantee their type correctness by checking the grammar (see [Pöi90] for the details). We need to bind an inequality to each grammar rule and to solve the system of inequalities in $\mathcal{S}$.

Example 3

Let us define Δ and sig as follows

$sig(\texttt{val}) = (-- \ c)$
$sig(\texttt{addr}) = (-- \ a)$
$sig(\texttt{dot}) = (c --)$
$sig(\texttt{plus}) = (c\ c -- \ c)$
$sig(\texttt{mult}) = (c\ c -- \ c)$
$sig(\texttt{store}) = (c\ a --)$
$sig(\texttt{fetch}) = (a -- \ c)$

The grammar on the left produces the system of inequalities on the right.

		$\mathbf{1} \leq \mathrm{Prog}$
$<\mathit{Prog}>$	$\rightarrow\ <\mathit{Stms}>$	$\mathrm{Prog} \leq \mathrm{Stms}$
$<\mathit{Stms}>$	$\rightarrow\ <\mathit{Stm}>$	$\mathrm{Stms} \leq \mathrm{Stm}$
	$\rightarrow\ <\mathit{Stms}>\ <\mathit{Stm}>$	$\mathrm{Stms} \leq \mathrm{Stms\ Stm}$
$<\mathit{Stm}>$	$\rightarrow\ <\mathit{Exp}>$ `dot`	$\mathrm{Stm} \leq \mathrm{Exp}\ (c --)$
	$\rightarrow\ <\mathit{Exp}>$ `addr store`	$\mathrm{Stm} \leq \mathrm{Exp}\ (-- \ a)\ (c\ a --)$
$<\mathit{Exp}>$	$\rightarrow\ <\mathit{Mon}>$	$\mathrm{Exp} \leq \mathrm{Mon}$
	$\rightarrow\ <\mathit{Exp}>\ <\mathit{Mon}>$ `plus`	$\mathrm{Exp} \leq \mathrm{Exp\ Mon}\ (c\ c -- \ c)$
$<\mathit{Mon}>$	$\rightarrow\ <\mathit{Term}>$	$\mathrm{Mon} \leq \mathrm{Term}$
	$\rightarrow\ <\mathit{Mon}>\ <\mathit{Term}>$ `mult`	$\mathrm{Mon} \leq \mathrm{Mon\ Term}\ (c\ c -- \ c)$
$<\mathit{Term}>$	$\rightarrow$ `val`	$\mathrm{Term} \leq (-- \ c)$
	$\rightarrow$ `addr fetch`	$\mathrm{Term} \leq (-- \ a)\ (a -- \ c)$
	$\rightarrow\ <\mathit{Exp}>$	$\mathrm{Term} \leq \mathrm{Exp}$

This grammar is an output grammar of a syntax directed translation scheme used in compiler compiler TARTU ([PST90]). It has a "normal" reduced input grammar which is not presented here.

The system has a solution

$\mathrm{Prog} = (—)$
$\mathrm{Stms} = (—)$
$\mathrm{Stm} = (—)$
$\mathrm{Exp} = (-- \ c)$
$\mathrm{Mon} = (-- \ c)$
$\mathrm{Term} = (-- \ c)$

The grammar produces only closed programs, but not *all* closed programs (e.g. `val val dot dot`).

As we see from previous small example the context free grammar is not a good tool to present Forth syntax. We could "improve" the grammar, but then it becomes complicated, huge and non-readable.

3 Stack effects and syntactic equations

Up to now we have mainly listed the results from existing works (terminology and notation between different works are more coordinated here). We have stated that the Forth syntax cannot be expressed by a context free grammar. In the following we try to show that stack effects serve as a formal syntax definition tool for Forth.

We already defined two non-empty languages $Valid(\Delta, sig)$ and $Closed(\Delta, sig)$. Formally this is enough to define a language, but we need to be convinced in usefulness of such a definition. One the one hand we can calculate the stack effect of a program to decide whether it belongs to the language (no more than semigroup is needed to do this). On the other hand we need a lot of algebraic properties of the polycyclic monoid to transform the "stack effect syntax notation" to some other form. That is the reason why we still consider that stack effects form at least an inverse semigroup. In this work we are not interested in (very important) questions of concrete type systems, rules for control structures e.t.c. Our interest is concentrated on binding the stack effect calculus to methods of syntax description. We will show that stack effect calculus and general rewriting rules (syntactic equations) are equivalent in particular cases.

Example 4

Let us have Δ and sig of example 3 and the language $Closed(\Delta, sig)$. We will show that the following syntactic equations define $Closed(\Delta, sig)$ identically (the grammar above defined only a subset of the language).

(1) *empty program* = `val dot`
(2) `val` = `addr fetch`
(3) `dot` = `addr store`
(4) `plus` = `dot dot val`
(5) `mult` = `plus`
(6) `fetch` = `fetch dot val`

This set of equations is not unique, we could choose different ones. First part is obvious – we can easily check that equations between stack effects hold for a given set Δ . To prove that the languages coincide we do some more – we forget about Δ and examine the equations. If we get the same (actually isomorphic) Δ proceeding from the equations then the goal is reached (Δ determines the equations and equations determine the same Δ).

Let us try to solve the following system in $\mathcal{S}$

(1) $(\text{—}) = (v_1 -- v_2)\ (d_1 -- d_2)$
(2) $(v_1 -- v_2) = (a_1 -- a_2)\ (f_1 -- f_2)$
(3) $(d_1 -- d_2) = (a_1 -- a_2)\ (s_1 -- s_2)$
(4) $(p_1 -- p_2) = (d_1 -- d_2)\ (d_1 -- d_2)\ (v_1 -- v_2)$
(5) $(m_1 -- m_2) = (p_1 -- p_2)$
(6) $(f_1 -- f_2) = (f_1 -- f_2)\ (d_1 -- d_2)\ (v_1 -- v_2)$

First equation gives $v_1 = \Lambda$, $d_2 = \Lambda$ and $d_1 = v_2$, where Λ denotes the empty sequence. Let us define $\alpha = d_1\ (= v_2)$ for the future use.

Second equation (together with facts we already know) allows to deduce that $a_1 = \Lambda$ and $\exists \tau \in \mathcal{T}^*: a_2 = \tau f_1\ \&\ v_2 = \tau f_2$. Let us also define $\beta = f_1$.

From the sixth and first equations we find that $\exists \sigma \in \mathcal{T}^*: f_2 = \sigma d_1 = \sigma\alpha$. We already know that $\alpha = v_2 = \tau f_2 = \tau\sigma\alpha$ and now we can conclude that $\tau = \sigma = \Lambda$, $a_2 = \beta$ and $f_2 = \alpha$.

Now we get s_1 and s_2 from the third equation $(\alpha --) = (-- \beta)\ (s_1 -- s_2)$ as follows: $s_2 = \Lambda$ and $s_1 = \alpha\beta$.

As we already know d_1, d_2, v_1 and v_2 , the fourth equation gives $p_1 = \alpha\alpha$ and $p_2 = \alpha$.

Finally the fifth equation defines $m_1 = \alpha\alpha$ and $m_2 = \alpha$.

Let us sum up the result.
$sig(\texttt{val}) = (\ --\ \alpha)$
$sig(\texttt{addr}) = (\ --\ \beta)$
$sig(\texttt{dot}) = (\alpha\ --\ )$
$sig(\texttt{plus}) = (\alpha\ \alpha\ --\ \alpha)$
$sig(\texttt{mult}) = (\alpha\ \alpha\ --\ \alpha)$
$sig(\texttt{store}) = (\alpha\ \beta\ --\ )$
$sig(\texttt{fetch}) = (\beta\ --\ \alpha)$

We still have two independent variables $\alpha, \beta \in \mathcal{T}^*$ here and we have used all the equations. The result is amazingly familiar – look at example 3.

As we have learned from this example the syntax rules may determine stack effects as well as stack effects determine the syntax.

Literature

1. [ClP67] Clifford A.H., Preston G.B. The algebraic theory of semigroups. Rhode Island, 1967.

2. [NiP70] Nivat M., Perrot J.F. Une généralisation du monoïde bicyclique. C.R.Acad.Sci. Paris, 271A, 1970, 824 – 827.

3. [PST90] Pöial J., Soo V., Tombak M. A Forth Oriented Compiler Compiler and its Applications. 1990 FORML Conference Proceedings, EuroFORML'90 Conference, Oct 12 – 14, 1990, Ampfield, Nr Romsey, Hampshire, UK, Forth Interest Group, Inc., San Jose, USA, 1991, 257 – 261.

4. [Pöi90] Pöial J. Algebraic Specifications of Stack-effects for Forth Programs. 1990 FORML Conference Proceedings, EuroFORML'90 Conference, Oct 12 – 14, 1990, Ampfield, Nr Romsey, Hampshire, UK, Forth Interest Group, Inc., San Jose, USA, 1991, 282 – 290.

5. [Pöi91] Pöial J. Multiple Stack-effects of Forth Programs. 1991 FORML Conference Proceedings, euroFORML'91 Conference, Oct 11 – 13, 1991, Marianske Lazne, Czechoslovakia, Forth Interest Group, Inc., Oakland, USA, 1992, 400 – 406.

6. [Pöi93] Pöial J. Some Ideas on Formal Specification of Forth Programs. 9th euroFORTH conference on the FORTH programming language and FORTH processors, Oct 15 – 18, 1993, Marianske Lazne,Czech Republic, 1993, 4 pp.

7. [StK92] Stoddart B., Knaggs P. The (Almost) Complete Theory of Forth Type Inference. Proc. EuroForth Conference 1992.

8. [StK93] Stoddart B., Knaggs P. Type Inference in Stack Based Languages. Formal Aspects of Computing, BCS, 1993, 5, 289 – 298.

9. [Sto93] Stoddart B. Self Reference and Type Inference in Forth. 9th euroFORTH conference on the FORTH programming language and FORTH processors, Oct 15 – 18, 1993, Marianske Lazne,Czech Republic, 1993, 9 pp.