Alternative Syntactic Methods for Defining Stack Based Languages

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

Abstract

Traditional formal methods of syntax description for programming languages (e.g. context free grammars) are not always convenient for practical applications. In some cases the language has quite weak syntactic structure and it is better to describe the correct programs using semantics of operations instead. There exists a class of widespread languages in practice that use stack machines for interpretation of programs, the so called stack based languages (Java virtual machine language [2], Forth, Postscript, etc.). Semantics of stack operations determines the language of correct programs in sense of parameter passing through the stack (see [4], [5], [6], [7] and [8] for the details). This is one alternative method to define the syntax of a stack based language. The main question is whether there exists a better way of defining the same language.

In this paper we investigate systems of syntactic equations (general rewriting rules) on sequences of stack operations. Stack operations have certain effect on the stack that can be described using input and output parameters. Stack effect calculus allows to define the language of correct sequences of stack operations (programs). It is important to know under which conditions the system of syntactic equations and the set of stack effects define the same language, because both methods seem to work better for stack based languages than context free grammars.

1 Stack effect calculus

The main goal of stack effect calculus is static type checking of stack machine programs. Types, subtypes, "wildcard" types and rules for calculating resulting stack effects for different constructs have been introduced in [4], [5] and [8].

Let us define the basic notation and list some results.

We will consider that stack effects (type signatures) form a polycyclic monoid ([3]). This holds for simple signatures (uniquely defined stack effects without "wildcards" and subtyping [4]). This also holds for compound signatures (multiple stack effects [5]) with essential restrictions. Set of multiple stack effects in general is not even an inverse semigroup but there always exists a subset which is a polycyclic monoid. For simplicity we can use the set of simple signatures everywhere because it represents some common properties of all inverse semigroups ([1] introduces the basic theory of semigroups).

Let *types* be denoted by a, b, c ...

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 operation finds an input argument on the stack which has the incompatible 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

$s = (s_1 -- s_2)$ 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 $\emptyset$.

$\mathbf{1} = (\ \text{—}\ )$ 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

These products are calculated using the definitions above.

$(a -- bc)(c -- de) = (a -- bde)$
$(ab -- c)(dc -- e) = (dab -- e)$
$(a -- b)(bc -- d) = \emptyset$
$(--\ ab)(ab --) = \mathbf{1}$
$(a -- b)(cb - d) = (ca -- d)$
$(ca -- cb)(cb -- d) = (ca -- d)$

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

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

and for non-zero effects the following equivalence holds

$$(--s_1)(t_1--t_2)(s_2--) = \mathbf{1} \Leftrightarrow (s_1--s_2) \leq (t_1--t_2)$$

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).

2 Stack effects as an alternative syntax definition tool

Let us have a set of stack operations Δ. We can build programs by writing sequences of stack operations (let us forget about control transfer instructions at the beginning). The set of all "programs" (including these which make no sense) is Δ^*. Each operation $p \in \Delta$ has a given stack effect $sig(p) \in \mathcal{S}$. Mapping $sig : \Delta^* \rightarrow \mathcal{S}$ is defined as homomorphism

$sig(\Lambda) = \mathbf{1}$ for empty program Λ,

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

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,

$$\Lambda \in Closed \subset Valid \subset \Delta^*$$

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

1) 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*.

2) "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 operates correctly but does not use all (bottom) elements on the stack.

3) 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 [4]). We need to bind an inequality to each grammar rule and to solve the system of inequalities in $\mathcal{S}$.

3 Syntactic equations

We defined a non-empty language $Closed(\Delta, sig)$ and 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 properties are 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. It has been shown that the stack effect calculus and general rewriting rules (syntactic equations) are equivalent in particular cases (see [6] and [7]).

Pair (Δ, sig) allows to define equations between programs in Δ^* if corresponding equations between non-zero type signatures in $\mathcal{S}$ hold:

$$\omega_1 = \omega_2 \ \Leftrightarrow\ sig(\omega_1) = sig(\omega_2) \neq \emptyset$$

Instead of introducing some kind of an "axiom" for the system of equations let us start from the empty sequence Λ ($sig(\Lambda) = \mathbf{1}$). We need at least one equation with one side empty. "Derivation step" is substitution of any subsequence (including empty sequence) to another one if there exists an according equation between (sub)sequences. Reminder – we have only "terminal" sequences in this formalism.

Example 3

The simplest non-trivial stack language has two operations: one that produces a stack item and another that consumes this item. We can describe this language in three ways.

Terminal alphabet for the language is $\Delta = \{p, q\}$

First, let us use stack effects and have
$sig(p) = (\ -- \ a)$
$sig(q) = (a \ -- \ )$
$Closed(\Delta, sig) = \{\omega \in \Delta^* : sig(\omega) = \mathbf{1}\}$

Second method is the shortest – we have just one syntactic equation (often used in theory of semigroups to define the bicyclic monoid):
$\Lambda = pq$

The most inconvenient (and fuzzy) is the context free grammar
$S_0 \to S$
$S_0 \to \Lambda$
$S \to pq$
$S \to pSq$
$S \to SS$

The grammar becomes huge and complicated when we add few more operations.

Example 4

Let us have the following system in $\mathcal{S}$
(1) $\mathbf{1} = vt$
(2) $v = qr$
(3) $t = qs$
(4) $p = ttv$
(5) $m = p$
(6) $r = rtv$

This system generates exactly the same language as the following set of stack effects (a and c are free variables in $\mathcal{T}^*$, proof in [6]).
$v = (\ -- \ c)$
$q = (\ -- \ a)$
$t = (c \ -- \ )$
$p = (c\ c \ -- \ c)$
$m = (c\ c \ -- \ c)$
$s = (c\ a \ -- \ )$
$r = (a \ -- \ c)$

Example 5

One more example (actually, this is a subset of Java virtual machine commands, see [2] for JVM).
(1) $\mathbf{1} = sr$
(2) $\mathbf{1} = smpx$
(3) $\mathbf{1} = ut$
(4) $s = mq$
(5) $q = qrs$
(6) $v = qrqrm$
(7) $z = rm$
(8) $w = ttu$
(9) $p = u$
(10) $y = pxp$

First equation gives $s_1 = \Lambda$, $r_2 = \Lambda$ and $r_1 = s_2$. Let us define $\alpha = r_1$ $(= s_2)$ for the future use. Now $s = (\ -- \ \alpha)$ and $r = (\alpha \ -- \ )$.

Fifth equation (together with facts we already know) allows to deduce that $\exists \beta \in \mathcal{T}^*$: $q_2 = \beta\alpha$.

Fourth equation can now be expressed as $(\ -- \ \alpha) = (m_1 \ -- \ m_2)\ (q_1 \ -- \ \beta\alpha)$ that gives us $m_1 = \Lambda$, $\exists \gamma \in \mathcal{T}^*$: $\alpha = \gamma\beta\alpha$ and $m_2 = \gamma q_1$.

Consequently, $\gamma = \beta = \Lambda$ and we can define $\delta = m_2 = q_1$. Now $m = (\ -- \ \delta)$ and $q = (\delta \ -- \ \alpha)$.

Equations (6) and (7) define $v = (\delta\ \delta \ -- \ \delta)$ and $z = (\alpha \ -- \ \delta)$.

Third equation is analogous to the (1): we have $u_1 = t_2 = \Lambda$ and we introduce a new sequence $\phi = u_2 = t_1$. Now $u = (\ -- \ \phi)$ and $t = (\phi \ -- \ )$.

Again, equations (8) and (9) allow direct calculations: $w = (\phi\ \phi \ -- \ \phi)$ and $p = (\ -- \ \phi)$.

Now it is time to use the second equation for $x = (smp)^{-1} = p^{-1}m^{-1}s^{-1}$ that gives us $x = (\alpha\ \delta\ \phi \ -- \ )$.

Finally, from (10) we get $y = (\alpha\ \delta \ -- \ \phi)$.

Let us sum up the result.
$m = (\ -- \ \delta)$
$p = (\ -- \ \phi)$
$q = (\delta \ -- \ \alpha)$
$r = (\alpha \ -- \ )$
$s = (\ -- \ \alpha)$
$t = (\phi \ -- \ )$
$u = (\ -- \ \phi)$
$v = (\delta\ \ \delta \ -- \ \delta)$
$w = (\phi\ \phi \ -- \ \phi)$
$x = (\alpha\ \delta\ \phi \ -- \ )$
$y = (\alpha\ \delta \ -- \ \phi)$
$z = (\alpha \ -- \ \delta)$

We still have three independent variables $\alpha, \delta, \phi \in \mathcal{T}^*$ here and we have used all the equations.

Conclusion: syntactic equations and stack effects define the same language in this example.

The following context free grammar generates only a subset of this language, but still contains 19 productions. Sequence $smqrmpx$, for example, cannot be derived in this grammar. To generate it by equations we have to apply equations (2), (1) and (4):
$\Lambda \Rightarrow smpx \Rightarrow ssrmpx \Rightarrow smqrmpx$

The fact that the grammar generates only closed sequences can be deduced by solving the corresponding system of inequalities (see [5] about building systems of inequalities for grammars).
$S \to T$
$T \to U$
$T \to TU$
$U \to A\mathbf{r}$
$U \to AMF\mathbf{x}$
$U \to F\mathbf{t}$
$A \to M\mathbf{q}$
$A \to \mathbf{s}$
$M \to N$
$M \to MN\mathbf{v}$
$N \to \mathbf{m}$
$N \to A\mathbf{z}$
$N \to M$
$F \to G$
$F \to FG\mathbf{w}$
$G \to \mathbf{p}$
$G \to AM\mathbf{y}$
$G \to \mathbf{u}$
$G \to F$

It is not realistic to express the whole language determined by syntactic equations using context free grammar of any kind – it takes hundreds of productions to cover all possible substitutions (it is not quite clear how to produce such a grammar automatically and whether it makes sense).

Much better solution is to investigate mappings between syntactic equations and sets of stack effects. Stack languages are often described in form of dictionary of stack operations that contains stack effects for these operations. If it is possible to prove that for each set of stack effects there exists the system of syntactic equations that defines the language of correct (closed) programs then we have demonstrated that stack effect calculus is at least as powerful syntax description tool as the context free grammar.

Even if it is hard to find a general method and the system of syntactic equations exists only for particular cases there is still a good chance to use these particular cases in practice, e.g. for client-side checking of Java-programs (important topic to improve the security of Internet).

4 Basic types of equations

We already used some properties of stack effects to prove that languages in example 5 coincide. The following list of cases helps us to analyse particular equations.

Equations of type $st = \mathbf{1}$

$s, t \in \mathcal{S} \;\; \& \;\; st = \mathbf{1} \;\; \Leftrightarrow$
$\Leftrightarrow \exists \alpha \in \mathcal{T}^* : \; s = (\; -- \; \alpha \;) \;\; \& \;\; t = (\; \alpha \; -- \;)$

Equations of type $rst = \mathbf{1}$

$r, s, t \in \mathcal{S} \;\; \& \;\; rst = \mathbf{1} \;\; \Leftrightarrow$
$\Leftrightarrow \exists \alpha, \beta, \gamma \in \mathcal{T}^* : \; r = (\; -- \; \alpha\beta \;) \;\; \&$
$\& \;\; s = (\; \beta \; -- \; \gamma \;) \;\; \& \;\; t = (\; \alpha\gamma \; -- \;)$

Equations of type $rs = r$

$r, s \in \mathcal{S} \;\; \& \;\; rs = r \;\; \& \;\; r \neq \emptyset \;\; \Leftrightarrow$
$\Leftrightarrow \exists \alpha, \beta, \gamma \in \mathcal{T}^* : \; r = (\; \alpha \; -- \; \beta\gamma \;) \;\; \&$
$\& \;\; s = (\; \gamma \; -- \; \gamma \;)$

Equations of type $st = t$

$s, t \in \mathcal{S} \;\; \& \;\; st = t \;\; \& \;\; t \neq \emptyset \;\; \Leftrightarrow$
$\Leftrightarrow \exists \alpha, \beta, \gamma \in \mathcal{T}^* : \; s = (\; \alpha \; -- \; \alpha \;) \;\; \&$
$\& \;\; t = (\; \beta\alpha \; -- \; \gamma \;)$

Equations of type $s = rst$

As the first step, it is easier to express the inequality $s \leq rst$ in terms of stack effects. The problem is that we can use it in one direction only ($s \mapsto rst$) when performing substitutions on corresponding sequences of stack operations.

$r, s, t \in \mathcal{S} \;\; \& \;\; s \leq rst \;\; \& \;\; s \neq \emptyset \;\; \Leftrightarrow$
$\Leftrightarrow \exists \alpha, \beta, \gamma \in \mathcal{T}^* : \; \alpha r_1 = s_1 \;\; \&$
$\& \;\; \alpha r_2 = \beta s_1 \;\; \& \;\; \gamma t_1 = \beta s_2 \;\; \& \;\; \gamma t_2 = s_2$

When analysing the equation $s = rst$, we have the following five cases ($\alpha, \beta, \gamma, \delta \in \mathcal{T}^*$):

$(r_1 \; -- \; r_2) \; (\alpha r_2 \equiv \gamma\alpha r_1 \; -- \; t_2) \; (\gamma t_2 \; -- \; t_2)$
$(r_1 \; -- \; \beta r_1) \; (r_1 \; -- \; \beta\delta t_2 \equiv \delta t_1) \; (t_1 \; -- \; t_2)$
$(r_1 \; -- \; \beta\gamma r_1) \; (\gamma r_1 \; -- \; t_2) \; (\gamma\beta t_2 \; -- \; t_2)$
$(s_1 \; -- \; \beta\gamma s_1) \; (s_1 \; -- \; \beta t_2) \; (\gamma\beta t_2 \; -- \; t_2)$
$(r_1 \; -- \; r_1) \; (\alpha r_1 \; -- \; \delta t_1) \; (t_1 \; -- \; t_1)$

5 Conclusions

Interest to stack machines and stack based languages is not extinguished but gains new popularity in context of developing platform independent software (e.g. Java). There are several areas where formal analysis of stack programs may be useful – debugging software, security checks when downloading software from Internet, compilation, optimisation, etc.

Syntax of stack based languages is hard and unnatural to express using traditional context free grammars. There are two alternative possibilities – stack effect calculus and system of syntactic equations. For particular cases we can prove that both methods define the same language. To achieve more general solutions some basic types of equations are analysed.

Syntactic equations allow to apply pattern matching techniques to the analysis of programs in stack based languages. The other form – stack effects – is human readable and often used in definition of stack machine operations. Both methods have advantages in case of stack based languages that do not have traditional "strong" syntax.

Acknowledgements

This work has been supported by Estonian Science Foundation, grant no. 3276.

References

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

[2] *The Java Virtual Machine Specification,* Sun Microsystems, 74 pp., 1995.

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

[4] 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, pp. 282 – 290, 1991.

[5] 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, pp. 400 – 406, 1992.

[6] Pöial J., "Forth and Formal Language Theory," *EuroForth'94, Nov 4 – 6, 1994*, Winchester, UK, pp. 47 – 52, 1994.

[7] Pöial J., "Validation of Stack Effects in Java Bytecode," *Proc. of the Fifth Symposium on Programming Languages and Software Tools, June 7 – 8, 1997, Jyväskylä, Finland,* Report C-1997-37, Department of Computer Science, Univ. Helsinki, pp. 128 – 134, 1997.

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