Λ-superposition of Stack Languages

Jaanus Pöial *

Abstract

There exists a class of widespread languages that use stack machines for interpretation of programs, the so called stack based languages (Java virtual machine language, Forth, Postscript, several intermediate program representation languages and low level languages in some embedded systems). Semantics of stack operations determines the language of correct programs in sense of parameter passing through the stack. This is one alternative method to define the syntax of a stack based language, the so called stack effect calculus. The other method is based on systems of syntactic equations (general rewriting rules for terminal sequences) on sequences of stack operations. Both methods seem to have better expression power for the stack based languages than traditional context free grammars.

Below we introduce stack language using the stack effect calculus approach and define a new operation on stack languages called Λ-superposition. This operation reflects a naïve way of combining stack machine programs that seems to cover all possibilities. But it occurs that given two command systems both defining a stack language the resulting "mixture" of these two systems may generate more powerful language than Λ-superposition of initial languages. If that is not the case we call these systems non-intervening. Using non-intervening command systems it is easy to predict behavior of mixed programs (it is restricted to properties of Λ-superposition) otherwise we need to consider synergetic effects.

Key words: Programming Languages, Program Analysis, Stack Machines, Formal Methods, Execution Environments, Formal Language Theory, Inverse Semigroups

1 Introduction

Code analysis is an important issue in verification, optimization, parallelization, transformation and other formal manipulations on computer programs (see e.g. [3] about program analysis). In many cases the code is interpreted by stack machines like Java virtual machine, some (embedded) Forth system, Postscript device, Common Language Runtime, etc. Stack machine code is an interesting and amazingly non-trivial mathematical object to describe. But without a formal model it is impossible to solve any of the problems mentioned above.

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

Stack operations are often described using stack effects defining what are the input parameters and what are the output parameters of an operation. Stack effects as a formal model was introduced in [5]. This theory was applied to code generation using syntax directed translation schemes (verification of a scheme instead of verification of individual targets, see [11] about compiler technique). In [6] stack machine programs with control structures were investigated. Similar approach to stack effect calculus was also used in [12]. It is not always easy to express the stack language determined by stack effects using context free grammars (see examples in [10]). Floyd-Hoare style formal axiomatics to stack based languages was partially defined in [7].

In [8] the idea of syntactic equations to describe stack machine code sequences was introduced. Such equations also describe a language that can be equal to the language determined by stack effects. Later this approach was applied to Java byte-code patterns ([9]).

When we have two stack languages each defined by some set of stack effects we need to know if it is safe to run programs of these languages on the same stack machine simultaneously. Or, vice versa, is it possible to split certain command system to "independent" parts that can run on different machines.

In this paper we introduce two new notions to support investigation of these two problems. At first, we model a "safe combination" of two stack languages via Λ-superposition and then prove that not all possible combinations are safe. Secondly, we define non-intervening command systems that generate only "safe combinations".

Properties of stack based languages and their implementation have been investigated also by other groups (e.g. [1] and lately [2]).

2 Preliminaries

We recall some basic definitions from [10].

Let $\mathcal{T}$ be an alphabet (a finite set of different "types").

We denote the "stack effect" $(\alpha, \beta) \in \mathcal{T}^* \times \mathcal{T}^*$ by $(\ \alpha\ \ --\ \beta\ )$ and introduce the zero element $\mathbf{0}$ to report errors: $(\infty, \infty) = \mathbf{0}$, α is a list of input parameter types passed through the stack and β is a list of resulting output parameter types (rightmost element corresponds to the top of the stack). $\mathbf{0}$ indicates type mismatch when a stack operation receives input of "wrong" type.

The set of **stack effects** is defined: $\mathcal{S} = (\mathcal{T}^* \times \mathcal{T}^*) \cup \{\mathbf{0}\}$

The composition (product) of stack effects is determined by three rules

(1) $\forall s \in \mathcal{S} : s\mathbf{0} = \mathbf{0}s = \mathbf{0}$

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

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

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

(3) In all other cases the result is $\mathbf{0}$.

Pair ($--$) is the unit for composition and we use symbol $\mathbf{1}$ to express this.

This algebraic structure is (isomorphic to) the polycyclic monoid (see [4]) - certain 0-bisimple inverse semigroup.

Let us have a set of **stack operations** Δ.

Mapping $sig : \Delta^* \to \mathcal{S}$ that binds stack effects to stack operations is defined as homomorphism

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

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

Let us define a **stack language** $L(\Delta)$ as

$$L(\Delta) = \{w \mid w \in \Delta^* : sig(w) = \mathbf{1}\}$$

As we see the empty word Λ is always included.

3 Λ-superposition

Let $\Gamma \subset \Delta$ be a subset of stack commands. This set is said to be a command system if all the commands are "useful" in sense of

$$\forall p \in \Gamma \;\; \exists w_1, w_2 \in \Gamma^* : w_1 p w_2 \in L(\Gamma)$$

For empty set $\emptyset$ we define $L(\emptyset) = \{\Lambda\}$.

Proposition 1 *If $\Gamma_1 \subset \Delta$ and $\Gamma_2 \subset \Delta$ then*

$$L(\Gamma_1) \cap L(\Gamma_2) = L(\Gamma_1 \cap \Gamma_2)$$

$$L(\Gamma_1) \cup L(\Gamma_2) \subset L(\Gamma_1 \cup \Gamma_2)$$

Proof. $\Gamma_1 \cap \Gamma_2$ is a subset both in Γ_1 and Γ_2. If $w \in L(\Gamma_1 \cap \Gamma_2)$ then, obviously, $w \in L(\Gamma_1)$ and $w \in L(\Gamma_2)$, i.e.

$$L(\Gamma_1 \cap \Gamma_2) \subset L(\Gamma_1) \cap L(\Gamma_2)$$

If $w \in L(\Gamma_1)$ and $w \in L(\Gamma_2)$ then $w \in (\Gamma_1 \cap \Gamma_2)^*$. We also know that $sig(w) = \mathbf{1}$. Consequently, $w \in L(\Gamma_1 \cap \Gamma_2)$.

The second inclusion also follows from the definitions. $L(\Gamma_1 \cup \Gamma_2)$ may contain "mixed" sequences, so, in general, the equality does not hold. □

Definition 1 *Λ-superposition $\circ$ of stack languages $L(\Gamma_1)$ and $L(\Gamma_2)$ is defined as follows*

(1) If $w \in L(\Gamma_1) \cup L(\Gamma_2)$ then $w \in L(\Gamma_1) \circ L(\Gamma_2)$;

(2) If $u'u'' \in L(\Gamma_1) \circ L(\Gamma_2)$ and $v \in L(\Gamma_1) \circ L(\Gamma_2)$ then $u'vu'' \in L(\Gamma_1) \circ L(\Gamma_2)$;

(3) There are no other words in the Λ-superposition than these introduced by (1) and (2).

This definition is symmetric w.r.t. Γ_1 and Γ_2. We can substitute the empty word Λ to any existing word w (where $sig(w) = \mathbf{1}$) anywhere we want to. That is why the operation is called "Lambda-superposition".

Theorem 1 *The following subset relations hold*

$$L(\Gamma_1) \cup L(\Gamma_2) \subset L(\Gamma_1) \circ L(\Gamma_2) \subset L(\Gamma_1 \cup \Gamma_2)$$

Proof. The first inclusion is trivial following from (*1*).

For the second part we first notice that $L(\Gamma_1) \cup L(\Gamma_2) \subset \Gamma_1^* \cup \Gamma_2^* \subset (\Gamma_1 \cup \Gamma_2)^*$. Having words $u', u'', v \in (\Gamma_1 \cup \Gamma_2)^*$ also $u'vu'' \in (\Gamma_1 \cup \Gamma_2)^*$ holds. If we choose u', u'' and v as stated in (*2*) we get $L(\Gamma_1) \circ L(\Gamma_2) \subset (\Gamma_1 \cup \Gamma_2)^*$.

We know that $w \in L(\Gamma_1) \cup L(\Gamma_2)$ yields $sig(w) = \mathbf{1}$.

If we have $sig(u'u'') = \mathbf{1}$ and $sig(v) = \mathbf{1}$ then

$$sig(u'vu'') = sig(u')sig(v)sig(u'') = sig(u')sig(u'') = sig(u'u'') = \mathbf{1}$$

Consequently, $u'vu'' \in L(\Gamma_1 \cup \Gamma_2)$ and the theorem is proved. □

Example 1 *Let* $\mathcal{T} = \{k, l, m\}$,
$\Gamma_1 = \{a, b, c, d\}$,
$\Gamma_2 = \{e, f, g\}$
and
$sig(a) = (\ --\ \ k\ )$
$sig(b) = (\ k\ \ --\ l\ )$
$sig(c) = (\ l\ \ --\ k\ )$
$sig(d) = (\ l\ \ --\ )$
$sig(e) = (\ --\ \ m\ k\ )$
$sig(f) = (\ k\ \ --\ )$
$sig(g) = (\ m\ \ --\ )$

We can calculate signature products immediately for the following words to show that:
$abd,\ aabdbcbd \in L(\Gamma_1)$
$efg,\ efefggefg \in L(\Gamma_2)$
$abefgd,\ eabcbdfg,\ eaefgbdfg \in L(\Gamma_1) \circ L(\Gamma_2) \setminus (L(\Gamma_1) \cup L(\Gamma_2))$
$af,\ ebcfg,\ ebdg \in L(\Gamma_1 \cup \Gamma_2) \setminus (L(\Gamma_1) \circ L(\Gamma_2))$

This example demonstrates that, in general, all subset relations in the theorem are strict. These two command systems put together produce more programs than the result of their Λ-superposition (Λ-superposition is not powerful enough to express the process of construction of stack languages).

Definition 2 *Two command systems* $\Gamma_1, \Gamma_2 \subset \Delta$ *are said to be non-intervening if* $L(\Gamma_1) \circ L(\Gamma_2) = L(\Gamma_1 \cup \Gamma_2)$.

This definition reflects the situation when there are no synergetic effects resulting from the cooperation of two command systems (new stack language is just the Λ-superposition of old ones).

Example 2 *Let* $\mathcal{T} = \{k, l\}$

$\Gamma_1 = \{a, b\}$

$\Gamma_2 = \{b, c, d\}$

$sig(a) = (\ --\ \ k\ )$

$sig(b) = (\ k\ \ --\ )$

$sig(c) = (\ --\ \ l\ k\ )$

$sig(d) = (\ l\ \ --\ )$

It is possible to check that $ab \in L(\Gamma_1)$ *and* $cbd \in L(\Gamma_2)$. *These two words, on the other hand, define* $L(\Gamma_1 \cup \Gamma_2)$.

Let us forget about given signatures and introduce the following variables:

$x = sig(a) \qquad y = sig(b)$

$z = sig(c) \qquad w = sig(d)$

The system of equations (in $\mathcal{S}$*)*

$xy = \mathbf{1}$

$zyw = \mathbf{1}$

has the only solution:

$x = (\ --\ \ y_1\ ) \qquad y = (\ y_1\ \ --\ )$

$z = (\ --\ \ w_1\ y_1\ ) \qquad w = (\ w_1\ \ --\ )$

where we are free to choose y_1 *and* w_1. *In [10] some techniques of solving such systems are presented.*

Consequently, the language $L(\Gamma_1 \cup \Gamma_2)$ *is defined by equations* $sig(ab) = \mathbf{1}$ *and* $sig(cbd) = \mathbf{1}$.

But $L(\Gamma_1) \circ L(\Gamma_2)$ *gives the same result and we have two non-intervening command systems by definition.*

This example demonstrates that two command systems can have both common and individual commands and still be non-intervening.

Proposition 2 *If* $\Gamma_1 \subset \Gamma_2$ *then* Γ_1 *and* Γ_2 *are non-intervening.*

Proof. $L(\Gamma_1 \cup \Gamma_2) = L(\Gamma_2) \subset L(\Gamma_1) \cup L(\Gamma_2)$. Now we use the theorem and conclude that all three languages above have to be equal. $\Box$

4 Conclusion

There are different methods to define stack languages - stack effect calculus, syntactic equations, grammars, Floyd-Hoare semantics, etc. All these definitions have their benefits and drawbacks depending on the problems we want to solve. But the language itself as a mathematical object remains the same whatever method we choose.

Λ-superposition is an operation on languages. It is a natural way to combine stack languages and this paper indicates possibilities and restrictions of using this operation in program analysis for stack based execution environments.

References

[1] Ertl M.A., "Implementation of Stack-Based Languages on Register Machines," *Dissertation, Vienna Technical University*, 83 pp., 1996.

[2] Gassanenko M.L., "Threaded Code Execution and Return Address Manipulations from the Lambda Calculus Viewpoint," *EuroForth'99, September 17 – 20, Sankt-Petersburg*, 17 pp., 1999.

[3] Nielson F., Nielson H.-R., Hankin C., "Principles of Program Analysis," Springer, 450 pp., 1999.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] Pöial J. "Alternative Syntactic Methods for Defining Stack Based Languages," Proceedings of NWPER'98 Nordic Workshop on Programming Environment Research, Reports in Informatics No 152, University of Bergen, Norway, June 1998, 227 – 232.

[11] 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.

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