Stack effect calculus with typed wildcards, polymorphism and inheritance

Jaanus Pöial

University of Tartu, Estonia

Abstract

In early 1990s author introduced a formal stack effect calculus for verification of compilers that translated high level languages (Fortran, Modula) into Forth, see [PST90], [P90a], [P90h]. The calculus was partially applicable to static type checking of Forth programs, but this was not the primary goal these days. Stack effects (formal specifications of input and output parameters for stack operations) were defined using flat type space where different types were considered incompatible and no subtyping or inheritance was allowed. The so called wildcard types were introduced by sets of stack effects, see [P91]. This framework does not suite well with abstract stack machines that use principles of object orientation (see, for example, [AG98] about type checking in Java Virtual Machine). Peter Knaggs and Bill Stoddart improved the type signature algebra and introduced a lot of useful things (type variables, subtyping, reference types, wildcards, etc.), see [SK93], [K93].

In this presentation a modified framework for type checking is proposed to support typed wildcards and inheritance. Now it is possible to perform little more exact type calculations and express polymorphic operations. Every type symbol has its place in the type hierarchy and, at the same time, it may be treated as a wildcard symbol. Earlier approaches matched wildcards to concrete symbols (resulting in this concrete symbol) or to other wildcards (resulting in a new wildcard); this approach is more general allowing stepwise refinement of types. Not only the type checking is target here, but also the (static) choice of the right version for polymorphic operations (known as method overloading in object oriented languages). Given a type hierarchy, formal specifications for operations and a program we can refine the type signatures in the program according to the context where an operation appears. Experimental implementation of this framework is in progress.

References

[PST90] Tombak M., Soo V., Pöial J. *A Forth-Oriented Compiler Compiler and its Applications*. FORTH Dimensions (ISSN 0884-0822), Vol XVI No 5, Jan-Feb 1995, Forth Interest Group, Oakland, USA, 21-22.

[P90a] Pöial J. Algebraic *Specification of Stack Effects*. FORTH Dimensions (ISSN 0884-0822), Vol XVI No 4, Nov-Dec 1994, Forth Interest Group, Oakland, USA, 18-20.

[P90h] Pöial J. *A Bit of History*. FORTH Dimensions (ISSN 0884-0822), Vol XVI No 4, Nov-Dec 1994, Forth Interest Group, Oakland, USA, p. 17, 20.

[P91] Pöial J. *Multiple Stack-effects of Forth Programs*. 1991 FORML Conf. Proceedings, euroFORML'91 Conference, Oct 11 - 13, 1991, Marianske Lazne, Czechoslovakia, Forth Interest Group, Oakland, USA, 1992, 400-406.

[SK93] Bill Stoddart, Peter J. Knaggs. *Type Interference in Stack Based Languages*. Formal Aspects of Computing 5(4): 289-298 (1993).

[K93] Peter J. Knaggs. *Practical and Theoretical Aspects of Forth Software Development*. PhD thesis, School of Computing and Mathematics, University of Teesside, Middlesbrough, Cleveland, UK, March 1993.

[AG98] Allen Goldberg. *A Specification of Java Loading and Bytecode Verification*. In Proc. 5th ACM Conference on Computer and Communications Security (CCS'98), pages 49-58, October 1998.

This work was supported by Estonian Science Foundation grant No. 5279

Stack effect calculus with typed wildcards, polymorphism and inheritance

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

Supported by Estonian Science Foundation grant No 5279

Goal 1: early checking for possible problems when using stack machines and stack based languages

- ♦ Type checking in stack based languages
- ♦ “Postfix” is hard to read and maintain by human programmer => programming tools, intelligent editors,…
- ♦ Validation of code generation tools (e.g. compilers)

Goal 2: formal manipulations on stack programs

- optimisation
- transformation to different execution architectures
- parallelisation

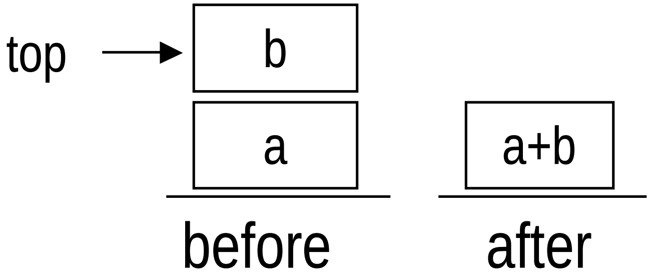
History

- 1990 – compiler compiler project:
  - Mati Tombak, Viljo Soo, Jaanus Pöial
    - Compilers for Modula 2 and Fortran
    - Formal stack effect calculus introduced for rough checking of translators (whether all possible generated Forth-programs are “type correct”)
    - Type checking of Forth programs was not interesting at the beginning
- 1990-1994 – type inference theory, mainly targeted to type checking:
  - Bill Stoddart, Peter Knaggs, Jaanus Pöial

Stack effects

Informal description

	OPERATION	STACK EFFECT	DESCRIPTION
e.g.	**+**	(a b -- a+b)	add two topmost elements



Stack effect calculus, the first approach

T - operand types (`char, flag, addr,` ...)

$\mathbf{T}^*$ - type lists (last type on the top)

Ø - type clash symbol (stack error)

The set of stack effects:

$$\mathbf{S} = (\mathbf{T}^* \times \mathbf{T}^*) \cup \{ \text{Ø} \}$$

$$(a \rightarrow b)$$

input parameters (types) output parameters (types)

Composition (multiplication)

For all s in **S**: $s \cdot \varnothing = \varnothing \cdot s = \varnothing$

For all a, b, c, d, e, f in $\mathbf{T}^*$:

$$\left|\begin{array}{l} (a \to b) \cdot (eb \to d) = (ea \to d) \\ (a \to fc) \cdot (c \to d) = (a \to fd) \\ \varnothing\text{, otherwise} \end{array}\right.$$

$\varnothing$ is zero

$1 = (\ \to\ )$ is unity for this operation

S is polycyclic monoid

Drawbacks

- Type system was too simple (“flat”)

  Wildcards as sets, control structures implemented using sets (J.Pöial, 1991)

  Type signatures with wildcards, type variables, subtyping, reference types, ... (B.Stoddart, P.Knaggs, 1991-93)

- Support for object oriented features was not developed enough: e.g. polymorphism, overloading, …

Second approach

- Type hierarchy (“more exact type wins”)
- Wildcards are typed and numbered uniquely in scope of analysis
- Composition is defined by rules (like in Stoddart-Knaggs approach) that affect the whole scope of analysis
- Polymorphism is supported by replacing the concept of wildcard from “the same element” to the “element of the same type”:

  Example. Let flag<x and n<x, where flag $\perp$ n

  plus (x[1] x[1] $\to$ x[1]) can be refined to

  (flag[1] flag[1] $\to$ flag[1]) or (n[1] n[1] $\to$ n[1])

Notation

t, u, … - types (just symbols)

$t \le u$ – t is subtype of u (t is more exact) or equal to u (subtype relation is transitive)

$t \perp u$ - t and u are incompatible types

t^i - type symbols with “wildcard” index

(index is unique for “the same type”)

a, b, c, d, … - type lists (top right) that represent the stack state

Notation (cont.)

$s = (a \rightarrow b)$ – stack effect (a – stack state before the operation, b – after)

$\varnothing$ - type clash (zero effect)

$(a \rightarrow b)\cdot(c \rightarrow d)$ - composition of stack effects $(a \rightarrow b)$ and $(c \rightarrow d)$ defined by rules

x, y – sequences of stack effects

y, where $u^j := t^k$ – substitution: all occurances of u^j in all type lists of sequence y are replased by t^k, where k is unique index over y

Rules

$$\frac{x \cdot \emptyset}{\emptyset} \qquad \frac{\emptyset \cdot x}{\emptyset}$$

$$\frac{x \cdot (a \rightarrow b) \cdot (\ \rightarrow d)}{x \cdot (a \rightarrow bd)} \qquad \frac{x \cdot (a \rightarrow\ ) \cdot (c \rightarrow d)}{x \cdot (ca \rightarrow d)}$$

$$\frac{x \cdot (a \rightarrow bt) \cdot (cu \rightarrow d), \text{ where } t \perp u}{\emptyset}$$

Rules (cont.)

$$\frac{x\cdot\left(a\rightarrow bt^{i}\right)\cdot\left(cu^{j}\rightarrow d\right),\ \text{where}\ \ t\leq u}{x\cdot\left(a\rightarrow b\right)\cdot\left(c\rightarrow d\right),\text{where}\ t^{i}:=t^{k}\ \ and\ \ u^{j}:=t^{k}}$$

$$\frac{x\cdot\left(a\rightarrow bt^{i}\right)\cdot\left(cu^{j}\rightarrow d\right),\ \text{where}\ \ u\leq t}{x\cdot\left(a\rightarrow b\right)\cdot\left(c\rightarrow d\right),\text{where}\ t^{i}:=u^{k}\ \ and\ \ u^{j}:=u^{k}}$$

Example (small subset)

- Type system:

  a-addr < c-addr < addr < x

  flag < x

  char < n < x

Example (cont.)

- Words and specifications:

DUP	(x[1] -- x[1] x[1])	
DROP	(x --)	
SWAP	(x[2] x[1] -- x[1] x[2])	
ROT	(x[3] x[2] x[1] -- x[2] x[1] x[3])	
OVER	(x[2] x[1] -- x[2] x[1] x[2])	
PLUS	(x[1] x[1] -- x[1])	“same type”
+	(x x -- x)	
@	(a-addr -- x)	
!	(x a-addr --)	
C@	(c-addr -- char)	
C!	(char c-addr --)	
DP	(-- a-addr)	
0=	(n -- flag)	

Example (cont.)

- “Programs” (should be live demo)

  Simple one:

  ```
  SWAP  SWAP
  ```

  Conflict:

  ```
  C@  !
  ```

  More exact analysis:

  ```
  0=  +  0=
  0= PLUS  0=
  ```

  Information moving backwards:

  ```
  OVER OVER + ROT ROT + C!
  OVER OVER PLUS ROT ROT PLUS  C!
  OVER OVER PLUS ROT ROT PLUS
  OVER OVER + ROT ROT PLUS C!
  OVER OVER PLUS ROT ROT + C!
  ```

Results

- Rules for composition of stack effects are modified to support subtyping (partially this was done in [StK93]). This is prerequisite for handling inheritance and operation overloading.
- Wildcards are interpreted in more general sense that allows to (statically) introduce polymorphic words (e.g. PLUS might be + for numbers and AND for Boolean flags).
- Composition is context sensitive: it affects the whole scope of analysis and it is possible to calculate exact signatures for polymorphic operations.
- Experimental implementation is in progress.