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

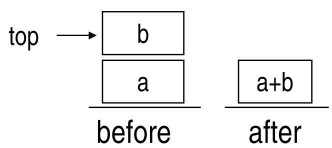
History

- 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:

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

$$(a \rightarrow b)$$

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 $\mathbf{T}^*$:

$$\begin{cases} (a \rightarrow b) \cdot (eb \rightarrow d) = (ea \rightarrow d) \\ (a \rightarrow fc) \cdot (c \rightarrow d) = (a \rightarrow fd) \\ \varnothing\text{, otherwise} \end{cases}$$

Ø is zero

$1 = (\rightarrow)$ 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] → x[1]) can be refined to
  (flag[1] flag[1] → flag[1]) or (n[1] n[1] → 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

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

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

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

Rules (cont.)

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

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

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.