Java Framework for Static Analysis of Forth Programs

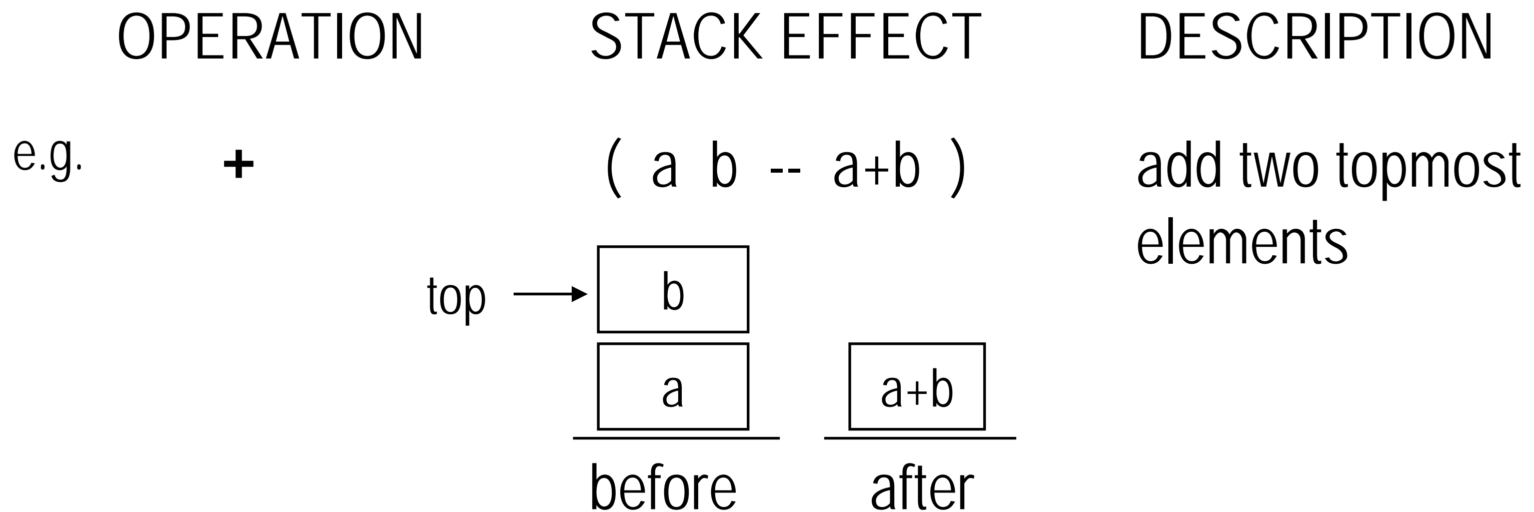
Jaanus Pöial

The Estonian Information Technology College

supported by Estonian Science Foundation grant no. 6713

Stack effects

Informal description



Stack effect calculus – 1990-s

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:

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

$$(a \rightarrow b)$$

input parameters (types) output parameters (types)

Composition (multiplication)

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

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

$(a \to b) \cdot (eb \to d) = (ea \to d)$

$(a \to fc) \cdot (c \to d) = (a \to fd)$

Ø, otherwise

Ø is zero

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

S is polycyclic monoid

Notation for rule based approach

t, u, … - types (just symbols)

t ≤ u – t is subtype of u (t is more exact) or equal to u (subtype relation is transitive)

t ⊥ u - t and u are incompatible types

t^i - type symbols with “wildcard” index
(index is unique for “the same type”)

Notation (cont.)

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

$s = (a \rightarrow b)$ – stack effect

(a – stack state before the operation, b – after)

Ø - type clash (zero effect)

Notation (cont.)

$(a \to b) \cdot (c \to d)$ - composition of stack effects $(a \to b)$ and $(c \to d)$ 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 replaced by t^k, where k is unique index over y

Rules

$$\frac{x \cdot \varnothing}{\varnothing}$$

$$\frac{\varnothing \cdot x}{\varnothing}$$

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

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

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

Rules (cont.)

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

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

Greatest lower bound

$$\frac{s \sqcap \mathbf{0}}{\mathbf{0}} \qquad \frac{r \sqcap s}{s \sqcap r}$$

If there exist type lists $a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3$ such that for all elements of the lists these subtyping relations hold elementwise

$a_3 = min(a_1, a_2)$
$b_3 = min(b_1, b_2)$
$c_3 = min(c_1, c_2)$
then the following rule is applicable, in all other cases the result is zero.

$$\frac{(c_1 a_1 \rightarrow c_2 b_1) \sqcap (a_2 \rightarrow b_2)}{(c_3 a_3 \rightarrow c_3 b_3)}$$

Loop invariant

$$\sqcap^* s = s \sqcap (s \cdot s)$$

The result of this operation is an idempotent element that most precisely describes the loop body s.

Handling branches and loops

Greatest lower bound operation and loop invariants in use:

$$\frac{s(\ \texttt{IF}\ \alpha\ \texttt{ELSE}\ \beta\ \texttt{THEN}\ )}{(\texttt{flag} \rightarrow) \cdot [s(\alpha) \sqcap s(\beta)]}$$

$$\frac{s(\ \texttt{BEGIN}\ \alpha\ \texttt{WHILE}\ \beta\ \texttt{REPEAT}\ )}{\sqcap^{*}[s(\alpha) \cdot (\texttt{flag} \rightarrow)] \cdot \sqcap^{*} s(\beta)}$$

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 )
```

Examples with control structures

```
: test1
  IF
    ROT
  ELSE
    @
  THEN ;
```

(a-addr[1] a-addr[1] a-addr[1] --
a-addr[1] a-addr[1] a-addr[1])

Examples (cont.)

```
: test2
   BEGIN
      SWAP OVER
   WHILE
      NOT
   REPEAT ;
                  (flag[1] flag[1] -- flag[1] flag[1])
: test3
      OR FALSE SWAP ;
```

Written in Java

Supports stack effect calculus:

- Evaluation of sequences of stack operations
- Greatest lower bound operation (evaluation of alternative branches)
- Finding loop invariants (idempotents)

Extensible (currently 8 basic classes)

Class TypeSymbol

Objects consist of two fields:

- Type name – symbolic name of a stack item, like:

  a-addr < c-addr < addr < x

  The typesystem uses type names as keys to access information about types
- Position index – used for stack manipulation words (like SWAP ROT OVER ...) to indicate that items have the same type (sometimes the item itself remains the same):

  SWAP (x[2] x[1] -- x[1] x[2])

  If positions are not important the index is zero

Matching

When two typesymbols match (describe the same data item on the stack) a new symbol is created that has minimal (most exact) type and new “fresh” position index (only if position index is not zero). Both matching symbols are replaced by this new symbol:

SWAP DUP @

(x[2] x[1] -- x[1] **x[2]**) (**x[1]** -- x[1] x[1]) (a-addr -- x)

(x[3] x[1] -- x[1] x[3]) (x[3] -- x[3] **x[3]**) (**a-addr** -- x)

(a-addr[**4**] x[1] -- x[1] a-addr[**4**])

(a-addr[**4**] – a-addr[**4**] a-addr[**4**]) (a-addr[**4**] -- x)

(a-addr[2] x[1] -- x[1] a-addr[2])

(a-addr[2] – a-addr[2] a-addr[2]) (a-addr[2] -- x)

Result is: (a-addr[2] x[1] -- x[1] a-addr[2] x)

Class TypeSystem

- Keeps information about subtyping.
- Type name is used as a key to access the matrix of relations between types: "incompatible", "subtype", "supertype", "synonym"
- Parameter of evaluation (we can evaluate the same program against different typesystems)

Class Tvector

Vector (list) of typesymbols

Describes the stack state (top right)

Class Spec

Describes the stack effect (specification)

- Vector left_side – stack state before execution
- Vector right_side – stack state after execution
- Workfields, like string for scanner words: ." (

Contains *glb* and *idemp* operations:

sp_glb = sp1.*glb* (sp2, typesys, specset)

sp_loop = sp_body.*idemp* (typesys, specset)

Class SpecSet

- Set of stack effects in use to evaluate a program
- Dynamic mapping from Forth words to corresponding stack effects
- Parameter of evaluation (we can evaluate the same program against different specsets, e.g. run-time vs. compile-time).

Class SpecList

Linear sequence of stack effects

Contains the *evaluate* operation for sequences:

spec = speclist.*evaluate* (typesys, specset)

Composition of stack effects is a particular case of evaluation

Represents the Forth program (currently implemented only for linear sequences of words)

If Forth text is pre-processed to discover syntactic structures

(it is better to avoid this approach – in principle there are no syntactic structures in Forth)

the result is stored as ProgText

Contains the main method

Includes a small demo to add annotations (formal stack comments) to the linear sequence of words

Used for testing

Further work

- Re-write in Forth
- Re-write for IDE (like Eclipse plugin: http://www.eclipse.org)
- Handle extensibility (everything can change: control structures, defining words, new scanner words, ...)

Pseudo-Forth

```
=DEF= PROG
PROG = ELEM / PROG ELEM .
ELEM = SIMPLE / DEFINITION .
SIMPLE = WORD / PARSER / CONSTANT .
WORD = <word> .
PARSER = PARSER<delim> / COMMENT .
PARSER<delim> =  WORD <string><delim> .
COMMENT = <comment> .
CONSTANT = <constant> .
DEFINITION = VARDEF / CONSTDEF / COLONDEF .
VARDEF = 'VARIABLE' NAME / 'CREATE' NAME .
CONSTDEF = SIMPLIST 'CONSTANT' NAME .
COLONDEF = ':' NAME CONTENT ';' /
        ':' NAME CONTENT 'CREATE' CONTENT 'DOES>' CONTENT ';' .
NAME = <word> .
CONTENT = CELEM / CONTENT CELEM / .                # CONTENT might be empty
CELEM = SIMPLE / STRUCTURE .
STRUCTURE = 'IF' CONTENT 'THEN' /
        'IF' CONTENT 'ELSE' CONTENT 'THEN' /
        'BEGIN' CONTENT 'WHILE' CONTENT 'REPEAT' /     # other structures similar
        '[' SIMPLIST ']' .
SIMPLIST = SIMPLE / SIMPLIST SIMPLE .
=END=
```