Typing Tools for Typeless Stack Languages

Jaanus Pöial

The Estonian Information Technology College

supported by Estonian Science Foundation grant no. 6713

Typeless stack language

- The same stack is used to pass parameters of different types
- No type information is available at runtime – just "cells"
- Type information is hardly ever used even at compile time – it is only in programmers mind

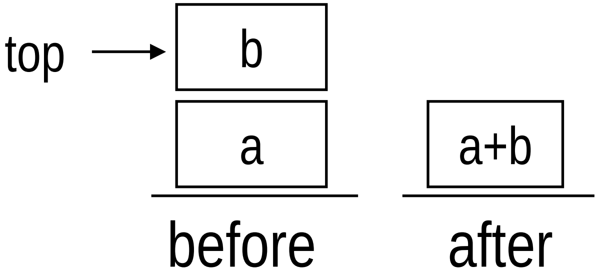
Typing

- Typing is not part of the language but part of code conventions and discipline – e.g. stack effect descriptions in Forth
- It is possible to introduce separate type checking tools (program analysis tools) on text level by extracting formal typing information from informal stack comments

Stack effects

Informal description

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



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**: $s \cdot \varnothing = \varnothing \cdot s = \varnothing$

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

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

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

$\varnothing$, otherwise

$\varnothing$ is zero

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

S is polycyclic monoid

Notation for rule based approach

t, u, ... - types (just symbols)

$t \leq u$ – 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”)

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 replased by t^k, where k is unique index over y

Rules

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

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

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

Rules (cont.)

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

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

"Must" vs. "may"-analysis

- "What is the possible stack state in a given program point? What might happen?" Impracticable question (hard to calculate, huge state space, unclear result), discussed in authors 1991 EuroForth paper

- “What guarantees that the stack state in a given program point is … ? What must happen?” Allows to find errors, easy to calculate using *glb*.

  Example: two *if*-branches have different stack effects

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

"May"-style (no implementation)

$$\frac{s(\ \texttt{IF}\ \alpha\ \texttt{ELSE}\ \beta\ \texttt{THEN}\ )}{[(\texttt{true}\rightarrow)\cdot s(\alpha)]\sqcup[(\texttt{false}\rightarrow)\cdot s(\beta)]}$$

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

Handling branches and loops

"Must"-style (abstraction)

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

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

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 )
```

Example (cont.)

Simple program:

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!

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 ;

: test3
    OR FALSE SWAP ;
```

Results

- Theoretical framework for stack analysis
- Implemented (in Java):
  - composition (for linear code)
  - greatest lower bound operation (for branching)
  - nearest idempotent (for loop invariants)