Programmianalüüsi raamistik magasinkeeltele

Jaanus Pöial

Tartu Ülikooli arvutiteaduse instituut,

ETF grant nr. 5279

jaanus@cs.ut.ee

Programmianalüüs

- Staatiline programmianalüüs - võimaldab uurida programmide omadusi ning teha otsustusi programmi käitumise kohta ilma programmi käivitamata. Eeldused:
  - Formaalne semantika kasutatava keele jaoks
  - Analüüsi raamistik: doomenid, reeglid, püsipunktide arvutus, jne.

Klassikalised analüüsid: "surnud" kood, muutujate väärtustamine, avaldiste optimiseerimine, ...

Võimalikud rakendused

- Analüsaatorid konkreetsete omaduste uurimiseks: selgitada "kahtlased" kohad olemasolevates (tihti suuremahulistes) programmides
- Kodeerija abivahendid: eriomadustega toimetid, kompilaatoritesse sisseehitatud kontrollivahendid, s.h. tüübikontroll, koodi valideerijad jne.

Kõrgendatud nõudmistega tarkvara loomisel on analüüsivahendid hinnas

Magasinorienteeritud keeled

- Pigem vahekeeled kui programmeerimiskeeled (näit. PostScript)
- Populaarsed piiratud ressurssidega keskkondades (näit. BIOS, smartcard, ...)
- Otseseks programmeerimiseks harjumatud, koostatud programme on raske lugeda

Tuntumad: Java virtuaalmasina kood, Forth, PostScript (praktiliselt kõik "virtuaalmasinad" taasleiutavad Forthi)

Eesmärgid

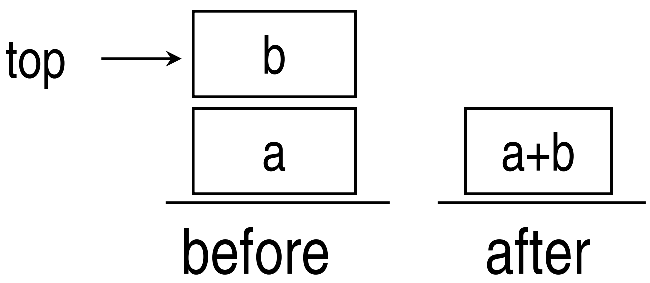
- Magasinkeelte formaalse semantika kirjeldamine (osaliselt tehtud)
- Analüüsi raamistiku loomine (käesolev ettekanne)
- Rakendamine (näiteks tööstusliku koodi analüüsimiseks - Giesecke & Devrient smart card???)

  Olemasolevatest süsteemidest lähedane on AbsInt StackAnalyzer: http://www.absint.com/aicall/stack.htm

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:

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

$$(\underbrace{a}_{\text{input parameters (types)}} \rightarrow \underbrace{b}_{\text{output parameters (types)}})$$

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

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

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.)

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

$\varnothing$ - type clash (zero effect)

$(a \to b)\cdot(c \to d)$ - 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 occurrences of u^j in all type lists of sequence y are replaced by t^k, where k is unique index over y

Rules

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

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

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

Rules (cont.)

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

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

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.

Tulemused

- Käesolevas ettekandes tutvustati raamistikku, mis lubab analüüsida magasinoperatsioonide jada ning arvutada operatsioonide tegelikku käitumist konkreetses kontekstis.
- Defineeriti formaalsed reeglid magasinefektide arvutamiseks, mis toetavad nn. jokkertüüpe ning tüüpide hierarhiat, samuti polümorfseid operatsioone.