Programmide teisendamine

Osaline väärtustamine

Jaak Pruulmann

jjpp@meso.ee

Mis värk on?

- Osaline väärtustamine e. spetsialiseerimine
  $f(x, y) = x^y, y = 5 \Rightarrow f_5(x) = x^5$
- Rakendatakse programmitekstidele
- Eesmärk: automaatselt rakendatavus
- Eesmärk: programmide genereerimine
- Kleene'i s-m-n teoreem

Näide: astendamine

- Algne programm p =

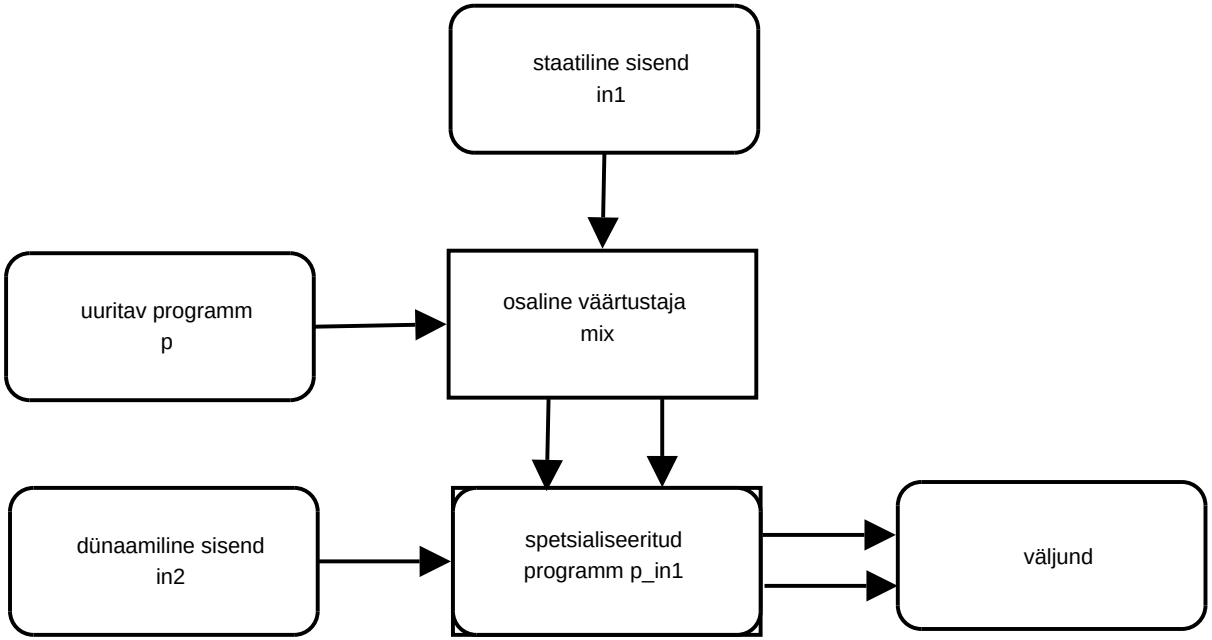
```
f(n, x) = if n = 0 then 1
else if even(n) then f(n/2, x) ^ 2
else x * f(n-1, x)
```

- Programm p pärast spetsialiseerimist staatilise sisendiga $n = 5$, p_5 =

```
f5(x) = x * ((x ^ 2) ^ 2)
```

Skeem

- Olgu meil programm $p(in1, in2)$.



Kontrolliks: $[[p]][in1, in2] = [[p_{in1}]]in2$

Tähistusi

- Olgu D kõigi väärtuste (sh programmitekstid) hulk.
- Kasutame keeli L (implementeerimiskeel), S (lähtetekstide keel) ja T (sihtkeel).
- Programmi tähendus $[[\_]]_L$ on funktsioon $D^* \rightarrow D$.
- Olgu meil programm p keeles L, siis $[[p]]_L$ on selle tähendus.
- $out = [[p]]_L[in_1, in_2, ..., in_n]$ tagastab p väljundi sisenditel in_1 kuni in_n või on defineerimata, kui p ei lõpeta.

Veel tähistusi

Kui in_1 on programmi kõik staatilised sisendid ja in_2 programmi kõik dünaamilised sisendid, siis

- $out = [[p]][in_1, in_2]$ on üheastmeline arvutus ja
- $p_{in1} = [[mix]][p, in_1], out = [[p_{in1}]]in_2$ on kaheastmeline arvutus, millest saame mix'i definitsiooni:
- $[[p]][in_1, in_2] = [[[[mix]][p, in_1]]]in2$ (kui üks pool on defineeritud, on ka teine defineeritud ning omab sama väärtust)

Milleks meile osaline väärtustamine?

- Harva muutuvad andmed
- Sageli: $t_{mix}(p, in1) + t_{p_{in1}}(in2) < t_p(in1, in2)$
- Efektiivsus vs üldisus
- Muud keerulised ülesanded: raytracing, andmebaasipäringud, teadusarvutused, pattern-matching

Kuidas me seda teeme?

- Automaatselt: *sümbolarvutus, avamine, programmipunktide spetsialiseerimine*
- Offline:
  - (0) annoteeri programmi tekst (*binding-time analysis*; märgitakse dynaamilist sisendit kasutavad osad)
  - (1) arvuta kõik märkimata (staatilised) avaldised
  - (2) ava kõik märkimata funktsiooniväljakutsed
  - (3) genereeri jääkkood (*residual code*) kõigile märgitud avaldistele
  - (4) genereeri jääkväljakutsed kõigile märgitud funktsioonide väljakutsetele

Kuidas me seda teeme?

- Online:
  - arvutab asju esimesel võimalusel, kasutades ainult ja kogu kättesaadavat infot
  - on üldjuhul parem, eriti struktuursete, osaliselt staatiliste andmete korral
  - väärtustaja peatumine pole päris garanteeritud jms probleemid

binding-time analysis

- annotatsioonide aluseks on teadmine selle kohta, millised sisendid on programmi spetsialiseerimisel teada.
- operatsioonid märgitakse kas eemaldatavaks või jäävaks (*eliminable, residual*)
- nõutakse, et eemaldatuks märgitud operatsioon oleks eemaldatav suvalise staatilise sisendi korral (kongruentsus, automaatselt saavutatav)
- sõltumata sisendist ei tohi spetsialiseerija saada produtseerida lõpmatult paljusid funktsioone ega lõpmatult suuri avaldisi (peatumine)

Kompilaatorid, interpretaatorid 1

- Enamus arvutusi on tehtavad kas ühe või mitme sammuga, kasutades programmide genereerimist.
- Levinud näited mitme sammuga arvutamisest on kompilaatorid ja parserigeneraatorid.
- Võrdlus: interpretaatorid on väiksemad, lihtsamad, aeglasemad.

Kompilaatorid, interpretaatorid 2

- Interpretaator:
  $out = [[source]]_S[in_1, in_2, ..., in_n]$
  $= [[int]]_L[source, in_1, in_2, ..., in_n]$
- S'i interpretaator keeles L:
  $[[source]]_S d = [[int]]_L[source, d]$
- Kompilaator:
  $target = [[compiler]]_L source$
  $output = [[source]]_S[in_1, in_2, ..., in_n]$
  $= [[target]]_T[in_1, in_2, ..., in_n]$
- S'ist T'sse kompileeriv kompilaator keeles L:
  $[[source]]_S d = [[\ [[compiler]]_L source]]_T d$

Kompilaatorid, interpretaatorid 3

- Spetsialiseeritud interpretaator = kompilaator: $target = [[mix]][int, source]$
- mix ei ole kompilaator!
- Kaotame interpreteerimis-*overhead*'i
- Väljundprogramm on alati korrektne (interpretaatori suhtes)
- Sihtkeel ei tarvitse kõige mõistlikum olla?
- Uusi andmetüüpe vms ei leiutata, teisendused on üldised ja automaatsed – käsitsi kodeeritud kompilaator on võibolla efektiivsem.

Interpreteerimiskulud

- üldiselt: $a_p * t_p(d) \leq t_{int}(p, d)$, kus a_p ei sõltu sisendist d kuid võib sõltuda programmist p.
- eksperimentaalselt kipub a_p olema 10
- $sint$ – self-interpreter:
  $[[p]]d = [[sint]][p, d] = [[\ [[mix]][sint, p]\ ]]d$
  $p' = [[mix]][sint, p] = p$

Generaatorite generaatorid

- Kui spetsialiseerija on sobivas keeles, saame ka seda spetsialiseerida – kiiremaks muuta.
- Praktikas ei kasutata üldiseid programme – spetsifikatsioonide täitjaid, sest need oleks liiga aeglased
- Samas on spetsifikatsioonid loetavamad ja kergemini muudetavad

Futamura projektsioonid

1. kompilaator:
   $out = [[source]]_S input = [[int]][source, input]$
   $= [[\ [[mix]]\ [int, source]\ ]] input = [[target]] input$
2. kompilaatorigeneraator:
   $compiler = [[mix]][mix, int]$
   $target = [[mix]][int, source] =$
   $[[\ [[mix]][mix, int]\ ]] source$
   $= [[compiler]] source$

Futamura projektsioonid

3. kompilaatorigeneraatori generaator:

$$cogen = [[mix]][mix, mix]$$
$$[[p]][in1, in2] =$$
$$[[\ [[mix]][p, in1]]]in2 =$$
$$[[\ [[\ [[mix]][mix, p]]]in1]]in2 =$$
$$[[\ [[\ [[\ [[mix]][mix, mix]]]p]]in1]]in2 =$$
$$[[\ [[\ [[cogen]]p]]in1]]in2$$

Automaatne genereerimine

$compiler = [[cogen]]int$ – L'is kirjutatud L'st L'i teisendaja.

- instrumenteerimine (debug-laused vms)
- CPS
- laisast virgaks
- sabarekursiooni kasutavaks

Veel

- semantikapõhised kompilaatorid
- osaline väärtustamine ei ravi kõiki hädasid
- üldised programmid
- pole nii automaatne, kui võiks olla