Koodi optimaalne korraldus

- korduvate koodiosade kokkutõstmine (*common subexpression evaluation/elimination = CSE*)

- tsüklite lahtikerimine (*loop unrolling*)

- sabarekursiooni optimeerimine (*tail call optimization*)

Koodi optimaalne korraldus

- iteratsiooni teisendamine sabarekursiooniks akumulaatorparameetrite lisamise teel (*accumulator introduction*)

- agaruse analüüsimine (*strictness analysis*)

Optimaalne keskkond

- kastide eemaldamine (*boxing analysis*, *unboxing*)
- viitade märgistamine (*pointer tagging*)
- ettearvutamine (*precomputation*)

Sihtarhitektuuri spetsiifika

- superkompileerimine (*supercompilation*)

- piiluaukude täitmine (*peephole optimization*)

Korduvate koodiosade kokkutõstmine

- aitab iseäranis aritmeetikat kiirendada
- AST'ga opereerides lihtsalt saavutatav
- parimate tagajärgede saavutamiseks võib kombineerida lihtalgebraliste teisendustega
- kõrvalefektiga keeltes ei tohi 'enne' ja 'pärast' muutuja-avaldisi kokku tõsta

Korduvate koodiosade kokkutõstmine

```
a := b * c + g;
d := b * c * d;
e := b * c;
```

Kõik kolm paremat poolt sisaldavad sama avaldist `b * c`. Selle võime algul välja arvutada ja seejärel korduvalt kasutada.

Korduvate koodiosade kokkutõstmine

Enne optimeerimist:

```
a := b * c + g;
d := b * c * d;
e := b * c;
```

Pärast optimeerimist:

```
e := b * c;
a := e + g;
d := e * d;
```

Nii hoiti kaks korrutustehet kokku.

Tsüklite lahtikerimine

- vähendab tsüklilõpusiirde *overhead*'i
- suurendab koodi
- iseäranis kasulik lihtsate vektoroperatsioonide kiirendamiseks
- optimaalne lahtikerimise maht sõltub sihtarhitektuurist ja ei ole enamasti ilmne

Tsüklite lahtikerimine

Enne optimeerimist:

```
for (i := 1; i <= n; i++)
    sum := sum + a[i];
```

Pärast optimeerimist:

```
for (i := 1; i <= n - 3; i += 4) {
    sum := sum + a[i];
    sum := sum + a[i + 1];
    sum := sum + a[i + 2];
    sum := sum + a[i + 3];
}
for (; i <= n; i++)
    sum := sum + a[i];
```

Sabarekursiooni optimeerimine

- ei ole üldjuhul hädavajalik — mida programmeerija saaks kirjutada sabarekursiivselt, saaks ta kirjutada ka iteratiivselt

- sabarekursiooni toetamine on siiski kasulik, kuna lubab programmeerijal kirjutada lihtsamini mõistetavat, rekursiivset koodi

- eeldab, et stäkifreimi minemaviskamiseks ei ole tarvis väljakutsuja täiendavat abi

Akumulaatorparameetrite lisamine

Paljudel juhtudel ei ole täidetud sabarekursiooni optimeerimiseks vajalik eeldus — pärast väljakutse toimumist on vajalik väljakutsuva protseduuri täiendav töö. Niisugustest olukordadest lahtisaamiseks võib mõnikord kompilaator lisada spetsiaalseid programmeerijale nähtamatuid akumulaatorparameetreid.

Akumulaatorparameetrite lisamine

Enne optimeerimist:

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (fac (n - 1)))))
```

Pärast optimeerimist:

```
(define (factorial n)
  (let sub ((n n) (acc 1))
    (if (= n 0)
      acc
      (sub (n - 1) (* acc (n - 1))))))
```

Parameetrikasutuse agaruse analüüsimine

- kasulik üksnes laisa väärtustamisega keelte korral
- rekursiivsete funktsioonide puhul ka analüüs rekursiivne
- kõrgemat järku funktsioonide puhul analüüs komplitseeritud

Kastide vajalikkuse analüüsimine

Kui translaator teab, et väärtus, mille ta kasti paneb, võetakse sealt kohe jälle välja, võib kasti loomise ära jätta. See eeldab võimalust nii kasti saaja kui saatja koodi muuta.

- hoiab kokku nii aega kui kuhjakasutust
- baasoperaatorite puhul saab konteksti sageli staatiliselt tuvastada
- võib raskendada silumisriistade tööd
- raske kasutada tõhusalt koos teekidega

Viitade märgistamine

- kastide eemaldamise üks vorme
- võimaldab kaste eemaldada struktuurtüüpidelt
- kiirendab sageli tüübikontrolli
- palju pisikesi ühesuguseid struktuure loovates keeltes (Lispi dialektid, struktuuritöötlemiseks kasutatav Prolog) praktiliseks kasutamiseks peaaegu hädavajalik

Viitade märgistamine

- kui viidamärgistust kasutatakse kogu koodis ühtmoodi objektides leiduvate tüübilahtrite asendajana, leevendab silumisriistade tööd raskendavaid asjaolusid

- baastüüpide (millel joonduspiiranguid ei ole) märgistamisel kulutab ära kasulikke bitte

- raskesti kasutatav sõnakaupa adresseeritavatel arhitektuuridel

Ettearvutamine

Avaldiste, mis sõltuvad väikesest arvust väikese muutumispiirkonnaga parameetritest, puhul võib translaator avaldiste kõikvõimalikud väärtused ette välja arvutada ning avaldise massiivipöördumiseks transleerida.

- kasulik paljude automaatide transleerimisel
- enamasti vähendab koodihulka
- suurendab andmehulka
- konkreetne tulu sõltub palju sihtarhitektuurist

Superkompileerimine

- leiab sageli parima võimaliku tulemuse
- väga ajamahukas kompileerimisaja suhtes
- enamasti tehakse väikeste tükikeste kaupa kompilaatori enda realiseerimise ajal

Näiteks `i386` peal saab täisarvude tavalise suuruse ja esituse korral absoluutväärtust arvutada niimoodi (sisend `EAX`, väljund `EDX`):

```
cdq
neg EAX
adc EDX, EDX
```

Piiluaukude täitmine

- vajalik peamiselt RISC-arhitektuuride korral
- enamasti realiseeritav lihtsa mustriasenduse teel juba genereeritud koodis