Metody Realizacji Języków Programowania

Generacja kodu pośredniego

Marcin Benke

MIM UW

13 grudnia 2010

Kod pośredni?

- Nie jest niezbędny — zwłaszcza przy generacji kodu na maszynę stosową.
- Przy generacji kodu na różne architektury wspólne transformacje niezależne od architektury.
- Ułatwia niektóre optymalizacje.
- Różne postaci — tu zajmiemy się najbardziej popularną: kodem czwórkowym.

Kod czwórkowy

Czwórka:

$$w := a_1 \oplus a_2$$

- argumenty a_1, a_2
- operacja $\oplus$
- lokalizacja wyniku w

Zwany również kodem trójadresowym, większość instrukcji zawiera bowiem trzy adresy: wyniku i dwu argumentów.

Instrukcje

- Przypisanie postaci x := y op z, gdzie op to jeden z operatorów +,-,*,/,and,or,xor.
- Przypisanie jednoargumentowe postaci x := op y, gdzie op to -, not.
- Kopiowanie postaci x := y.
- Skoki bezwarunkowe postaci goto L, gdzie L to adres w kodzie zazwyczaj reprezentowany przez etykietę;
- przed każdą instrukcją może wystąpić etykieta odnosząca się do pierwszego adresu występującego po niej.
- Skoki warunkowe postaci if x oprel y goto L, gdzie oprel to operator relacyjny (<,>,=,<=,>=, !=). Jeśli warunek jest spełniony, to skok jest wykonywany.

Instrukcje c.d.

- Wywoływanie funkcji, obsługiwane przez dwa rozkazy: `param x` i `t := call L, n`, służące odpowiednio do przekazania x jako kolejnego parametru funkcji oraz wywołania funkcji pod adresem L z n parametrami.
- Powrót z funkcji dokonywany jest przez rozkaz `return x`, gdzie x to wartość zwracana.
- Przypisania indeksowane postaci `x := y[i]` oraz `x[i] := y`. Pierwszy z tych rozkazów powoduje umieszczenie pod adresem `x` zawartości pamięci spod adresu `y + i`, drugi - umieszczenie pod adresem `x + i` wartości spod adresu `y`.

Generacja kodu dla wyrażeń arytmetycznych

$$e_1 \; op \; e_2$$

Przyjmijmy konwencję, że procedura `generuj` tworząca kod dla podwyrażenia oprócz samego kodu zwraca nazwę zmiennej, na której zapamiętany jest wynik. Wówczas nasza procedura mogłaby wyglądać następująco:

```
zmienna1, kod1 = generuj(e1)
zmienna2, kod2 = generuj(e2)
wynik = nowa_tymczasowa()
kod = kod1 + kod2 + wynik + " := "
    + zmienna1 + " op " + zmienna2
return wynik, kod
```

Przykład

Dla wyrażenia

$$(a * a - c) + b * b + d$$

wygenerujemy kod

```
t1 := a*a
t2 := t1 - c  // t2 = a*a - c
t3 := b*b
t4 := t2 + t3 // t4 = a*a - c + b*b
t5 := t4 + d
```

Oszczędzanie zmiennych tymczasowych

Gdy zmienna tymczasowa została już wykorzystana jako operand, możemy jej użyć ponownie. Możemy zatem nasz przykład przepisać następująco:

```
t1 := a*a
t1 := t1 - c  // t1 = a*a - c
t2 := b*b
t1 := t1 + t2 // t1 = a*a - c + b*b
t1 := t1 + d
```

Używając tylko 2 zmiennych tymczasowych zamiast 5.

Kolejność obliczeń

- Rozważmy wyrażenie $e_1 + e_2$, przy czym obliczenie e_1 wymaga k zmiennych tymczasowych (rejestrów), zaś e_2 — n zmiennych (załóżmy, że $n > k$).
- Jeśli możemy obliczać e_1 i e_2 w dowolnym porządku, to które lepiej obliczyć najpierw, aby zużyć jak najmniej zmiennych tymczasowych?

Kolejność obliczeń

- Rozważmy wyrażenie $e_1 + e_2$, przy czym obliczenie e_1 wymaga k zmiennych tymczasowych (rejestrów), zaś e_2 — n zmiennych (załóżmy, że $n > k$).
- Jeśli możemy obliczać e_1 i e_2 w dowolnym porządku, to które lepiej obliczyć najpierw, aby zużyć jak najmniej zmiennych tymczasowych?
  1. "najpierw łatwiejsze": k rejestrów dla obliczenia e_1, potem 1 przechowujący jego wartość plus n dla obliczenia e_2

$$\max(k, 1 + n) = 1 + n$$

Kolejność obliczeń

- Rozważmy wyrażenie $e_1 + e_2$, przy czym obliczenie e_1 wymaga k zmiennych tymczasowych (rejestrów), zaś e_2 — n zmiennych (załóżmy, że $n > k$).
- Jeśli możemy obliczać e_1 i e_2 w dowolnym porządku, to które lepiej obliczyć najpierw, aby zużyć jak najmniej zmiennych tymczasowych?
  1. "najpierw łatwiejsze": k rejestrów dla obliczenia e_1, potem 1 przechowujący jego wartość plus n dla obliczenia e_2

$$\max(k, 1 + n) = 1 + n$$

  2. "najpierw trudniejsze"

$$\max(n, 1 + k) = n$$

- Kolejność wyliczenia może zadecydować, czy uda nam się obliczyć wyrażenie tylko przy użyciu rejestrów (bez odsyłania wyników pośrednich do pamięci).

Wywołanie funkcji

$$f(e_1, \ldots, e_n)$$

```
zmienna_1, kod1 = generuj(e1)
...
zmienna_n, kod_n = generuj(en)
param en
wynik = nowa_tymczasowa()
kod = kod_1 + ...  + kod_n
    + "param " + zmienna_1
    + ...
    + "param " + zmienna_n
    + wynik + " := call " + f + "," + n
return wynik, kod
```

Przypisanie

$x := e$

```
zmienna, kod1 = generuj(e)
kod = kod1 + x + " := " + zmienna
return kod
```

Jeśli przypisanie jest wyrażeniem (jak w C, Javie) to

```
zmienna, kod1 = generuj(e)
kod = kod1 + x + " := " + zmienna
return kod, zmienna
```

Pętla while

while(*warunek*) instrukcja

Można wygenerować kod następujący:

```
L1: kod warunku, wynik w t
    if not t goto L2
    kod instrukcji
    goto L1
L2: ...
```

Można też trochę inaczej:

```
    goto L2
L1: kod instrukcji
L2: kod warunku, wynik w t
    if t goto L1
```

W pierwszym wariancie na n obrotów petli wykonujemy $2n$ skoków, w drugim — $n + 1$.

Instrukcja warunkowa

if(*warunek*) instrukcja$_1$ **else** instrukcja$_2$

Można wygenerować kod następujący:

```
        kod warunku, wynik w t
        if not t goto Lfalse
Ltrue:  kod instrukcji1
        goto Lend
Lfalse: kod instrukcji2
Lend:   ...
```

lub

```
        kod warunku, wynik w t
        if t goto Ltrue
Lfalse: kod instrukcji2
        goto Lend
Ltrue: kod instrukcji1
Lend:   ...
```

Skrócone tłumaczenie wyrażeń logicznych

Wyrażenia logiczne można tłumaczyć albo tak jak wyrażenia arytmetyczne, albo przy użyciu tzw. *kodu skaczącego*

$$w1 \&\& w2$$

```
if not w1 goto Lfalse
if not w2 goto Lfalse
kod Ltrue lub goto Ltrue
```

$$w1 || w2$$

```
if w1 goto Ltrue
if w2 goto Ltrue
kod Lfalse lub goto Lfalse
```

Przykład

```
if (i>=0) && (i<n) then I1 else I2
```

Można przetłumaczyć jako

```
        if i<0 goto Lfalse
        if i>n goto Lfalse
Ltrue:  I1
        goto Lend
Lfalse: I2
Lend:   ...
```

Generacja kodu na maszynę stosową

Kod dla instrukcji — jak dla kodu czwórkowego

Kod dla wyrażeń:

$$e_1 \; op \; e_2$$

```
kod e1
kod e2
op
```

$$f(e_1, \ldots, e_n)$$

```
kod e1
kod en
call f
```

(zakładając, że instrukcja `call` — jak w JVM — pobiera argumenty ze stosu).

Interpreter kodu czwórkowego: `iquadr`

Dostepny jest edukacyjny interpreter kodu czwórkowego `iquadr`
Implementowany dialekt:

- nieograniczona ilość rejestrów typu int i double, np. `i0`, `d1`
- zmienne lokalne i tymczasowe:
  `t2 := a + t1` pisane jako `$.i2 := a.i3 + $.i1`
  (identyfikator przed kropką ma charakter li tylko komentarza)
- brak zmiennych globalnych (tylko funkcje)
- konwersje między typami np. `$.i1 := b.d0`
- dodatkowa instrukcja `print`
- nie ma wbudowanego mechanizmu przekazywania parametrów
- dostęp do pamięci; adresowanie postaci `{rejestr+stała}`, np.
  `{sp.i1-1} := 2`,

Przykład

```
function main : int :
    $.d3  := 1.0
    lo.i0  := $.d3
    hi.i1  := lo.i0
    mx.i3  := 5000000
    mx.i2  := $.i3
    print lo.i0
L0: if hi.i1 >= mx.i2 goto L1
    print hi.i1
    $.i3  := lo.i0 + hi.i1
    hi.i1  := $.i3
    $.i3  := hi.i1 - lo.i0
    lo.i0  := $.i3
    goto L0
L1:
function end
```

iquadr — wywoływanie funkcji

iquadr nie ma osobnych instrukcji dla przekazywania parametrów.
Można je przekazywać w rejestrach (uwaga na rekurencję!)
Result in `i0` or `d0`

```
function main : int :
  $.i0 := -40
  $.i0 := ~$.i0
  $.d0 := 3.14159
  call add
  print $.i0
function end

function add : int -> double -> int :
  $.i1 := b.d0
  result.i0 := a.i0 + $.i1
function end
```

iquadr — wywoływanie funkcji

Można też zaprojektować swój własny protokół wywoływania funkcji, np. używając stosu:

```
function main : int :
  sp.i1 := 512
  bp.i2 := sp.i1
  {sp.i1-1} := 2
  {sp.i1-2} := 3
  sp.i1 := sp.i1 - 2
  call add
  print retval.i0
function end
```

iquadr — wywoływanie funkcji

```
function add : int -> int -> int :
  sp.i1 := sp.i1 - 1
  {sp.i1+0} := bp.i2
  bp.i2 := sp.i1
  a.i3 := {bp.i2+1}
  b.i4 := {bp.i2+2}
  retval.i0 := a.i3 + b.i4
function end
```