Script generated by TTT

Title: Simon: Compilerbau (27.06.2012)

Date: Wed Jun 27 14:07:29 CEST 2012

Duration: 80:22 min

Pages: 52

Beispiel: Übersetzungschema für Ausdrücke

Sei folgende Funktion gegeben:

```
void f(void) {
  int x,y,z;
  x = y+z*3;
}
```

- Sei Adressumgebung $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ gegeben.
- Sei R_4 das erste freie Register, d.h. $i = 4$.

$$\text{code}^4\ \texttt{x=y+z*3}\ \rho = \text{code}^4_R\ \texttt{y+z*3}\ \rho$$
$$\text{move}\ R_1\ R_4$$

Beispiel: Übersetzungschema für Ausdrücke

Sei folgende Funktion gegeben:

```
void f(void) {
  int x,y,z;
  x = y+z*3;
}
```

- Sei Adressumgebung $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ gegeben.
- Sei R_4 das erste freie Register, d.h. $i = 4$.

$$\text{code}^4\ \texttt{x=y+z*3}\ \rho = \text{code}^4_R\ \texttt{y+z*3}\ \rho$$
$$\text{move}\ R_1\ R_4$$
$$\text{code}^4_R\ \texttt{y+z*3}\ \rho = \text{move}\ R_4\ R_2$$
$$\text{code}^5_R\ \texttt{z*3}\ \rho$$
$$\text{add}\ R_4\ R_4\ R_5$$

Beispiel: Übersetzungschema für Ausdrücke

Sei folgende Funktion gegeben:

```
void f(void) {
  int x,y,z;
  x = y+z*3;
}
```

- Sei Adressumgebung $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ gegeben.
- Sei R_4 das erste freie Register, d.h. $i = 4$.

$$\text{code}^4\ \texttt{x=y+z*3}\ \rho = \text{code}^4_R\ \texttt{y+z*3}\ \rho$$
$$\text{move}\ R_1\ R_4$$
$$\text{code}^4_R\ \texttt{y+z*3}\ \rho = \text{move}\ R_4\ R_2$$
$$\text{code}^5_R\ \texttt{z*3}\ \rho$$
$$\text{add}\ R_4\ R_4\ R_5$$
$$\text{code}^5_R\ \texttt{z*3}\ \rho = \text{move}\ R_5\ R_3$$
$$\text{code}^6_R\ 3\ \rho$$
$$\text{mul}\ R_5\ R_5\ R_6$$
$$\text{code}^6_R\ 3\ \rho = \text{loadc}\ R_6\ 3$$

Beispiel: Übersetzungschema für Ausdrücke

Sei folgende Funktion gegeben:

```
void f(void) {
  int x,y,z;
  x = y+z*3;
}
```

- Sei Adressumgebung $\rho = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ gegeben.
- Sei R_4 das erste freie Register, d.h. $i = 4$.

$$
\begin{aligned}
\text{code}^4\ \texttt{x=y+z*3}\ \rho &= \text{code}^4_R\ \texttt{y+z*3}\ \rho \\
&\quad \text{move}\ R_1\ R_4 \\
\text{code}^4_R\ \texttt{y+z*3}\ \rho &= \text{move}\ R_4\ R_2 \\
&\quad \text{code}^5_R\ \texttt{z*3}\ \rho \\
&\quad \text{add}\ R_4\ R_4\ R_5 \\
\text{code}^5_R\ \texttt{z*3}\ \rho &= \text{move}\ R_5\ R_3 \\
&\quad \text{code}^6_R\ \texttt{3}\ \rho \\
&\quad \text{mul}\ R_5\ R_5\ R_6 \\
\text{code}^6_R\ \texttt{3}\ \rho &= \text{loadc}\ R_6\ 3
\end{aligned}
$$

$\rightsquigarrow$ die Zuweisung `x=y+z*3` wird übersetzt als

move R_4 R_2; move R_5 R_3; loadc R_6 3; mul R_5 R_5 R_6; add R_4 R_4 R_5; move R_1 R_4

Kapitel 4:
Anweisungen und Kontrollstrukturen

Über Anweisungen und Ausdrücke

Allgemeine Übersetzungsidee:

- $\text{code}^i\ s\ \rho$: geniere Kode für Anweisung s
- $\text{code}^i_R\ e\ \rho$: generiere Kode für Ausdruck e in R_i

Es gilt: $i, i+1, \ldots$ sind unbenutzte Register

Über Anweisungen und Ausdrücke

Allgemeine Übersetzungsidee:

- $\text{code}^i\ s\ \rho$: geniere Kode für Anweisung s
- $\text{code}^i_R\ e\ \rho$: generiere Kode für Ausdruck e in R_i

Es gilt: $i, i+1, \ldots$ sind unbenutzte Register

Für den *Ausdruck* $x = e$ mit $\rho\ x = a$ hatten wir definiert:

$$
\begin{aligned}
\text{code}^i_R\ x = e\ \rho &= \text{code}^i_R\ e\ \rho \\
&\quad \text{move}\ R_a\ R_i
\end{aligned}
$$

Allerdings ist $x = e$ auch eine *Anweisung*:

Über Anweisungen und Ausdrücke

Allgemeine Übersetzungsidee:

$\text{code}^i\ s\ \rho$: generiere Kode für Anweisung s

$\text{code}^i_R\ e\ \rho$: generiere Kode für Ausdruck e in R_i

Es gilt: $i, i+1, \ldots$ sind unbenutzte Register

Für den *Ausdruck* $x = e$ mit $\rho\ x = a$ hatten wir definiert:

$$\begin{aligned}\text{code}^i_R\ x = e\ \rho \quad &= \quad \text{code}^i_R\ e\ \rho \\ &\qquad\ \ \text{move}\ R_a\ R_i\end{aligned}$$

Allerdings ist $x = e$ auch eine *Anweisung*:

- Definiere:

$$\text{code}^i\ e_1 = e_2\ \rho \quad = \quad \text{code}^i_R\ e_1 = e_2\ \rho$$

Das temporäre Register R_i wird dabei ignoriert. Allgemeiner:

$$\text{code}^i\ e\ \rho = \text{code}^i_R\ e\ \rho$$

Über Anweisungen und Ausdrücke

Allgemeine Übersetzungsidee:

$\text{code}^i\ s\ \rho$: generiere Kode für Anweisung s

$\text{code}^i_R\ e\ \rho$: generiere Kode für Ausdruck e in R_i

Es gilt: $i, i+1, \ldots$ sind unbenutzte Register

Für den *Ausdruck* $x = e$ mit $\rho\ x = a$ hatten wir definiert:

$$\begin{aligned}\text{code}^i_R\ x = e\ \rho \quad &= \quad \text{code}^i_R\ e\ \rho \\ &\qquad\ \ \text{move}\ R_a\ R_i\end{aligned}$$

Allerdings ist $x = e$ auch eine *Anweisung*:

- Definiere:

$$\text{code}^i\ e_1 = e_2\ \rho \quad = \quad \text{code}^i_R\ e_1 = e_2\ \rho$$

Das temporäre Register R_i wird dabei ignoriert. Allgemeiner:

$$\text{code}^i\ e\ \rho = \text{code}^i_R\ e\ \rho$$

- Beobachtung: Die Zuweisung an e_1 ist ein Seiteneffekt beim Auswerten des Ausdrucks $e_1 = e_2$.

Übersetzung von Anweisungsfolgen

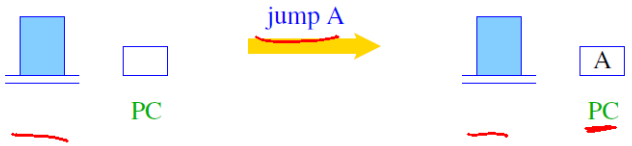
Der Code für eine Anweisungsfolge ist die Konkatenation des Codes for die einzelnen Anweisungen in der Folge:

$$\begin{aligned}\text{code}^i\ (s\ ss)\ \rho \quad &= \quad \text{code}^i\ s\ \rho \\ &\qquad\ \ \text{code}^i\ ss\ \rho \\ \text{code}^i\ \varepsilon\ \rho \quad &= \quad \text{//}\quad \textit{leere Folge von Befehlen}\end{aligned}$$

Beachte: s ist eine Anweisung, ss ist eine Folge von Anweisungen

Sprünge

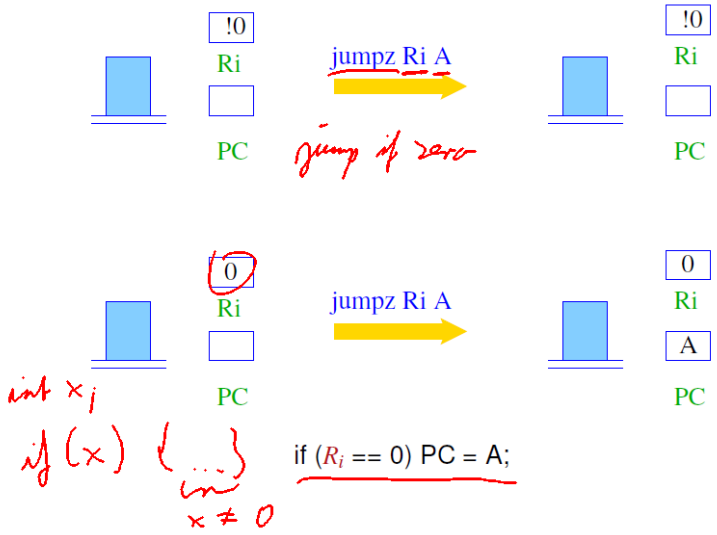
Um von linearer Ausführungsreihenfolge abzuweichen, benötigen wir Sprünge:



PC = A;

Bedingte Sprünge

Ein bedingter Sprung verzweigt je nach Wert in R_i:



if (R_i == 0) PC = A;

Verwaltung von Kontrollfluss

Zur Übersetzung von Kontrollfluss Anweisungen müssen Sprünge erzeugt werden.

- bei der Übersetung eines `if (c)` Konstrukts ist nicht klar, an welcher Adresse gesprungen werden muss, wenn `c` falsch ist

Verwaltung von Kontrollfluss

Zur Übersetzung von Kontrollfluss Anweisungen müssen Sprünge erzeugt werden.

- bei der Übersetung eines `if (c)` Konstrukts ist nicht klar, an welcher Adresse gesprungen werden muss, wenn `c` falsch ist
- Instruktionsfolgen können unterschiedlich arrangiert werden
  - minimiere dabei die Anzahl der unbedingten Sprünge
  - minimiere so dass weniger innerhalb Schleifen gesprungen wird
  - ersetze weite Sprünge (far jumps) durch nahe Sprünge (near jumps)

Verwaltung von Kontrollfluss

Zur Übersetzung von Kontrollfluss Anweisungen müssen Sprünge erzeugt werden.

- bei der Übersetung eines `if (c)` Konstrukts ist nicht klar, an welcher Adresse gesprungen werden muss, wenn `c` falsch ist
- Instruktionsfolgen können unterschiedlich arrangiert werden
  - minimiere dabei die Anzahl der unbedingten Sprünge
  - minimiere so dass weniger innerhalb Schleifen gesprungen wird
  - ersetze weite Sprünge (far jumps) durch nahe Sprünge (near jumps)
- organisiere die Instruktionen in Blöcke ohne Sprünge

Verwaltung von Kontrollfluss

Zur Übersetzung von Kontrollfluss Anweisungen müssen Sprünge erzeugt werden.

- bei der Übersetung eines `if (c)` Konstrukts ist nicht klar, an welcher Adresse gesprungen werden muss, wenn c falsch ist
- Instruktionsfolgen können unterschiedlich arrangiert werden
  - minimiere dabei die Anzahl der unbedingten Sprünge
  - minimiere so dass weniger innerhalb Schleifen gesprungen wird
  - ersetze weite Sprünge (far jumps) durch nahe Sprünge (near jumps)
- organisiere die Instruktionen in Blöcke ohne Sprünge

Dazu definieren wir:

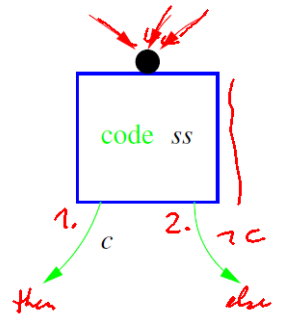
Definition

Ein Basisblock (basic block) besteht aus

- einer Folge von Anweisungen ss, die keine Sprünge enthält
- einer ausgehenden Menge von Kanten zu anderen Basisblöcken
- wobei jede Kante mit einer Bedingung annotiert sein kann

Basisblöcke in der Register C-Maschine

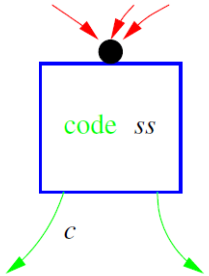
Die R-CMa verfügt nur über einen bedingten Sprung, jumpz.



Die ausgehenden Kanten haben eine spezielle Form:

Basisblöcke in der Register C-Maschine

Die R-CMa verfügt nur über einen bedingten Sprung, jumpz.

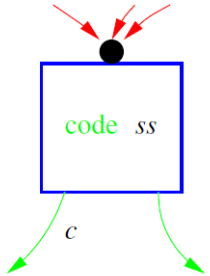


Die ausgehenden Kanten haben eine spezielle Form:

1. eine Kante (unbedingter Sprung), mit jump übersetzt

Basisblöcke in der Register C-Maschine

Die R-CMa verfügt nur über einen bedingten Sprung, jumpz.

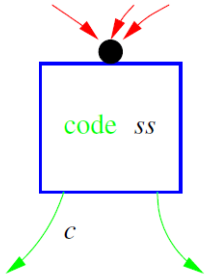


Die ausgehenden Kanten haben eine spezielle Form:

1. eine Kante (unbedingter Sprung), mit jump übersetzt
2. eine Kante mit Bedingung $c = 0$ (jumpz), eine Kante ohne Bedinung (jump)

Basisblöcke in der Register C-Maschine

Die R-CMa verfügt nur über einen bedingten Sprung, jumpz.



Die ausgehenden Kanten haben eine spezielle Form:

1. eine Kante (unbedingter Sprung), mit jump übersetzt
2. eine Kante mit Bedingung $c = 0$ (jumpz), eine Kante ohne Bedinung (jump)
3. eine Liste von Kanten (jumpi) und eine default Kante (jump); wird verwendet für `switch` Anweisungen (kommt später)

Beschreibung von Kontrollfluss Übersetzungen

Der Einfachheit halber verwenden wir *symbolische Sprungziele* zur Beschreibung der Übersetzung von Kontrollflussinstruktionen.

- Ein zweiten Durchlauf ist nötig, um symbolische Adressen durch absolute Code-Adressen zu ersetzen.

Beschreibung von Kontrollfluss Übersetzungen

Der Einfachheit halber verwenden wir *symbolische Sprungziele* zur Beschreibung der Übersetzung von Kontrollflussinstruktionen.

- Ein zweiten Durchlauf ist nötig, um symbolische Adressen durch absolute Code-Adressen zu ersetzen.

Alternativ könnten direkt *relative* Sprünge eingeführt werden:

- relative Sprünge haben Ziele relative zum aktuellen PC
- oft reichen kleinere Adressen aus ($\rightsquigarrow$ near jumps)
- der Code wird relokierbar, d. h. kann im Speicher unverändert hin und her geschoben werden (position independent code, PIC)
- Der erzeugte Code wäre im Prinzip direkt ausführbar.

Beschreibung von Kontrollfluss Übersetzungen

Der Einfachheit halber verwenden wir *symbolische Sprungziele* zur Beschreibung der Übersetzung von Kontrollflussinstruktionen.

- Ein zweiten Durchlauf ist nötig, um symbolische Adressen durch absolute Code-Adressen zu ersetzen.

Alternativ könnten direkt *relative* Sprünge eingeführt werden:

- relative Sprünge haben Ziele relative zum aktuellen PC
- oft reichen kleinere Adressen aus ($\rightsquigarrow$ near jumps)
- der Code wird relokierbar, d. h. kann im Speicher unverändert hin und her geschoben werden (position independent code, PIC)
- Der erzeugte Code wäre im Prinzip direkt ausführbar.

Basisböcke haben den Vorteil, dass sie später zur *Programmoptimierung* weiterverwendet werden können. (Z.B. Reduzierung der Anzahl an Sprüngen.)

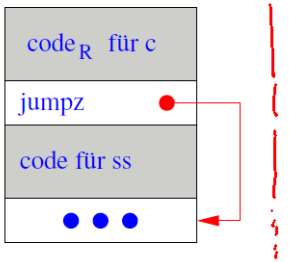
Bedingte Anweisung, einseitig

Betrachten wir zuerst $s \equiv$ **if** (c) ss.
Wir beschreiben die Code-Erzeugung zunächst ohne Basic-Blöcke.

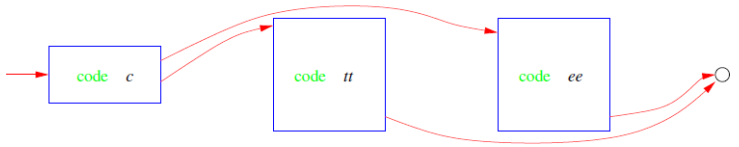
Idee:

- Lege den Code zur Auswertung von c und ss hintereinander in den Code-Speicher;
- Füge Sprung-Befehlen so ein, dass ein korrekter Kontroll-Fluss gewährleistet ist

$$\begin{aligned} \text{code}^i\ s\ \rho \quad = \quad & \text{code}^i_R\ c\ \rho \\ & \text{jumpz}\ R_i\ A \\ & \text{code}^i\ ss\ \rho \\ & A: \quad \ldots \end{aligned}$$

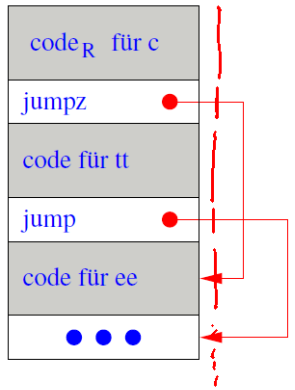


Bedingte Anweisung, zweiseitig



Übersetzung von **if** (c) tt **else** ee.

$$\begin{aligned} \text{code}^i\ \texttt{if}(c)\ tt\ \texttt{else}\ ee\ \rho \quad = \quad & \text{code}^i_R\ c\ \rho \\ & \text{jumpz}\ R_i\ A \\ & \text{code}^i\ tt\ \rho \\ & \text{jump}\ B \\ A: \quad & \text{code}^i\ ee\ \rho \\ B: \quad & \end{aligned}$$



Beispiel für if-Anweisung

Sei $\rho = \{x \mapsto 4, y \mapsto 7\}$ und s die Anweisung:

```
if (x>y) {        /* (i) */
  x = x - y;      /* (ii) */
} else {
  y = y - x;      /* (iii) */
}
```

Dann liefert $\text{code}^i\ s\ \rho$:

Beispiel für if-Anweisung

Sei $\rho = \{x \mapsto 4, y \mapsto 7\}$ und s die Anweisung:

```
if (x>y) {        /* (i) */
  x = x - y;      /* (ii) */
} else {
  y = y - x;      /* (iii) */
}
```

Dann liefert $\text{code}^i\ s\ \rho$:

(i)

```
move Ri R4
move Ri+1 R7
gr Ri Ri Ri+1
jumpz Ri A
```

(ii)

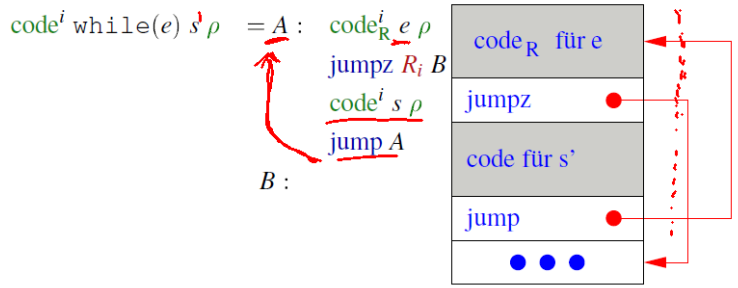
```
move Ri R4
move Ri+1 R7
sub Ri Ri Ri+1
move R4 Ri
jump B
```

(iii)

```
A : move Ri R7
    move Ri+1 R4
    sub Ri Ri Ri+1
    move R7 Ri
B :
```

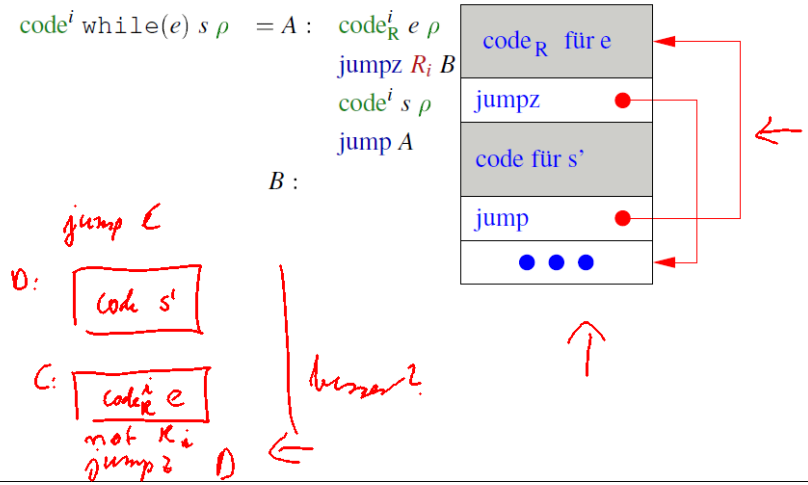
Iterative Anweisungen

Betrachte schließlich die Schleife $s \equiv$ **while** (e) s'. Dafür erzeugen wir:



Iterative Anweisungen

Betrachte schließlich die Schleife $s \equiv$ **while** (e) s'. Dafür erzeugen wir:

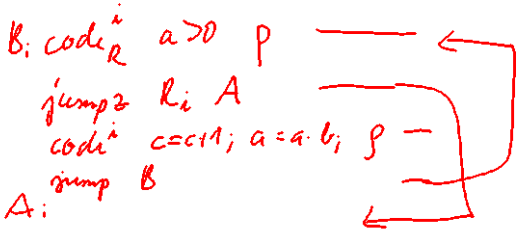


Beispiel: Übersetzung von Schleifen

Sei $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ und s die Anweisung:

```
while (a>0) {    /* (i) */
  c = c + 1;     /* (ii) */
  a = a - b;     /* (iii) */
}
```

Dann liefert $\text{code}^i\ s\ \rho$ die Folge:

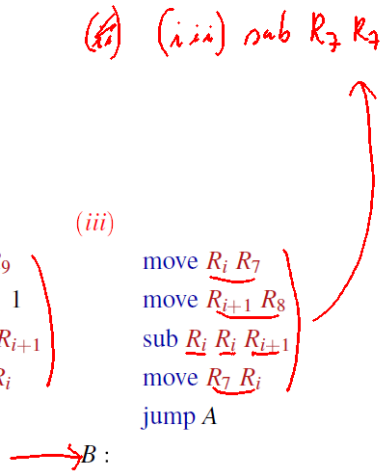


Beispiel: Übersetzung von Schleifen

Sei $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ und s die Anweisung:

```
while (a>0) {    /* (i) */
  c = c + 1;     /* (ii) */
  a = a - b;     /* (iii) */
}
```

Dann liefert $\text{code}^i\ s\ \rho$ die Folge:



for-Schleifen

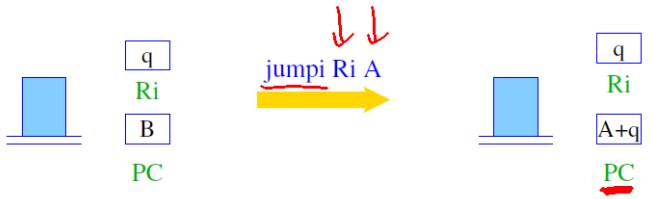
Die **for**-Schleife $s \equiv$ **for** $(e_1; e_2; e_3)\ s'$ ist äquivalent zu der Statementfolge $e_1;$ **while** $(e_2)\ \{s'\ e_3;\}$ – sofern s' keine **continue**-Anweisung enthält.
Darum übersetzen wir:

$$
\begin{array}{rcl}
\text{code}^i\ \texttt{for}(e_1;e_2;e_3)\ s\ \rho & = & \text{code}^i_R\ e_1\ \rho \\
& & A: \quad \text{code}^i_R\ e_2\ \rho \\
& & \quad\ \text{jumpz}\ R_i\ B \\
& & \quad\ \text{code}^i\ s\ \rho \\
& & \quad\ \text{code}^i_R\ e_3\ \rho \\
& & \quad\ \text{jump}\ A \\
& & B:
\end{array}
$$

Das switch-Statement

Idee:

- Unterstütze Mehrfachverzweigung in *konstanter Zeit*
- Benutze *Sprungtabelle*, die an der i-ten Stelle den Sprung an den Anfang der i-tem Alternative enthält.
- Eine Möglichkeit zur Realisierung besteht in der Einführung von *indizierten Sprüngen*.



PC = A + R_i;

Aufeinanderfolgende Alternativen

Sei ein **switch** s mit k aufeinanderfolgenden **case** Fällen gegeben:

```
switch (e) {
  case c_0: s_0; break;
  ⋮
  case c_{k-1}: s_{k-1}; break;
  default: s; break;
}
```

d.h. $c_i + 1 = c_{i+1}$ für $i = [0, k-1]$.

Aufeinanderfolgende Alternativen

Sei ein **switch** s mit k aufeinanderfolgenden **case** Fällen gegeben:

```
switch (e) {
  case c_0: s_0; break;
  ⋮
  case c_{k-1}: s_{k-1}; break;
  default: s; break;
}
```

d.h. $c_i + 1 = c_{i+1}$ für $i = [0, k-1]$. Definiere $\text{code}^i\ s\ \rho$ wie folgt:

$$
\begin{array}{rcl}
\text{code}^i\ s\ \rho & = & \text{code}^i_R\ e\ \rho \\
& & \textit{check}^i\ c_0\ c_{k-1}\ B\ 0 \\
& A_0: & \text{code}^i\ s_0\ \rho \\
& & \text{jump}\ D \\
& \vdots & \vdots \\
& A_{k-1}: & \text{code}^i\ s_{k-1}\ \rho \\
& & \text{jump}\ D
\end{array}
\qquad
\begin{array}{rcl}
& B: & \text{jump}\ A_0 \\
& \vdots & \vdots \\
& & \text{jump}\ A_{k-1} \\
& C: & \text{code}^i\ s \\
& D: &
\end{array}
$$

Aufeinanderfolgende Alternativen

Sei ein **switch** s mit k aufeinanderfolgenden **case** Fällen gegeben:

```
switch (e) {
  case c_0: s_0; break;
  ⋮
  case c_{k-1}: s_{k-1}; break;
  default: s; break;
}
```

d.h. $c_i + 1 = c_{i+1}$ für $i = [0, k-1]$. Definiere $\text{code}^i\ s\ \rho$ wie folgt:

$$
\begin{array}{rcl}
\text{code}^i\ s\ \rho & = & \text{code}^i_R\ e\ \rho \\
 & & \mathit{check}^i\ c_0\ c_{k-1}\ B \\
 & A_0: & \text{code}^i\ s_0\ \rho \\
 & & \text{jump}\ D \\
 & \vdots & \vdots \\
 & A_{k-1}: & \text{code}^i\ s_{k-1}\ \rho \\
 & & \text{jump}\ D
\end{array}
\qquad
\begin{array}{ll}
B: & \text{jump}\ A_0 \\
\vdots & \vdots \\
 & \text{jump}\ A_{k-1} \\
C: &
\end{array}
$$

$\mathit{check}^i\ l\ u\ B$ prüft ob $l \le R_i < u$ gilt und springt entsprechend.

Übersetzung des check^i Makros

Das Makro $\mathit{check}^i\ l\ u\ B$ prüft ob $l \le R_i < u$. Sei $k = u - l$.

- wenn $l \le R_i < u$, springt es nach $B + R_i - l$
- wenn $R_i < l$ oder $R_i \ge u$ springt es nach C

$$
\begin{array}{ll}
B: & \text{jump}\ A_0 \\
\vdots & \vdots \\
 & \text{jump}\ A_{k-1} \\
C: &
\end{array}
$$

Übersetzung des check^i Makros

Das Makro $\mathit{check}^i\ l\ u\ B$ prüft ob $l \le R_i < u$. Sei $k = u - l$.

- wenn $l \le R_i < u$, springt es nach $B + R_i - l$
- wenn $R_i < l$ oder $R_i \ge u$ springt es nach C

Wir definieren:

$$
\begin{array}{rcl}
\mathit{check}^i\ l\ u\ B & = & \text{loadc}\ R_{i+1}\ l \\
 & & \text{geq}\ R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz}\ R_{i+2}\ E \\
 & & \text{sub}\ R_i\ R_i\ R_{i+1} \\
 & & \text{loadc}\ R_{i+1}\ k \\
 & & \text{geq}\ R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz}\ R_{i+2}\ D \\
 & E: & \text{loadc}\ R_i\ k \\
 & D: & \text{jumpi}\ R_i\ B
\end{array}
\qquad
\begin{array}{ll}
B: & \text{jump}\ A_0 \\
\vdots & \vdots \\
 & \text{jump}\ A_{k-1} \\
C: &
\end{array}
$$

Übersetzung des check^i Makros

Das Makro $\mathit{check}^i\ l\ u\ B$ prüft ob $l \le R_i < u$. Sei $k = u - l$.

- wenn $l \le R_i < u$, springt es nach $B + R_i - l$
- wenn $R_i < l$ oder $R_i \ge u$ springt es nach C

Wir definieren:

$$
\begin{array}{rcl}
\mathit{check}^i\ l\ u\ B & = & \text{loadc}\ R_{i+1}\ l \\
 & & \text{geq}\ R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz}\ R_{i+2}\ E \\
 & & \text{sub}\ R_i\ R_i\ R_{i+1} \\
 & & \text{loadc}\ R_{i+1}\ k \\
 & & \text{geq}\ R_{i+2}\ R_i\ R_{i+1} \\
 & & \text{jumpz}\ R_{i+2}\ D \\
 & E: & \text{loadc}\ R_i\ k \\
 & D: & \text{jumpi}\ R_i\ B
\end{array}
\qquad
\begin{array}{ll}
B: & \text{jump}\ A_0 \\
\vdots & \vdots \\
 & \text{jump}\ A_{k-1} \\
C: &
\end{array}
$$

Beachte: ein Sprung $\text{jumpi}\ R_i\ B$ mit $R_i = k$ landet bei C.

Übersetzung der Sprungtabelle

Dies ist nur eine Übersetzung für *spezielle* **`switch`**-Anweisungen.

- Beginnt die Tabelle mit 0 statt mit u, brauchen wir den R-Wert von e nicht vermindern, bevor wir ihn als Index benutzen.
- Sind sämtliche möglichen Werte von e sicher im Intervall $[l, u]$, können wir auf check verzichten.
- Kann die Übersetzung von **`switch`**-Anweisungen als L-Attributierung implementiert werden?

Übersetzung der Sprungtabelle

Dies ist nur eine Übersetzung für *spezielle* **`switch`**-Anweisungen.

- Beginnt die Tabelle mit 0 statt mit u, brauchen wir den R-Wert von e nicht vermindern, bevor wir ihn als Index benutzen.
- Sind sämtliche möglichen Werte von e sicher im Intervall $[l, u]$, können wir auf check verzichten.
- Kann die Übersetzung von **`switch`**-Anweisungen als L-Attributierung implementiert werden?
  - nein, denn Marke B is unbekannt wenn $check^i$ übersetzt wird
  - flexiblere Anordnung wenn Kode-Sequenzen aneinandergehängt werden können
  - im Extrem: Übersetzung mit Basisblöcken

Aufeinanderfolgende Alternativen

Sei ein **`switch`** s mit k aufeinanderfolgenden **`case`** Fällen gegeben:

```
switch (e) {
  case c_0: s_0; break;
  ⋮
  case c_{k-1}: s_{k-1}; break;
  default: s; break;
}
```

d.h. $c_i + 1 = c_{i+1}$ für $i = [0, k-1]$. Definiere $\text{code}^i\ s\ \rho$ wie folgt:

$$
\begin{array}{lll}
\text{code}^i\ s\ \rho & = & \text{code}^i_{\text{R}}\ e\ \rho \\
 & & check^i\ c_0\ c_{k-1}\ B \\
A_0: & & \text{code}^i\ s_0\ \rho \\
 & & \text{jump}\ D \\
\vdots & & \vdots \\
A_{k-1}: & & \text{code}^i\ s_{k-1}\ \rho \\
 & & \text{jump}\ D
\end{array}
\qquad
\begin{array}{ll}
B: & \text{jump}\ A_0 \\
\vdots & \vdots \\
 & \text{jump}\ A_{k-1} \\
C: &
\end{array}
$$

$check^i\ l\ u\ B$ prüft ob $l \le R_i < u$ gilt und springt entsprechend.

Übersetzung der Sprungtabelle

Dies ist nur eine Übersetzung für *spezielle* **`switch`**-Anweisungen.

- Beginnt die Tabelle mit 0 statt mit u, brauchen wir den R-Wert von e nicht vermindern, bevor wir ihn als Index benutzen.
- Sind sämtliche möglichen Werte von e sicher im Intervall $[l, u]$, können wir auf check verzichten.
- Kann die Übersetzung von **`switch`**-Anweisungen als L-Attributierung implementiert werden?
  - nein, denn Marke B is unbekannt wenn $check^i$ übersetzt wird
  - flexiblere Anordnung wenn Kode-Sequenzen aneinandergehängt werden können
  - im Extrem: Übersetzung mit Basisblöcken

Allgemeine Übersetzung der switch-Anweisung

Im Allgemeinen können die Werte der Alternativen weit auseinander liegen.

- generiere eine Folge von expliziten Tests, wie für die `if`-Anweisung

Allgemeine Übersetzung der switch-Anweisung

Im Allgemeinen können die Werte der Alternativen weit auseinander liegen.

- generiere eine Folge von expliziten Tests, wie für die `if`-Anweisung
- bei n verschiedenen Tests kann binäre Suche angewendet werden $\rightsquigarrow O(\log n)$ Tests

Allgemeine Übersetzung der switch-Anweisung

Im Allgemeinen können die Werte der Alternativen weit auseinander liegen.

- generiere eine Folge von expliziten Tests, wie für die `if`-Anweisung
- bei n verschiedenen Tests kann binäre Suche angewendet werden $\rightsquigarrow O(\log n)$ Tests
- hat die Menge der getesteten Werte kleine Lücken (≤ 3), so ist Sprungtabelle effizienter und platzsparender

Allgemeine Übersetzung der switch-Anweisung

Im Allgemeinen können die Werte der Alternativen weit auseinander liegen.

- generiere eine Folge von expliziten Tests, wie für die `if`-Anweisung
- bei n verschiedenen Tests kann binäre Suche angewendet werden $\rightsquigarrow O(\log n)$ Tests
- hat die Menge der getesteten Werte kleine Lücken (≤ 3), so ist Sprungtabelle effizienter und platzsparender
- eventuell kann man mehrere Sprungtabellen für verschiedene, zusammenhängende Mengen generieren

Allgemeine Übersetzung der switch-Anweisung

Im Allgemeinen können die Werte der Alternativen weit auseinander liegen.

- generiere eine Folge von expliziten Tests, wie für die **`if`**-Anweisung
- bei n verschiedenen Tests kann binäre Suche angewendet werden $\leadsto O(\log n)$ Tests
- hat die Menge der getesteten Werte kleine Lücken (≤ 3), so ist Sprungtabelle effizienter und platzsparender
- eventuell kann man mehrere Sprungtabellen für verschiedene, zusammenhängende Mengen generieren
- ein Suchbaum mit Tabellen kann durch *profiling* anders angeordnet werden, so dass häufig genommene Pfade weniger Tests erfordern

Übersetzung in Basic Blocks

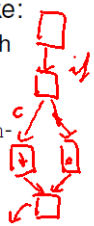
Problem: Wie verknüpft man die verschiedenen Basic Blöcke?
Idee:

- beim Übersetzen einer Funktion: erzeuge einen leeren Basic Block und speichere den Zeiger im Konten der Funktion

Übersetzung in Basic Blocks

Problem: Wie verknüpft man die verschiedenen Basic Blöcke?
Idee:

- beim Übersetzen einer Funktion: erzeuge einen leeren Basic Block und speichere den Zeiger im Konten der Funktion
- reiche diesen Basic Block zur Übersetzung der Anweisungen runter
- eine Zuweisung wird am Ende des Basic Blocks angehängt
- eine zweiseitige **`if`** Anweisung erzeugt drei neue, leere Blöcke:
  1. einen für den `then`-Zweig, verbunden mit aktuellem Block durch jumpz Kante
  2. einen für den **`else`**-Zweig, durch jump Kante verbunden
  3. einen für die nachfolgenden Anweisungen, verbunden mit `then`- und **`else`**-Zweig durch jump Kante



Übersetzung in Basic Blocks

Problem: Wie verknüpft man die verschiedenen Basic Blöcke?
Idee:

- beim Übersetzen einer Funktion: erzeuge einen leeren Basic Block und speichere den Zeiger im Konten der Funktion
- reiche diesen Basic Block zur Übersetzung der Anweisungen runter
- eine Zuweisung wird am Ende des Basic Blocks angehängt
- eine zweiseitige **`if`** Anweisung erzeugt drei neue, leere Blöcke:
  1. einen für den `then`-Zweig, verbunden mit aktuellem Block durch jumpz Kante
  2. einen für den **`else`**-Zweig, durch jump Kante verbunden
  3. einen für die nachfolgenden Anweisungen, verbunden mit `then`- und **`else`**-Zweig durch jump Kante
- ähnlich für andere Konstrukte

Zur besseren Navigation sollten auch *Rückwärtskanten* zwischen den Blöcken eingefügt werden.

Kapitel 5:
Funktionen