Script generated by TTT

Title: Simon: Compilerbau (04.07.2012)

Date: Wed Jul 04 14:04:40 CEST 2012

Duration: 93:52 min

Pages: 93

Kapitel 5:
Funktionen

Aufbau einer Funktion

Die Definition einer Funktion besteht aus

- einem Namen, mit dem sie aufgerufen werden kann;
- einer Spezifikation der formalen Parameter;
- evtl. einem Ergebnistyp;
- einem Anweisungsteil.

In C gilt:

$\text{code}^i_R\ f\ \rho$ = loadc _f mit _f Anfangsadresse des Codes für f

Beachte:

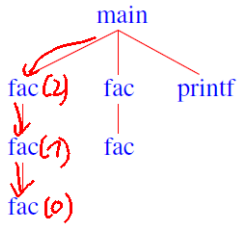
- auch Funktions-Namen müssen eine Adresse zugewiesen bekommen
- da die Größe von Funktionen nicht vor der Übersetzung bekannt ist, müssen die Adressen der Funktionen anschließend eingetragen werden

Speicherverwaltung bei Funktionen

```
int fac(int x) {
  if (x<=0) return 1;
  else return x*fac(x-1);
}
```

```
int main(void) {
  int n;
  n = fac(2) + fac(1);
  printf("%d", n);
}
```

Zu einem Ausführungszeitpunkt können mehrere Instanzen der gleichen Funktion aktiv sein, d. h. begonnen, aber noch nicht beendet sein.
Der Rekursionsbaum im Beispiel:



Speicherverwaltung von Funktionsvariablen

Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

Idee zur Implementierung:

Speicherverwaltung von Funktionsvariablen

Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

Idee zur Implementierung:

- lege einen speziellen Speicherbereich für jeden Aufruf einer Funktion an.
- in sequentiellen Programmiersprachen können diese Speicherbereiche auf dem Keller verwaltet werden

Speicherverwaltung von Funktionsvariablen

Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

Idee zur Implementierung:

- lege einen speziellen Speicherbereich für jeden Aufruf einer Funktion an.
- in sequentiellen Programmiersprachen können diese Speicherbereiche auf dem Keller verwaltet werden
- jede Instanz einer Funktion erhält dadurch einen Bereich auf dem Stack

Speicherverwaltung von Funktionsvariablen

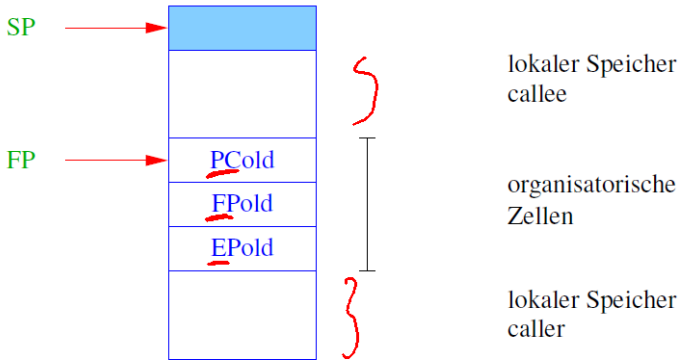
Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

Idee zur Implementierung:

- lege einen speziellen Speicherbereich für jeden Aufruf einer Funktion an.
- in sequentiellen Programmiersprachen können diese Speicherbereiche auf dem Keller verwaltet werden
- jede Instanz einer Funktion erhält dadurch einen Bereich auf dem Stack
- diese Bereiche heißen Keller-Rahmen (oder stack frame)

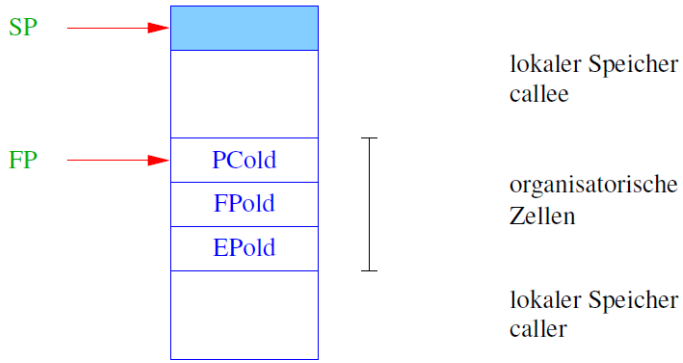
Kellerrahmen-Organisation

- Stapel Präsentation: wächst nach oben, zu höheren Adressen
- SP bestimmt die letzte, benutzte Stapeladresse



Kellerrahmen-Organisation

- Stapel Präsentation: wächst nach oben, zu höheren Adressen
- SP bestimmt die letzte, benutzte Stapeladresse



- FP $\widehat{=}$ Frame Pointer; zeigt auf die letzte organisatorische Zelle
- wird zur Wiederherstellung des letzten Kellerrahmens benutzt

Arbeitsteilung beim Funktionsaufruf

Definition

Sei f die aktuelle Funktion, die die Funktion g aufruft.

- f heißt *caller*
- g heißt *callee*

Der Code für den Aufruf muss auf den Caller und den Callee verteilt werden.
Die Aufteilung kann nur so erfolgen, dass der Teil, der von Informationen des Callers abhängt, auch dort erzeugt wird und analog für den Callee.

Beobachtung:

Den Platz für die aktuellen Parameter kennt nur der Caller:
Beispiel: `printf(char *fi ...)`

Prinzip vom Funktionsaufruf und Rücksprung

Aktionen beim Betreten von g:

1.	Berechnung der Anfangsadresse von g		
2.	Berechnung der aktuellen Parameter		
3.	Retten aller caller-save Register	saveloc	stehen in f
4.	Retten von FP, EP	mark	
5.	Setzen des neuen FP		
6.	Retten von PC und Sprung an den Anfang von g	call	
7.	Setzen des neuen EP	enter	stehen in g
8.	Allokieren der lokalen Variablen	alloc	

Aktionen bei Verlassen von g:

1.	Berechnung des Rückgabewerts		
2.	Rücksetzen der Register FP, EP, SP	return	stehen in g
3.	Rücksprung in den Code von f, d. h. Restauration des PC		
4.	Wiederherstellen der caller-save Register	restoreloc	stehen in f
5.	Aufräumen des Stack	pop k	

Registerverwaltung bei Funktionsaufrufen

Die zwei Registersätze (global und lokal) werden wie folgt verwendet:

- automatische Variablen leben in *lokalen* Registern R_i
- Zwischenergebnisse leben auch in *lokalen* Registern R_i
- Parameter leben in *globalen* Registern R_i (mit $i \leq 0$)
- globale Variablen:

Registerverwaltung bei Funktionsaufrufen

Die zwei Registersätze (global und lokal) werden wie folgt verwendet:

- automatische Variablen leben in *lokalen* Registern R_i
- Zwischenergebnisse leben auch in *lokalen* Registern R_i
- Parameter leben in *globalen* Registern R_i (mit $i \leq 0$)
- globale Variablen: wie nehmen erstmal an, es gäbe keine

Konvention:

- die $i$te Argument einer Funktion wir in Register R_i übergeben

Registerverwaltung bei Funktionsaufrufen

Die zwei Registersätze (global und lokal) werden wie folgt verwendet:

- automatische Variablen leben in *lokalen* Registern R_i
- Zwischenergebnisse leben auch in *lokalen* Registern R_i
- Parameter leben in *globalen* Registern R_i (mit $i \leq 0$)
- globale Variablen: wie nehmen erstmal an, es gäbe keine

Konvention:

- die $i$te Argument einer Funktion wir in Register R_i übergeben
- der Rückgabewert einer Funktion wird in R_0 gespeichert
- lokale Register werden von der aufrufenden Funktion gespeichert

Registerverwaltung bei Funktionsaufrufen

Die zwei Registersätze (global und lokal) werden wie folgt verwendet:

- automatische Variablen leben in *lokalen* Registern R_i
- Zwischenergebnisse leben auch in *lokalen* Registern R_i
- Parameter leben in *globalen* Registern R_i (mit $i \leq 0$)
- globale Variablen: wie nehmen erstmal an, es gäbe keine

Konvention:

- die $i$te Argument einer Funktion wir in Register R_i übergeben
- der Rückgabewert einer Funktion wird in R_0 gespeichert
- lokale Register werden von der aufrufenden Funktion gespeichert

Definition

Sei f eine Funktion die g aufruft. Ein Register R_i heißt

- *caller-saved*, wenn f R_i sichert und g es überschreiben darf
- *callee-saved*, wenn f R_i nicht sichert und g es vor dem Rücksprung wiederherstellen muss

Übersetzung von Funktionsaufrufen

Ein Funktionsaufruf $g(e_1, \ldots e_n)$ wird nun wie folgt übersetzt:

$$
\begin{array}{lcl}
\text{code}^i_R\ g(e_1, \ldots e_n)\ \rho & = & \text{code}^i_R\ g\ \rho \\
 & & \text{code}^{i+1}_R\ e_1\ \rho \\
 & & \vdots \\
 & & \text{code}^{i+n}_R\ e_n\ \rho \\
 & & \text{move}\ R_{-1}\ R_{i+1} \\
 & & \vdots \\
 & & \text{move}\ R_{-n}\ R_{i+n} \\
 & & \text{saveloc}\ R_1\ R_{i-1} \\
 & & \text{mark} \\
 & & \text{call}\ R_i \\
 & & \text{restoreloc}\ R_1\ R_{i-1} \\
 & & \text{move}\ R_i\ R_0
\end{array}
$$

Übersetzung von Funktionsaufrufen

Ein Funktionsaufruf $g(e_1, \ldots e_n)$ wird nun wie folgt übersetzt:

$$
\begin{array}{lcl}
\text{code}^i_R\ g(e_1, \ldots e_n)\ \rho & = & \text{code}^i_R\ g\ \rho \\
 & & \text{code}^{i+1}_R\ e_1\ \rho \\
 & & \vdots \\
 & & \text{code}^{i+n}_R\ e_n\ \rho \\
 & & \text{move}\ R_{-1}\ R_{i+1} \\
 & & \vdots \\
 & & \text{move}\ R_{-n}\ R_{i+n} \\
 & & \text{saveloc}\ R_1\ R_{i-1} \\
 & & \text{mark} \\
 & & \text{call}\ R_i \\
 & & \text{restoreloc}\ R_1\ R_{i-1} \\
 & & \text{move}\ R_i\ R_0
\end{array}
$$

Neue Instruktionen:

- saveloc $R_i\ R_j$ legt die Register $R_i, R_{i+1} \ldots R_j$ auf dem Stapel ab
- mark rettet organisatorische Zellen
- call R_i ruft Funktion auf die an Adresse R_i liegt
- restoreloc $R_i\ R_j$ nimmt $R_j, R_{j-1}, \ldots R_i$ vom Stapel runter

Übersetzung von Funktionsaufrufen

Ein Funktionsaufruf $g(e_1, \ldots e_n)$ wird nun wie folgt übersetzt:

$$
\begin{array}{lcl}
\text{code}^i_R\ g(e_1, \ldots e_n)\ \rho & = & \text{code}^i_R\ g\ \rho \\
 & & \text{code}^{i+1}_R\ e_1\ \rho \\
 & & \vdots \\
 & & \text{code}^{i+n}_R\ e_n\ \rho \\
 & & \text{move}\ R_{-1}\ R_{i+1} \\
 & & \vdots \\
 & & \text{move}\ R_{-n}\ R_{i+n} \\
 & & \text{saveloc}\ R_1\ R_{i-1} \\
 & & \text{mark} \\
 & & \text{call}\ R_i \\
 & & \text{restoreloc}\ R_1\ R_{i-1} \\
 & & \text{move}\ R_i\ R_0
\end{array}
\quad \stackrel{?}{=} \quad
\begin{array}{l}
\text{code}^i_R\ e_1\ \rho \\
\text{move}\ R_{-1}\ R_i \\
\vdots \\
\text{code}^i_R\ e_n\ \rho \\
\text{move}\ R_{-n}\ R_i \\
\text{code}^i_R\ g\ \rho \\
\text{saveloc}\ R_1\ R_{i-1} \\
\text{mark} \\
\text{call}\ R_i \\
\text{restoreloc}\ R_1\ R_{i-1} \\
\text{move}\ R_i\ R_0
\end{array}
$$

Neue Instruktionen:

- saveloc $R_i\ R_j$ legt die Register $R_i, R_{i+1} \ldots R_j$ auf dem Stapel ab
- mark rettet organisatorische Zellen
- call R_i ruft Funktion auf die an Adresse R_i liegt
- restoreloc $R_i\ R_j$ nimmt $R_j, R_{j-1}, \ldots R_i$ vom Stapel runter

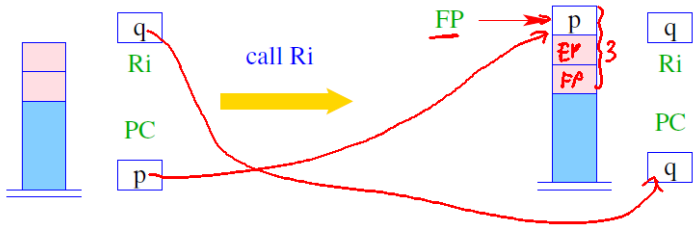
Retten von EP und FP

Der Befehl mark legt Platz für Rückgabewert und organisatorische Zellen an und rettet FP und EP.


```
S[SP+1] = EP;
S[SP+2] = FP;
SP = SP + 2;
```

Aufrufen einer Funktion

Der Befehl call rettet den aktuellen Wert des PC als Fortsetzungs-Adresse und setzt FP und PC.



S[SP] = PC;
SP = SP+1;
FP = SP;
PC = Ri;

Rückgabewerte einer Funktion

Die globalen Register werden auch benutzt um den Rückgabewert zu übermitteln:

$$\begin{aligned} \text{code}^i\ \texttt{return}\ e\ \rho \quad = \quad & \text{code}^i_R\ e\ \rho \\ & \text{move}\ R_0\ R_i \\ & \text{return} \end{aligned}$$

Rückgabewerte einer Funktion

Die globalen Register werden auch benutzt um den Rückgabewert zu übermitteln:

$$\begin{aligned} \text{code}^i\ \texttt{return}\ e\ \rho \quad = \quad & \text{code}^i_R\ e\ \rho \\ & \text{move}\ R_0\ R_i \\ & \text{return} \end{aligned}$$

Alternative ohne Rückgabewert:

$$\text{code}^i\ \texttt{return}\ \rho \quad = \quad \text{return}$$

Rückgabewerte einer Funktion

Die globalen Register werden auch benutzt um den Rückgabewert zu übermitteln:

$$\begin{aligned} \text{code}^i\ \texttt{return}\ e\ \rho \quad = \quad & \text{code}^i_R\ e\ \rho \\ & \text{move}\ R_0\ R_i \\ & \text{return} \end{aligned}$$

Alternative ohne Rückgabewert:

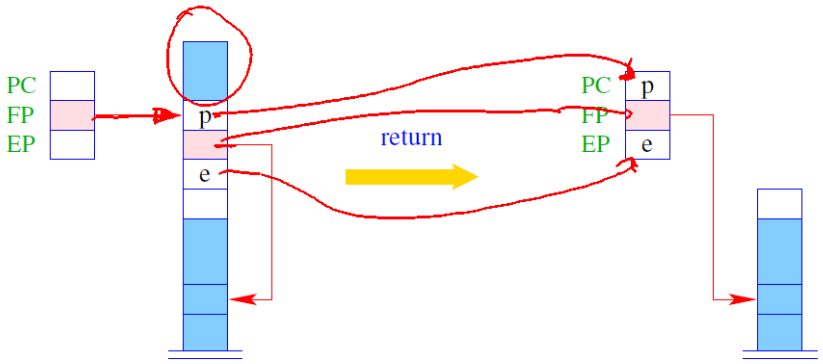
$$\text{code}^i\ \texttt{return}\ \rho \quad = \quad \text{return}$$

Globale Register werden ansonsten im Funktionsrumpf nicht benutzt:

- Vorteil: an jeder Stelle im Rumpf können andere Funktionen aufgerufen werden ohne *globale* Register retten zu müssen
- Nachteil: beim Eintritt in eine Funktion müssen die *globalen* Register gesichert werden

Rücksprung aus einer Funktion

Der Befehl return gibt den aktuellen Keller-Rahmen auf. D.h. er restauriert die Register PC, EP und FP.



PC = S[FP]; EP = S[FP-2];
SP = FP-3; FP = S[SP+2];

Übersetzung ganzer Funktionen

Die Übersetzung einer Funktion ist damit wie folgt definiert:

$$\begin{aligned}\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho\ \ =\ \ & \text{enter}\ q\\ & \text{move}\ R_{l+1}\ R_{-1}\\ & \vdots\\ & \text{move}\ R_{l+n}\ R_{-n}\\ & \text{code}^{l+n+1}\ ss\ \rho'\\ & \text{return}\end{aligned}$$

Randbedinungen:

Übersetzung ganzer Funktionen

Die Übersetzung einer Funktion ist damit wie folgt definiert:

$$\begin{aligned}\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho\ \ =\ \ & \text{enter}\ q\\ & \text{move}\ R_{l+1}\ R_{-1}\\ & \vdots\\ & \text{move}\ R_{l+n}\ R_{-n}\\ & \text{code}^{l+n+1}\ ss\ \rho'\\ & \text{return}\end{aligned}$$

Randbedinungen:

- die Funktion hat n Parameter

Übersetzung ganzer Funktionen

Die Übersetzung einer Funktion ist damit wie folgt definiert:

$$\begin{aligned}\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho\ \ =\ \ & \text{enter}\ q\\ & \text{move}\ R_{l+1}\ R_{-1}\\ & \vdots\\ & \text{move}\ R_{l+n}\ R_{-n}\\ & \text{code}^{l+n+1}\ ss\ \rho'\\ & \text{return}\end{aligned}$$

Randbedinungen:

- die Funktion hat n Parameter
- die lokalen Variablen sind in den Registern $R_1, \ldots R_l$ gespeichert

Übersetzung ganzer Funktionen

Die Übersetzung einer Funktion ist damit wie folgt definiert:

$$\begin{aligned}\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho\ \ =\ \ & \text{enter}\ q\\ & \text{move}\ R_{l+1}\ R_{-1}\\ & \vdots\\ & \text{move}\ R_{l+n}\ R_{-n}\\ & \text{code}^{l+n+1}\ ss\ \rho'\\ & \text{return}\end{aligned}$$

Randbedinungen:

- die Funktion hat n Parameter
- die lokalen Variablen sind in den Registern $R_1, \ldots R_l$ gespeichert
- die Parameter der Funktion stehen in den Registern $R_{-1}, \ldots R_{-n}$

Übersetzung ganzer Funktionen

Die Übersetzung einer Funktion ist damit wie folgt definiert:

$$\begin{aligned}\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho\ \ =\ \ & \text{enter}\ q\\ & \text{move}\ R_{l+1}\ R_{-1}\\ & \vdots\\ & \text{move}\ R_{l+n}\ R_{-n}\\ & \text{code}^{l+n+1}\ ss\ \rho'\\ & \text{return}\end{aligned}$$

Randbedinungen:

- die Funktion hat n Parameter
- die lokalen Variablen sind in den Registern $R_1, \ldots R_l$ gespeichert
- die Parameter der Funktion stehen in den Registern $R_{-1}, \ldots R_{-n}$
- ρ' ist die um lokale Variablen $decls$ und Funktions-Parameter $args$ erweiterte Umgebung ρ

Übersetzung ganzer Funktionen

Die Übersetzung einer Funktion ist damit wie folgt definiert:

$$\begin{aligned}\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho\ \ =\ \ & \text{enter}\ q\\ & \text{move}\ R_{l+1}\ R_{-1}\\ & \vdots\\ & \text{move}\ R_{l+n}\ R_{-n}\\ & \text{code}^{l+n+1}\ ss\ \rho'\\ & \text{return}\end{aligned}$$

Randbedinungen:

- die Funktion hat n Parameter
- die lokalen Variablen sind in den Registern $R_1, \ldots R_l$ gespeichert
- die Parameter der Funktion stehen in den Registern $R_{-1}, \ldots R_{-n}$
- ρ' ist die um lokale Variablen $decls$ und Funktions-Parameter $args$ erweiterte Umgebung ρ
- return nicht immer nötig

Übersetzung ganzer Funktionen

Die Übersetzung einer Funktion ist damit wie folgt definiert:

$$\begin{aligned}\text{code}^1\ t_r\ \texttt{f}(args)\{decls\ \ ss\}\ \rho\ \ =\ \ & \text{enter}\ q\\ & \text{move}\ R_{l+1}\ R_{-1}\\ & \vdots\\ & \text{move}\ R_{l+n}\ R_{-n}\\ & \text{code}^{l+n+1}\ ss\ \rho'\\ & \text{return}\end{aligned}$$

Randbedinungen:

- die Funktion hat n Parameter
- die lokalen Variablen sind in den Registern $R_1, \ldots R_l$ gespeichert
- die Parameter der Funktion stehen in den Registern $R_{-1}, \ldots R_{-n}$
- ρ' ist die um lokale Variablen $decls$ und Funktions-Parameter $args$ erweiterte Umgebung ρ
- return nicht immer nötig

Sind die move Instruktionen immer nötig?

Übersetzen ganzer Programme

Ein Programm $P = F_1; \ldots F_n$ muss eine `main` Funktion enthalten.

$$
\begin{array}{lll}
\text{code}^1\ P\ \rho & = & \text{loadc } R_1 \text{ \_main} \\
 & & \text{mark} \\
 & & \text{call } R_1 \\
 & & \text{halt} \\
\_f_1: & & \text{code}^1\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
\_f_n: & & \text{code}^1\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Übersetzen ganzer Programme

Ein Programm $P = F_1; \ldots F_n$ muss eine `main` Funktion enthalten.

$$
\begin{array}{lll}
\text{code}^1\ P\ \rho & = & \text{loadc } R_1 \text{ \_main} \\
 & & \text{mark} \\
 & & \text{call } R_1 \\
 & & \text{halt} \\
\_f_1: & & \text{code}^1\ F_1\ \rho \oplus \rho_{f_1} \\
 & & \vdots \\
\_f_n: & & \text{code}^1\ F_n\ \rho \oplus \rho_{f_n}
\end{array}
$$

Diskussion:

- $\rho = \emptyset$ enthält die Adressen der globalen Variablen
- ρ_{f_i} enthält die Adressen der lokalen Variablen
- $\rho_1 \oplus \rho_2 = \lambda x . \begin{cases} \rho_2(x) & \text{if } x \in \text{dom}(\rho_2) \\ \rho_1(x) & \text{otherwise} \end{cases}$

Übersetzung der Fakultätsfunktion

Betrachte:

```
int fac(int x) {
 if (x<=0) then
   return 1;
 else
   return x*fac(x-1);
}
```

```
_fac:  enter 5          3 mark+call
       move R1 R-1      save param.
i = 2  move R2 R1       if (x<=0)
       loadc R3 0
       leq R2 R2 R3
       jumpz R2 _A      to else
       loadc R2 1       return 1
       move R0 R2
       return
       jump _B          code is dead
```

```
_A:    move R2 R1       x*fac(x-1)
i = 3  move R3 R1       x-1
i = 4  loadc R4 1
       sub R3 R3 R4     code^3
i = 3  move R-1 R3      fac(x-1)
       loadc R3 _fac
       saveloc R1 R2
       mark
       call R3
       restoreloc R1 R2
       move R3 R0
       mul R2 R2 R3
       move R0 R2       return x*...
       return
_B:    return
```


Themengebiet:

Variablen im Speicher

Register versus Speicher

Bisher:

- alle Variablen sind in Registern gespeichert
- alle Funktionsargumente und Rückgabewerte auch

Register versus Speicher

Bisher:

- alle Variablen sind in Registern gespeichert
- alle Funktionsargumente und Rückgabewerte auch

Beschränkungen:

- in einer realen Maschine gibt es nur endlich viele Register
- C erlaubt es, die Adresse von Variablen zu nehmen
- Felder können nicht übersetzt werden, wegen der Indizierung

Register versus Speicher

Bisher:

- alle Variablen sind in Registern gespeichert
- alle Funktionsargumente und Rückgabewerte auch

Beschränkungen:

- in einer realen Maschine gibt es nur endlich viele Register
- C erlaubt es, die Adresse von Variablen zu nehmen
- Felder können nicht übersetzt werden, wegen der Indizierung

Idee: speichere Variablen auch auf dem Keller

Variablen im Speicher: L-Wert und R-Wert

Variablen können auf zwei Weisen verwendet werden.

Beispiel: `a[x] = y + 1`

Für `y` sind wir am Inhalt der Zelle, für `a[x]` an der Adresse interessiert.

R-Wert von x	=	Inhalt von x
L-Wert von x	=	Adresse von x

Berechne R-Wert und L-Wert im Register R_i:

$\text{code}_R^i\ e\ \rho$	liefert den Code zur Berechnung des R-Werts von e in der Adress-Umgebung ρ
$\text{code}_L^i\ e\ \rho$	analog für den L-Wert

Achtung:
Nicht jeder Ausdruck verfügt über einen L-Wert (z.B.: $x + 1$).

Adressumgebung

Eine Variable kann in einem von vier konzeptionell verschiedenen Bereichen existieren.

1. Global: eine Variable ist global
2. Lokal: eine Variable liegt auf dem aktuellen Kellerrahmen
3. Register: eine Variable ist in lokalem R_i oder globalem R_i

>0 ≤ 0

Adressumgebung

Eine Variable kann in einem von vier konzeptionell verschiedenen Bereichen existieren.

1. Global: eine Variable ist global
2. Lokal: eine Variable liegt auf dem aktuellen Kellerrahmen
3. Register: eine Variable ist in lokalem R_i oder globalem R_i

Entsprechend definieren wir $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ wie folgt:

- $\rho\, x = \langle G, a \rangle$: Variable x ist an absoluter Adresse a gespeichert
- $\rho\, x = \langle L, a \rangle$: Variable x ist an Adresse $FP + a$ gespeichert
- $\rho\, x = \langle R, a \rangle$: Variable x ist im Register R_a gespeichert

Adressumgebung

Eine Variable kann in einem von vier konzeptionell verschiedenen Bereichen existieren.

1. Global: eine Variable ist global
2. Lokal: eine Variable liegt auf dem aktuellen Kellerrahmen
3. Register: eine Variable ist in lokalem R_i oder globalem R_i

Entsprechend definieren wir $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ wie folgt:

- $\rho\, x = \langle G, a \rangle$: Variable x ist an absoluter Adresse a gespeichert
- $\rho\, x = \langle L, a \rangle$: Variable x ist an Adresse $FP + a$ gespeichert
- $\rho\, x = \langle R, a \rangle$: Variable x ist im Register R_a gespeichert

Beachte: eine Variable x kann nur einen Eintrag in ρ haben. Allerdings:

- ρ könnte unterschiedlich sein an verschiedenen Programmpunkten

Adressumgebung

Eine Variable kann in einem von vier konzeptionell verschiedenen Bereichen existieren.

1. Global: eine Variable ist global
2. Lokal: eine Variable liegt auf dem aktuellen Kellerrahmen
3. Register: eine Variable ist in lokalem R_i oder globalem R_i

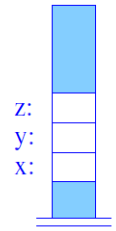
Entsprechend definieren wir $\rho : Var \to \{G, L, R\} \times \mathbb{Z}$ wie folgt:

- $\rho\, x = \langle G, a \rangle$: Variable x ist an absoluter Adresse a gespeichert
- $\rho\, x = \langle L, a \rangle$: Variable x ist an Adresse $FP + a$ gespeichert
- $\rho\, x = \langle R, a \rangle$: Variable x ist im Register R_a gespeichert

Beachte: eine Variable x kann nur einen Eintrag in ρ haben. Allerdings:

- ρ könnte unterschiedlich sein an verschiedenen Programmpunkten
- d.h. x kann einem Register zugewiesen sein
- *und* an einem anderen Punkt einer Speicherzelle

Notwendigkeit von Variablen im Speicher

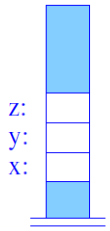


Globale Variablen:

- könnten programm-weit den Registern $R_1 \ldots R_n$ zugeordnet sein

Weiterhin:

Notwendigkeit von Variablen im Speicher

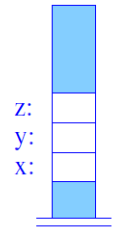


Globale Variablen:

- könnten programm-weit den Registern $R_1 \ldots R_n$ zugeordnet sein
- separate Übersetzung schwierig, da Funktionscode von n abhängt

Weiterhin:

Notwendigkeit von Variablen im Speicher



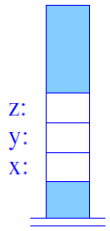
Globale Variablen:

- könnten programm-weit den Registern $R_1 \ldots R_n$ zugeordnet sein
- separate Übersetzung schwierig, da Funktionscode von n abhängt
- besser: speichere globale Variablen im Speicher

Weiterhin:

- eine Variable x (`int` oder `struct`), deren Adresse genommen wurde, muss im Speicher allokiert werden, d.h. $\rho\, x = \langle L, o \rangle$ oder $\rho\, x = \langle G, o \rangle$

Notwendigkeit von Variablen im Speicher



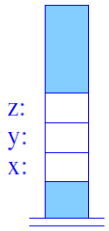
Globale Variablen:

- könnten programm-weit den Registern $R_1 \ldots R_n$ zugeordnet sein
- separate Übersetzung schwierig, da Funktionscode von n abhängt
- besser: speichere globale Variablen im Speicher

Weiterhin:

- eine Variable x (`int` oder `struct`), deren Adresse genommen wurde, muss im Speicher allokiert werden, d.h. $\rho\, x = \langle L, o \rangle$ oder $\rho\, x = \langle G, o \rangle$
- auf ein Feld (array) wird immer mittels Zeiger zugegriffen, muss also auch im Speicher allokiert werden

Notwendigkeit von Variablen im Speicher



Globale Variablen:

- könnten programm-weit den Registern $R_1 \ldots R_n$ zugeordnet sein
- separate Übersetzung schwierig, da Funktionscode von n abhängt
- besser: speichere globale Variablen im Speicher

Weiterhin:

- eine Variable x (**`int`** oder **`struct`**), deren Adresse genommen wurde, muss im Speicher allokiert werden, d.h. $\rho\, x = \langle L, o\rangle$ oder $\rho\, x = \langle G, o\rangle$
- auf ein Feld (array) wird immer mittels Zeiger zugegriffen, muss also auch im Speicher allokiert werden
- Optimierung: Speichere Elemente eines **`struct`** in Registern, während Zeigerzugriffe diese nicht verändern können

Übersetzung von Zuweisungen

Zuweisungen wie `x=2*y` wurden bisher übersetzt durch:

- das Ermitteln das R-Wertes von `2*y` in Register R_i,
- das Kopieren des Inhalts von R_i in das Register $\rho(x)$

Formal: Sei $\rho(x) = \langle R, j\rangle$, dann gilt:

$$\begin{aligned} \text{code}^i_{\text{R}}\ x = e_2\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e_2\ \rho \\ &\qquad \text{move}\ R_j\ R_i \end{aligned}$$

Übersetzung von Zuweisungen

Zuweisungen wie `x=2*y` wurden bisher übersetzt durch:

- das Ermitteln das R-Wertes von `2*y` in Register R_i,
- das Kopieren des Inhalts von R_i in das Register $\rho(x)$

Formal: Sei $\rho(x) = \langle R, j\rangle$, dann gilt:

$$\begin{aligned} \text{code}^i_{\text{R}}\ x = e_2\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e_2\ \rho \\ &\qquad \text{move}\ R_j\ R_i \end{aligned}$$

Aber: undefiniertes Resultat, falls $\rho\, x = \langle L, a\rangle$ oder $\rho\, x = \langle G, a\rangle$.

Übersetzung von Zuweisungen

Zuweisungen wie `x=2*y` wurden bisher übersetzt durch:

- das Ermitteln das R-Wertes von `2*y` in Register R_i,
- das Kopieren des Inhalts von R_i in das Register $\rho(x)$

Formal: Sei $\rho(x) = \langle R, j\rangle$, dann gilt:

$$\begin{aligned} \text{code}^i_{\text{R}}\ x = e_2\ \rho \quad &= \quad \text{code}^i_{\text{R}}\ e_2\ \rho \\ &\qquad \text{move}\ R_j\ R_i \end{aligned}$$

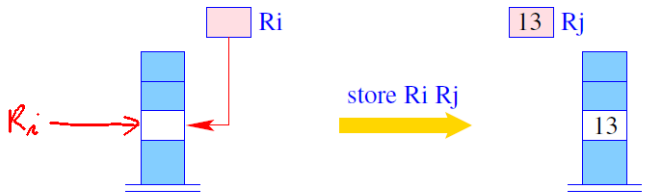
Aber: undefiniertes Resultat, falls $\rho\, x = \langle L, a\rangle$ oder $\rho\, x = \langle G, a\rangle$.

Idee:

- Berechne den R-Werte von e_2 im Register R_i,
- berechne den L-Werte von e_1 im Register R_{i+1} und
- schreibe e_2 and Adresse e_1 mit einem store Befehl.

Übersetzung von L-Werten

Neue Instruktion: store R_i R_j mit Semantik $S[R_i] = R_j$



Definition für Anweisungen:

$$\text{code}^i\ e\ \rho = \text{code}^i_{\text{R}}\ e\ \rho$$

Wie wird $x = e$ (mit $\rho\ x = \langle G, a \rangle$) nun übersetzt?

Übersetzung von L-Werten

Neue Instruktion: store R_i R_j mit Semantik $S[R_i] = R_j$

Definition für Anweisungen:

$$\text{code}^i\ e\ \rho = \text{code}^i_{\text{R}}\ e\ \rho$$

Wie wird $x = e$ (mit $\rho\ x = \langle G, a \rangle$) nun übersetzt?

- Daher definiere für den Fall dass $e_1 = x$ und $\rho\ x = \langle R, j \rangle$ *nicht* gilt:

$$\begin{aligned} \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho \quad = \quad & \text{code}^i_{\text{R}}\ e_2\ \rho \\ & \text{code}^{i+1}_{\text{L}}\ e_1\ \rho \\ & \text{store}\ R_{i+1}\ R_i \end{aligned}$$

Übersetzung von L-Werten

Neue Instruktion: store R_i R_j mit Semantik $S[R_i] = R_j$

Definition für Anweisungen:

$$\text{code}^i\ e\ \rho = \text{code}^i_{\text{R}}\ e\ \rho$$

Wie wird $x = e$ (mit $\rho\ x = \langle G, a \rangle$) nun übersetzt?

- Daher definiere für den Fall dass $e_1 = x$ und $\rho\ x = \langle R, j \rangle$ *nicht* gilt:

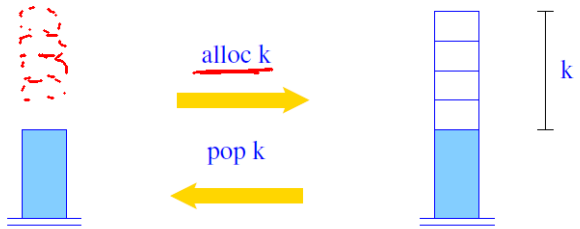
$$\begin{aligned} \text{code}^i_{\text{R}}\ e_1 = e_2\ \rho \quad = \quad & \text{code}^i_{\text{R}}\ e_2\ \rho \\ & \text{code}^{i+1}_{\text{L}}\ e_1\ \rho \\ & \text{store}\ R_{i+1}\ R_i \end{aligned}$$

- Berechne den L-Wert einer Variable wie folgt:

$$\text{code}^i_{\text{L}}\ x\ \rho \quad = \quad \text{loadc}\ R_i\ a$$

Reservierung von Speicher für lokale Variablen

Gegeben: eine Funktion mit k lokalen `int` Variablen, deren Adresse genommen wurden.



alloc k	SP = SP + k;
pop k	SP = SP - k;

Der Befehl alloc k reserviert auf dem Keller Platz für die lokalen Variablen, pop k gibt sie wieder frei.

Zugriff auf lokale Variablen

Zugriffe auf lokale Variablen erfolgt relativ zum aktuellen FP. Darum modifizieren wir code_L für Variablen-Namen.

Für $\rho\, x = \langle L, a\rangle$ definieren wir

$$\text{code}^i_\text{L}\ x\ \rho = \text{loadrc}\ R_i\ a \text{ if } \rho\, x = \langle L, a\rangle$$

Der Befehl loadrc R_i k berechnet die Summe von FP und k.

$$R_i = FP + k$$

Allgemeiner L-Wert von Variablen

Die Adresse einer Variablen wird wie folgt in R_i berechnet:

$$\text{code}^i_\text{L}\ x\ \rho = \begin{cases} \text{loadc}\ R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \text{loadrc}\ R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \end{cases}$$

Allgemeiner L-Wert von Variablen

Die Adresse einer Variablen wird wie folgt in R_i berechnet:

$$\text{code}^i_\text{L}\ x\ \rho = \begin{cases} \text{loadc}\ R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \text{loadrc}\ R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \end{cases}$$

Achtung: für $\rho\, x = \langle R, j\rangle$ ist code^i_L nicht definiert!

Allgemeiner L-Wert von Variablen

Die Adresse einer Variablen wird wie folgt in R_i berechnet:

$$\text{code}^i_\text{L}\ x\ \rho = \begin{cases} \text{loadc}\ R_i\ a & \text{if } \rho\, x = \langle G, a\rangle \\ \text{loadrc}\ R_i\ a & \text{if } \rho\, x = \langle L, a\rangle \end{cases}$$

Achtung: für $\rho\, x = \langle R, j\rangle$ ist code^i_L nicht definiert!

Beobachtung:

- intuitiv: ein Register hat keine Adresse
- ein Register darf während der Codegenerierung nie als L-Wert auftreten
- dies erfordert eine Fallunterscheidung bei der Übersetzung von Zuweisungen

Makro-Befehle zum Zugriff auf lokale Variablen

Definiere: Der Befehl load R_i R_j setzt R_i auf den Wert an Adresse R_j.

Makro-Befehle zum Zugriff auf lokale Variablen

Definiere: Der Befehl load R_i R_j setzt R_i auf den Wert an Adresse R_j.

Damit: loadrc R_i a; load R_j R_i: setze R_j auf x mit $\rho\ x = \langle L, a \rangle$.

Makro-Befehle zum Zugriff auf lokale Variablen

Definiere: Der Befehl load R_i R_j setzt R_i auf den Wert an Adresse R_j.

Damit: loadrc R_i a; load R_j R_i: setze R_j auf x mit $\rho\ x = \langle L, a \rangle$.

Allgemein: Lade Variable x in Register R_i:

$$\text{code}^i_{\text{R}}\ x\ \rho = \begin{cases} \text{loada}\ R_i\ a & \text{if } \rho\ x = \langle G, a \rangle \\ \text{loadr}\ R_i\ a & \text{if } \rho\ x = \langle L, a \rangle \\ \text{move}\ R_i\ R_j & \text{if } \rho\ x = \langle R, i \rangle \end{cases}$$

Makro-Befehle zum Zugriff auf lokale Variablen

Definiere: Der Befehl load R_i R_j setzt R_i auf den Wert an Adresse R_j.

Damit: loadrc R_i a; load R_j R_i: setze R_j auf x mit $\rho\ x = \langle L, a \rangle$.

Allgemein: Lade Variable x in Register R_i:

$$\text{code}^i_{\text{R}}\ x\ \rho = \begin{cases} \text{loada}\ R_i\ a & \text{if } \rho\ x = \langle G, a \rangle \\ \text{loadr}\ R_i\ a & \text{if } \rho\ x = \langle L, a \rangle \\ \text{move}\ R_i\ R_j & \text{if } \rho\ x = \langle R, i \rangle \end{cases}$$

Analog: Für Schreiboperationen definiere:

$$\begin{aligned} \text{storer}\ a\ R_j &\equiv \text{loadrc}\ R_i\ a \\ &\quad\ \text{store}\ R_i\ R_j \\ \text{storea}\ a\ R_j &\equiv \text{loadc}\ R_i\ a \\ &\quad\ \text{store}\ R_i\ R_j \end{aligned}$$

D.h. storea a R_j ist *Makro*. Definiere Spezialfall (mit $\rho\ x = \langle G, a \rangle$):

$$\begin{aligned} \text{code}^i_{\text{R}}\ x = e_2\ \rho &= \text{code}^i_{\text{R}}\ e_2\ \rho \\ &\quad\ \text{code}^{i+1}_{\text{L}}\ x\ \rho \\ &\quad\ \text{store}\ R_{i+1}\ R_i \end{aligned}$$

Datentransfer Instruktionen der R-CMa

Lesezugriffe und Schreibzugriffe der R-CMa :

Instruktion	Semantik	Intuition
load R_i R_j	$R_i \leftarrow S[R_j]$	lade Wert von Adresse
loada R_i c	$R_i \leftarrow S[c]$	lade globale Variable
loadr R_i c	$R_i \leftarrow S[FP + c]$	lade lokale Variable
store R_i R_j	$S[R_i] \leftarrow R_j$	speichere Wert an Adresse
storea c R_i	$S[c] \leftarrow R_i$	schreibe globale Variable
storer c R_i	$S[FP + c] \leftarrow R_i$	schreibe lokale Variable

Instruktionen zur Adressberechnung:

Instruktion	Semantik	Intuition
loadc R_i c	$R_i \leftarrow c$	lade Konstante
loadrc R_i c	$R_i \leftarrow FP + c$	lade Konstante rel. zu FP

Instruktionen für den allgemeinen Datentransfer:

Instruktion	Semantik	Intuition
move R_i R_j	$R_i \leftarrow R_j$	transferiere Wert zw. Registern
move R_i k R_j	$[S[SP + i] \leftarrow S[R_j + i]]_{i=0}^{k-1}$ $R_i \leftarrow SP; SP \leftarrow SP + k$	kopiere k Werte auf den Keller

Bestimmung der Adress-Umgebung

Variablen können verschiedenen Tag in Symboltabelle haben:

1. globale Variablen, die außerhalb von Funktionen definiert werden;
2. lokale (automatische) Variablen, die innerhalb von Funktionen definiert werden;
3. Register (automatische) Variablen, die innerhalb von Funktionen definiert werden.

Beispiel:

```
int x, y;
void f(int v, int w) {
  int a;
  if (a>0) {
    int b;
    g(&b);
  } else {
    int c;
  }
}
```

v	$\rho(v)$
x	⟨ , ⟩
y	⟨ , ⟩
v	⟨ , ⟩
w	⟨ , ⟩
a	⟨ , ⟩
b	⟨ , ⟩
c	⟨ , ⟩

Funktionsargumente auf dem Keller

- C erlaubt sogenannte *variadic functions*
- unbekannte Anzahl an Parametern: $R_{-1}, R_{-2}, \ldots$
- Problem: callee kann auf Register nicht mit Index zugreifen

Beispiel:

```
int printf(const char * format, ...);
char *s =
  "Hello %s!\nIt's %i to %i!\n";

int main(void) {
  printf(s ,"World", 5, 12);
  return 0;
}
```

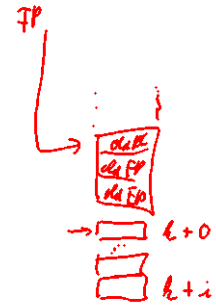

Funktionsargumente auf dem Keller

- C erlaubt sogenannte *variadic functions*
- unbekannte Anzahl an Parametern: $R_{-1}, R_{-2}, \ldots$
- Problem: callee kann auf Register nicht mit Index zugreifen

Beispiel:

```
int printf(const char * format, ...);
char *s =
  "Hello %s!\nIt's %i to %i!\n";

int main(void) {
  printf(s ,"World", 5, 12);
  return 0;
}
```



Idee:

- schiebe *variadic* Parameter von *rechts nach links* auf Keller
- Der erste Parameter liegt direkt unterhalb von PC, FP, EP
- Für einen Prototypen $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k, \ \ldots)$ setzen wir:

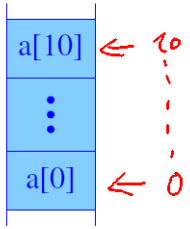
$$x_1 \mapsto \langle R, -1 \rangle \qquad x_k \mapsto \langle R, -k \rangle$$

x_{k+1} wäre $\langle L, -2 - |\tau_{k+1}| \rangle$ $\quad$ x_{k+i} wäre $\langle L, -2 - |\tau_{k+1}| - \ldots - |\tau_{k+i}| \rangle$

Felder

Beispiel: `int [11] a;`

- Das Feld a enthält 11 Elemente und benötigt darum 11 Zellen.
- $\rho\, a$ ist die Adresse des Elements `a[0]`.



Definiere Funktion $|\cdot|$ um den Platzbedarf eines Typs zu berechnen:

$$|t| = \begin{cases} 1 & \text{falls } t \text{ ein Basistyp} \\ k \cdot |t'| & \text{falls } t \equiv t'[k] \end{cases}$$

Dann ergibt sich für die Deklaration $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

$$\begin{aligned} \rho\, x_1 &= 1 \\ \rho\, x_i &= \rho\, x_{i-1} + |t_{i-1}| \qquad \text{für } i > 1 \end{aligned}$$

$|\cdot|$ kann zur Übersetzungszeit berechnet werden, also auch ρ.
Beachte: $|\cdot|$ ist nötig zur Implementierung von C's **`sizeof`** Operator

Übersetzung von Felderzugriffen

Erweitere code_L und code_R auf Ausdrücke mit indizierten Feldzugriffen.

Sei `t[c] a;` die Deklaration eines Feldes a.
Um die Adresse des Elements `a[i]` zu bestimmen, müssen wir $\rho\, a + |t| * \textit{(R-Wert von i)}$ ausrechnen. Folglich:

$$\begin{aligned} \text{code}_L^i\ e_2[e_1]\ \rho \;=\; & \text{code}_R^i\ e_1\ \rho \\ & \text{code}_R^{i+1}\ e_2\ \rho \\ & \text{loadc}\ R_{i+2}\ |t| \\ & \text{mul}\ R_{i+1}\ R_{i+1}\ R_{i+2} \\ & \text{add}\ R_i\ R_i\ R_{i+1} \end{aligned}$$

Übersetzung von Felderzugriffen

Erweitere code_L und code_R auf Ausdrücke mit indizierten Feldzugriffen.

Sei `t[c] a;` die Deklaration eines Feldes a.
Um die Adresse des Elements `a[i]` zu bestimmen, müssen wir $\rho\, a + |t| * \textit{(R-Wert von i)}$ ausrechnen. Folglich:

$$\begin{aligned} \text{code}_L^i\ e_2[e_1]\ \rho \;=\; & \text{code}_R^i\ e_1\ \rho \\ & \text{code}_R^{i+1}\ e_2\ \rho \\ & \text{loadc}\ R_{i+2}\ |t| \\ & \text{mul}\ R_{i+1}\ R_{i+1}\ R_{i+2} \\ & \text{add}\ R_i\ R_i\ R_{i+1} \end{aligned}$$

Bemerkung:

- In C ist ein Feld ein *Zeiger*. Ein deklariertes Feld `a` ist eine *Zeiger-Konstante*, deren R-Wert die Anfangsadresse von `a` ist.
- Formal setzen wir für ein Feld `e` $\text{code}_R^i\ e\ \rho = \text{code}_L^i\ e\ \rho$;
- In C sind äquivalent (als L-Werte, nicht vom Typ):

`2[a]` `a[2]` `a+2`

Strukturen (Records)

Achtung: Komponenten-Namen von Strukturen dürfen sich überlappen.
Hier: Komponenten-Umgebung ρ_{st} bezieht sich auf den gerade übersetzten Struktur-Typ st.
Sei **`struct { int a; int b; } x;`** Teil einer Deklarationsliste.

- `x` erhält die erste freie Zelle des Platzes für die Struktur als Relativ-Adresse.
- Für die Komponenten vergeben wir Adressen *relativ* zum Anfang der Struktur, hier $a \mapsto 0$, $b \mapsto 1$.

Strukturen (Records)

Achtung: Komponenten-Namen von Strukturen dürfen sich überlappen.
Hier: Komponenten-Umgebung ρ_{st} bezieht sich auf den gerade übersetzten Struktur-Typ st.
Sei `struct { int a; int b; } x;` Teil einer Deklarationsliste.

- `x` erhält die erste freie Zelle des Platzes für die Struktur als Relativ-Adresse.
- Für die Komponenten vergeben wir Adressen *relativ* zum Anfang der Struktur, hier $a \mapsto 0, b \mapsto 1$.

Sei allgemein $t \equiv$ `struct { }` $t_1\ v_1; \ldots; t_k\ v_k$ `}`. Dann sei

$$|t| := \sum_{i=1}^{k} |t_i| \qquad \rho_{st}\, v_1 := 0 \qquad \rho_{st}\, v_i := \rho_{st}\, v_{i-1} + |t_{i-1}| \quad \text{für } i > 1$$

Damit erhalten wir:

$$\begin{array}{lcl} \text{code}^i_{\text{L}}\ (e.c)\ \rho & = & \text{code}^i_{\text{L}}\ e\ \rho \\ & & \text{loadc}\ R_{i+1}\ (\rho_{st}\ c) \\ & & \text{add}\ R_i\ R_i\ R_{i+1} \end{array}$$

Zeiger in C

Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

1. Zeiger zu erzeugen, d.h. Zeiger auf Speicherzellen zu setzen; sowie
2. Zeiger zu dereferenzieren, d. h. durch Zeiger auf die Werte von Speicherzellen zugreifen.

Erzeugen von Zeigern:

- Die Anwendung des Adressoperators `&` liefert einen Zeiger auf eine Variable, d. h. deren Adresse ($\widehat{=}$ L-Wert). Deshalb:

$$\text{code}^i_{\text{R}}\ \&e\ \rho \quad = \quad \text{code}^i_{\text{L}}\ e\ \rho$$

Beispiel:
Sei `struct { int a; int b; } x;` mit $\rho = \{x \mapsto 13\}$ und $\rho_{st} = \{a \mapsto 0, b \mapsto 1\}$ gegeben.
Dann ist

$$\begin{array}{lcl} \text{code}^i_{\text{L}}\ (x.b)\ \rho & = & \text{loadc}\ R_{i+1}\ 13 \\ & & \text{loadc}\ R_i\ 1 \\ & & \text{add}\ R_i\ R_i\ R_{i+1} \end{array}$$

Dereferenzieren von Zeigern

Die Anwendung des Operators $\star$ auf den Ausdruck e liefert den Inhalt der Speicherzelle, deren Adresse der L-Wert von e ist:

$$\begin{array}{lcl} \text{code}^i_{\text{R}}\ *e\ \rho & = & \text{code}^i_{\text{L}}\ e\ \rho \\ & & \text{load}\ R_i\ R_i \end{array}$$

Beispiel: Betrachte für

```
struct t { int a[7]; struct t *b; };
int i,j;
struct t *pt;
```

den Ausdruck $e \equiv$ `((pt -> b) -> a)[i+1]`

Wegen `e->a` $\equiv$ `(*e).a` gilt:

$$\begin{array}{lcl} \text{code}^i_{\text{L}}\ (e \to a)\ \rho & = & \text{code}^i_{\text{L}}\ e\ \rho \\ & & \text{loadc}\ R_{i+1}\ (\rho\, a) \\ & & \text{add}\ R_i\ R_i\ R_{i+1} \end{array}$$

Berechnung der R-Werte von Funktionen

- Ähnlich deklarierten Feldern, werden Funktions-Namen als konstante Zeiger auf Funktionen aufgefasst. Dabei ist der R-Wert dieses Zeigers gleich der Anfangs-Adresse der Funktion.
- Achtung! Für eine Variable `int (*) () g` sind die beiden Aufrufe

  `(*g) ()` und `g()`

  äquivalent. Die Dereferenzierungen eines Funktions-Zeigers ist unnötig und wird ignoriert.

Folglich:

$$\begin{array}{lcll} \text{code}_{\text{R}}\ f\ \rho & = & \text{loadc}\ (\rho f) & f \text{ ein Funktions-Name} \\ \text{code}_{\text{R}}\ (*e)\ \rho & = & \text{code}_{\text{R}}\ e\ \rho & e \text{ ein Funktions-Zeiger} \end{array}$$

Übergeben von Zusammengesetzten Parametern

Betrachte folgenden Deklarationen:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

⇝ Wie übergibt man einen nicht-Basistypen als Parameter?

- Idee: *caller* übergibt Zeiger auf Struktur

Übergeben von Zusammengesetzten Parametern

Betrachte folgenden Deklarationen:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

⇝ Wie übergibt man einen nicht-Basistypen als Parameter?

- Idee: *caller* übergibt Zeiger auf Struktur
- Problem: *callee* könnte übergebenen Parameter ändern

Übergeben von Zusammengesetzten Parametern

Betrachte folgenden Deklarationen:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

⇝ Wie übergibt man einen nicht-Basistypen als Parameter?

- Idee: *caller* übergibt Zeiger auf Struktur
- Problem: *callee* könnte übergebenen Parameter ändern
- Lösung: *caller* legt eine Kopie der Struktur an

Übergeben von Zusammengesetzten Parametern

Betrachte folgenden Deklarationen:

```
typedef struct { int x, y; } point_t;
int distToOrigin(point_t);
```

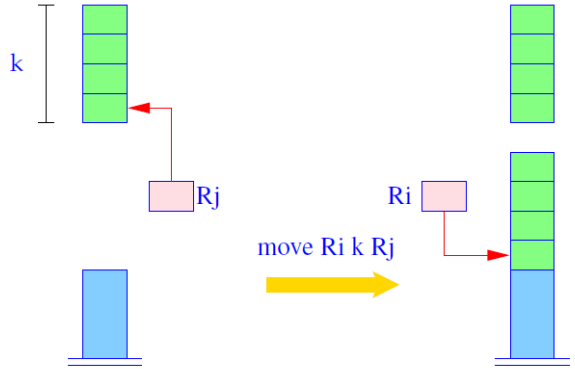
⇝ Wie übergibt man einen nicht-Basistypen als Parameter?

- Idee: *caller* übergibt Zeiger auf Struktur
- Problem: *callee* könnte übergebenen Parameter ändern
- Lösung: *caller* legt eine Kopie der Struktur an

$$\begin{aligned} \text{code}_R^i\ e\ \rho \quad = \quad & \text{code}_L^{i+1}\ e\ \rho \\ & \text{move}\ R_i\ k\ R_{i+1} \end{aligned} \qquad e \text{ eine Struktur der Größe } k$$

Kopien von Speicherbereichen

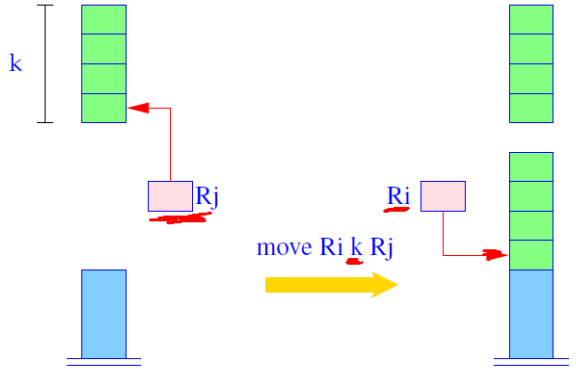
Die move Instruktion kopiert k Elemente auf den Stack.




```
for (i = k-1; i≥0; i--)
      S[SP+i] = S[Rj+i];
Ri = SP;
SP = SP+k;
```


Kopien von Speicherbereichen

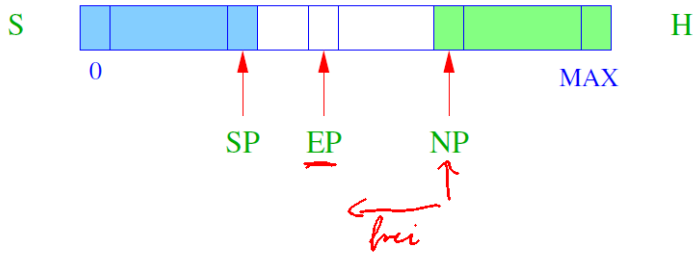
Die move Instruktion kopiert k Elemente auf den Stack.




```
for (i = k-1; i≥0; i--)
      S[SP+i] = S[Rj+i];
Ri = SP;
SP = SP+k;
```


Der Heap

Zeiger gestatten auch den Zugriff auf anonyme, dynamisch erzeugte Datenelemente, deren Lebenszeit nicht dem LIFO-Prinzip unterworfen ist.
⇝ Wir benötigen einen potentiell beliebig großen Speicherbereich H: den Heap (Halde). Implementierung:



NP ≙ New Pointer; zeigt auf unterste belegte Haldenzelle.
EP ≙ Extreme Pointer; zeigt auf die Zelle, auf die der SP maximal zeigen kann (innerhalb der aktuellen Funktion).

Invarianten des Heaps und des Stacks

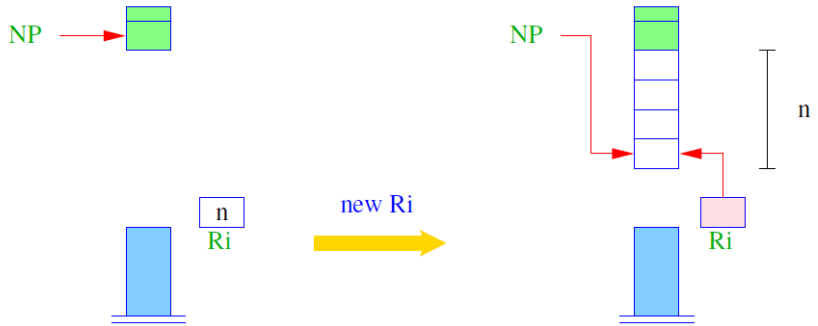
- Stack und Heap dürfen sich nicht überschneiden
- eine Überschneidung kann bei jeder Erhöhung von SP eintreten (Stack Overflow)
- oder bei einer Erniedrigung des NP eintreten (Out Of Memory)

Dynamisch Allokierter Speicher

Es gibt eine weitere Arte, Zeiger zu erzeugen:

- ein Aufruf von `malloc` liefert einen Zeiger auf eine Heap-Zelle:

$$\text{code}^i_R\ \textbf{malloc}\,(e)\ \rho \quad = \quad \text{code}^i_R\ e\ \rho$$
$$\text{new}\ R_i$$

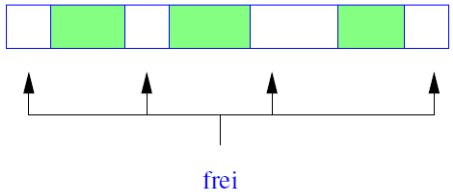



```
if (NP - R[i] <= EP) R[i] = NULL; else {
  NP = NP - R[i];
  R[i] = NP;
}
```


Freigabe von Speicherplatz

Ein mit `malloc` allokierter Bereich muss irgendwann mit `free` wieder freigegeben werden.
Probleme:

- Der freigegebene Speicherbereich könnte noch von anderen Zeigern referenziert (dangling references).
- Nach einiger Freigabe könnte der Speicher etwa so aussehen (fragmentation):



Mögliche Implementierungen:

1. Nimm an, der Programmierer weiß, was er tut. Verwalte dann die freien Abschnitte (etwa sortiert nach Größe) in einer speziellen Datenstruktur;
   ⇝ `malloc` wird teuer
2. Tue nichts, d.h.:

$$\text{code}^i\ \textbf{free}(e)\ \rho \quad = \quad \text{code}^i_R\ e\ \rho$$

   ⇝ einfach und effizient, aber nicht für reaktive Progaramme
3. Benutze eine automatische, evtl. “konservative” Garbage-Collection, die gelegentlich sicher nicht mehr benötigten Heap-Platz einsammelt und dann `malloc` zur Verfügung stellt.

Kapitel 4:

Erweiterte Übersetzung von Funktionen

Realistische Register Maschinen

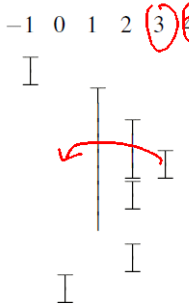
Die R-CMa ist geeignet, auf einfache Weise Code für eine Registermaschine zu erzeugen.

Registerfärbung in der Fakultätsfunktion

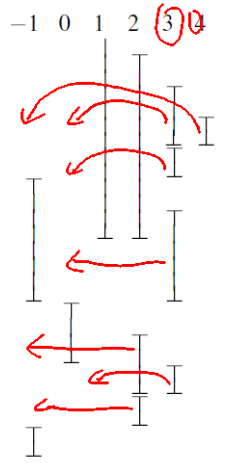
Betrachte: def-use liveness

```
int fac(int x) {
 if (x<=0) then
   return 1;
 else
   return x*fac(x-1);
}
```

```
_fac:  enter 5
       move R1 R-1
       move R2 R1
       loadc R3 0
       leq R2 R2 R3
       jumpz R2 _A
       loadc R2 1
       move R0 R2
       return
       jump _B
```




```
_A:    move R2 R1
       move R3 R1
       loadc R4 1
       sub R3 R3 R4
       move R-1 R3
       loadc R3 _fac
       saveloc R1 R2
       mark
       call R3
       restoreloc R1 R2
       move R3 R0
       mul R2 R2 R3
       move R0 R2
       return
_B:    return
```

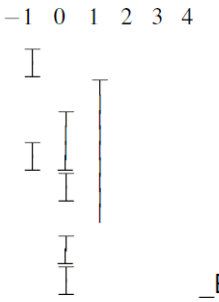


Registerfärbung in der Fakultätsfunktion

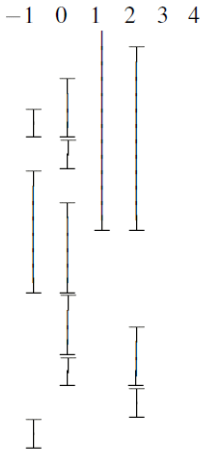
Betrachte: def-use liveness coloring

```
int fac(int x) {
 if (x<=0) then
   return 1;
 else
   return x*fac(x-1);
}
```

```
_fac:  enter 5
       move R1 R-1
       move R0 R1
       loadc R-1 0
       leq R0 R0 R-1
       jumpz R2 _A
       loadc R2 1
       move R0 R2
       return
       jump _B
```




```
_A:    move R2 R1
       move R0 R1
       loadc R-1 1
       sub R0 R0 R-1
       move R-1 R0
       loadc R3 _fac
       saveloc R1 R2
       mark
       call R3
       restoreloc R1 R2
       move R0 R0
       mul R2 R2 R0
       move R0 R2
       return
_B:    return
```



Ausblick

Registerverteilung hat weitere Aufgaben:

- unnötige move Instruktionen müssen vermieden werden
- Variablen müssen auf den Stack ausgelagert werden
  - ⇝ evtl. benötigt dies wiederum Register
- übersetze Funktionen in eine single static assignment Form

Ausblick

Registerverteilung hat weitere Aufgaben:

- unnötige move Instruktionen müssen vermieden werden
- Variablen müssen auf den Stack ausgelagert werden
  - $\rightsquigarrow$ evtl. benötigt dies wiederum Register
- übersetze Funktionen in eine single static assignment Form
- optimale Färbung möglich (allerdings müssen evtl. Register getauscht werden)

$\rightsquigarrow$ Vorlesung *Programmoptimierung*

Schematisch präsentierte liveness-Analyse verbesserungsfähig:

- nach $x \leftarrow y + 1$ ist x nur lebendig wenn y lebendig ist

Ausblick

Registerverteilung hat weitere Aufgaben:

- unnötige move Instruktionen müssen vermieden werden
- Variablen müssen auf den Stack ausgelagert werden
  - $\rightsquigarrow$ evtl. benötigt dies wiederum Register
- übersetze Funktionen in eine single static assignment Form
- optimale Färbung möglich (allerdings müssen evtl. Register getauscht werden)

$\rightsquigarrow$ Vorlesung *Programmoptimierung*

Schematisch präsentierte liveness-Analyse verbesserungsfähig:

- nach $x \leftarrow y + 1$ ist x nur lebendig wenn y lebendig ist
- saveloc hält Register unnötig am Leben $\rightsquigarrow$ Zwischensprache
- gibt es *optimale* Regeln für die liveness-Analyse?

$\rightsquigarrow$ Vorlesung *Programmoptimierung*

Wie berechnet man die liveness Mengen, Registerverteilung zügig?

$\rightsquigarrow$ Vorlesung *Programmoptimierung*

Programming Models and Code Generation WWW