TRANSFORMATION

Dienstag, den 22. Juni1999
Aufgabe der Codeerzeugung 162
Die P-Maschine 166
Transformation einfacher Ausdrücke 169
Transformation von Zuweisungen 171
Transformation von Befehlen 174

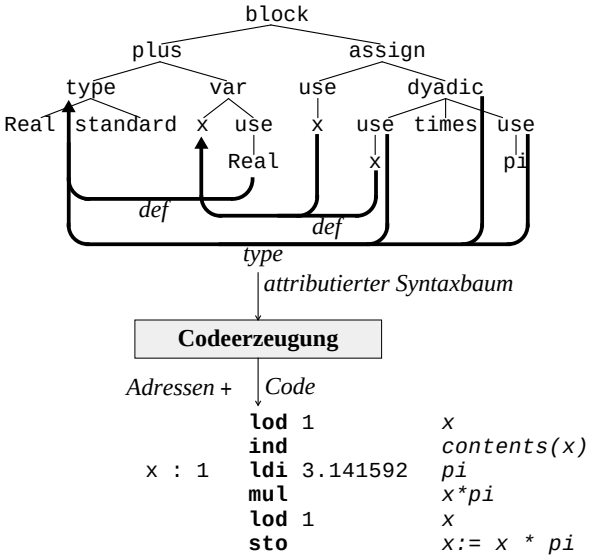
Dienstag, den 29. Juni1999
Darstellung von Datentypen 179
Zugriff auf Variablen 180
Haldenvariablen 194

Dienstag, den 6. Juli1999
Transformation von Prozeduren und Parameterübergabe 197
Transformation des Hauptprogramms 214
Transformation von Moduln (statisch á la Modula und Oberon) 215
Transformation von Klassen und Objekten 216

Aufgabe der Codeerzeugung

Beschränkung
gradlinige Codeerzeugung
- keine Optimierung
- keine ausgefuchste Registerzuteilung

zwei Teilaufgaben
- Speicherorganisation (Größenberechnung, Adreßvergabe)
- Befehlsauswahl (*code selection*)



abstrakte Übersetzung imperativer Programmiersprachen

Konzepte höherer Sprachen...	... und ihre Transformation in Maschinensprache
Datentyp	Byte, Halbwort, Wort, Doppelwort, Block
primitive Operation	arithmetische oder logische Instruktion
Variable	(relative, indirekte) Speicheradresse, *Register*
Zuweisung	Laden und Speichern
Kontrollstruktur	Sprünge und bedingte Sprünge
Block, Prozedurvereinb.	Kellerrahmen, Umgebungszeiger (stat., dyn.)
Funktion / Prozeduraufruf	Unterprogrammsprung + Umgebungsumschaltung
Parameterübergabe	Kopieren von Wert bzw. Adresse
Modul, Objekt	externer Sprung, Objekttabelle

Beschränkungen
sequenzielle Sprachen
"abstrakte" Zielmaschine
- RISC-"Architektur" (*reduced instruction set computer*)
- auf die Quellsprache zugeschnitten
  (die *P-Maschine* wurde für den Züricher Pascal-Übersetzer entwickelt)

Grundlage
Kapitel 2 "Übersetzung imperativer Programmiersprachen" aus
R. Wilhelm, D. Maurer (1996): *Übersetzerbau – Theorie, Konstruktion, Generierung*, Springer, 2. Auflage.
(abgesehen von einigen zu Pascal-bezogenen Stellen)

Transformation

ein rekursive Transformationsschema
code : Programm(teil) $\times$ Adreßumgebung $\rightarrow$ Befehlsfolge
- ordnet jedem Programmteil eine Befehlsfolge zu
- benutzt zusätzlich eine Adreßumgebung α: ID $\rightarrow$ [0 .. s_{max}],
  die Variablennamen Speicheradressen zuordnet
- wird manchmal mit Kontext indiziert (W = Wert, A= Adresse usw.)

vereinfachende Annahmen
wir ignorieren, wie die Struktur des Programms erkannt wird
- ☞ lexikalische und syntaktische Analyse

wir vereinfachen stark, wie die Adreßumgebung bestimmt wird
- ☞ Kontextanalyse

Übersetzung und Ausführung (Interpretation)

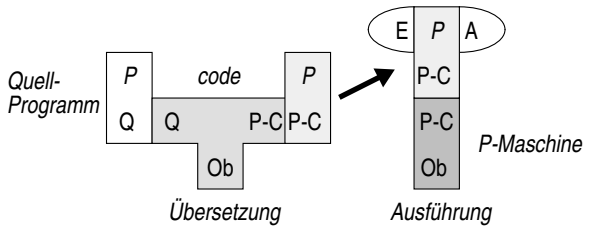
statisch bestimmt (zur Übersetzungszeit)

- Quellprogramm
- Typen
- Zielcode
- relative Adressen von Variablen und Prozeduren
- konstante Werte

dynamisch (zur Laufzeit)

Werte von Variablen und Parametern
Rekursionstiefe
Ausführungsstelle

Vorbild PL0



aber

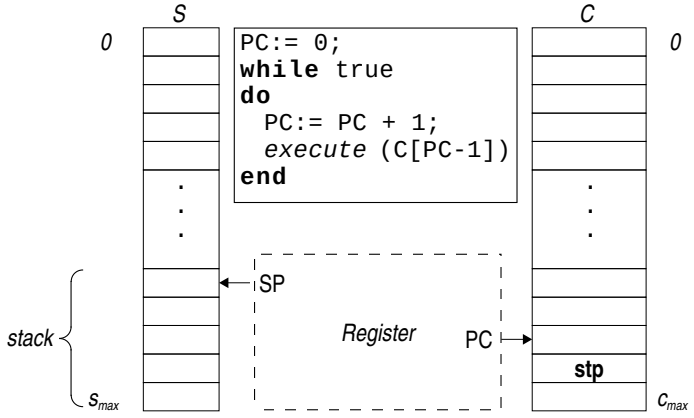
- der Übersetzer wird mit Gleichungen definiert (als funktionales Programm)

Die P-Maschine

Ausstattung

Programmspeicher *C*: `array` c_{max} `of Instruction;`
Datenspeicher *S*: `array` s_{max} `of Word;`
darin wird ein Stapel (*stack*) verwaltet
Spezialregister, z B.

- *program counter* PC
- *stack pointer* SP
- und weitere mehr



Bemerkungen

die komplizierte PC-Arithmetik hat mit Sprüngen zu tun
Programmabbruch durch Ausführen des Haltebefehls **stp**
die Speicherorganisation muß noch verfeinert werden

- Halde
- Prozedurrahmen

Befehle (*instructions*) der P-Maschine

Format

op	mod_1	mod_2

Operationsnummer bzw. Akronym (*opcode*)
bis zu zwei Modifikatoren (Adressen, Zahlenkonstanten

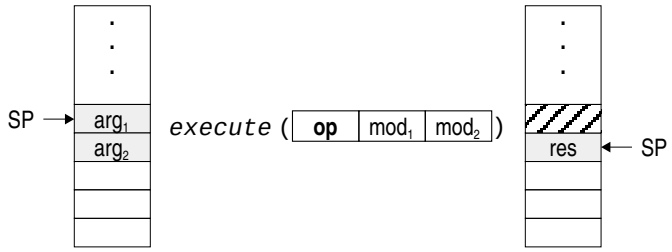
Beispiele

add		

ujp	lab	

chk	lwb	upb

Bedeutung der Befehle



die Operation "*popt*" einige Werte vom Stapel
rechnet (mit Hilfe der Modifikatoren)
sie "*pusht*" ihr Ergebnis auf den Stapel zurück
außerdem kann sie die Register benutzten oder überschreiben

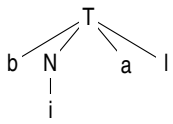
Bemerkungen zu den Befehlen der P-Maschine

Speicherzellen und Modifikatoren sind einfach getypt

Zahlen (N)

- ganz (i)
- Fließkomma

Wahrheitswerte(b): 0..1
Speicheradressen (a): 0..s_{max}
Befehlsadressen (l): 0..c_{max}



vereinfachende Annahmen

Instruktionen sind nicht wirklich gleich lang

- zwischen kleinen und großen Konstanten bzw. Adressen wird nicht unterschieden

Speicherworte sind nicht wirklich gleich lang

- zwischen ganzen und Fließkommazahlen wird nicht unterschieden

Transformation einfacher Ausdrücke

Aufbau einfacher Ausdrücke

E	::=	ID	Bezeichner (*später: Variablenzugriff V*)
	|	C	einfache Konstanten (124, true, '%')
	|	$\oplus$ E	monadischer Operationsaufruf
	|	E $\otimes$ E	dyadischer Operationsaufruf

Befehle der P-Maschine für einfache Operationen und Operanden

Befehl	Bedeutung	mop / dop
$mop_{\oplus}$	`S[SP] := ⊕ S[SP];`	mop_{-} = **neg** $mop_{\mathbf{not}}$ = **not**

$dop_{\otimes}$	`S[SP-1] := S[SP-1] ⊗ S[SP];`
	`SP := SP - 1;`
	(= *push(pop* $\oplus_T$ *pop)*)
		dop_{+} = **add** dop_{-} = **sub**
		dop_{*} = **mul** $dop_{/}$ = **div**
		$dop_{\mathbf{and}}$ = **and**
		$dop_{\mathbf{or}}$ = **or**
		$dop_{=}$ = **equ** $dop_{>=}$ = **geq**
		$dop_{<=}$ = **leq** $dop_{>}$ = **grt**
		$dop_{<}$ = **les** $dop_{/=}$ = **neq**

ldc q	`SP := SP + 1;` `S[SP] := q;`	
ind	`S[SP] := S[S[SP]];`	

mvs q	`for i := q-1 down to 0 do`
	`  S[SP + i] := S[S[SP+i]];`
	`SP:= SP + q -1`	

das Transformationsschema

$code_W\ (c)\ \alpha$ = **ldc** repr(c) (repr bildet Konstanten auf Zahlen ab)
$code_W\ (x)\ \alpha$ = $code_A\ (x)\ \alpha$; **ind** = **ldc** $\alpha(x)$; **ind** (siehe unten)
$code_W\ (x)\ \alpha$ = $code_A\ (x)\ \alpha$; **mvs** *size*($\beta(x)$)
$code_W\ (\oplus\ E)\ \alpha$ = $code_W\ (E)\ \alpha$; $mop_{\oplus}$
$code_W\ (E \otimes F)\ \alpha$ = $code_W\ (E)\ \alpha$; $code_W\ (F)\ \alpha$; $dop_{\otimes}$

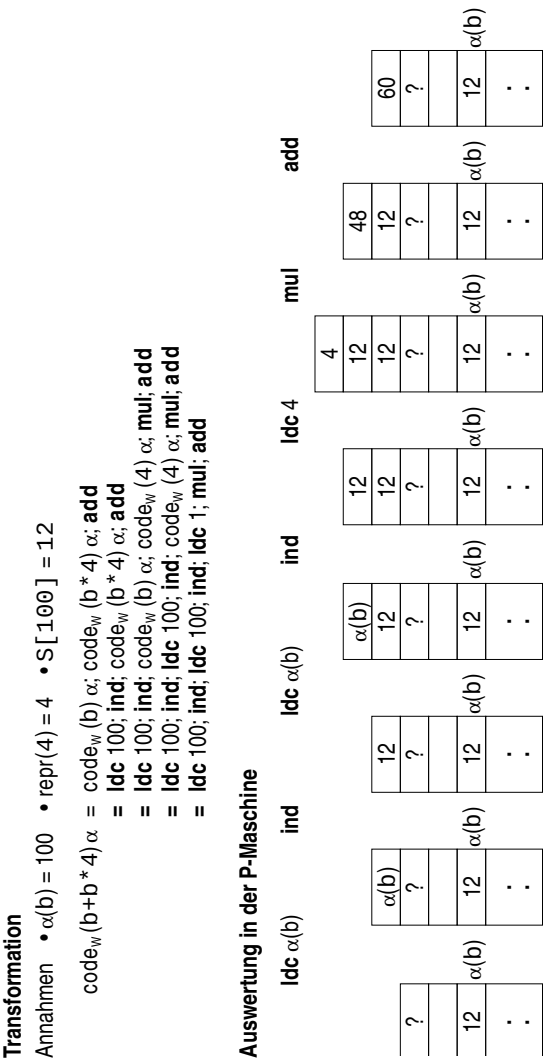
Beispiel: Übersetzung und Auswertung von Ausdrücken

Transformation

Annahmen • $\alpha(b) = 100$ • repr(4) = 4 • `S[100] = 12`

$code_W(b+b*4)\ \alpha$ = $code_W(b)\ \alpha$; $code_W(b*4)\ \alpha$; **add**
= **ldc** 100; **ind**; $code_W(b*4)\ \alpha$; **add**
= **ldc** 100; **ind**; $code_W(b)\ \alpha$; $code_W(4)\ \alpha$; **mul**; **add**
= **ldc** 100; **ind**; **ldc** 100; **ind**; $code_W(4)\ \alpha$; **mul**; **add**
= **ldc** 100; **ind**; **ldc** 100; **ind**; **ldc** 1; **mul**; **add**

Auswertung in der P-Maschine



Zuweisungen

Aufbau

x := e

Bedeutung

- bestimme die Adresse $\alpha(x)$
- bestimme den Wert w(e)
- speichere den Wert w(e) unter der Adresse $\alpha(x)$ ab

Befehle der P-Maschine für einfache Operationen und Operanden

Befehl	Bedeutung
lda q **ldc** q	`SP := SP + 1;` `S[SP] := q;`
ind	`S[SP] := S[S[SP]];`
sto	`S[S[SP-1]] := S[SP];` `SP := SP - 2;`

mvs q	`for i := q-1 down to 0 do`
	`  S[SP + i] := S[S[SP] + i];`
	`SP:= SP + q -11`

(vorläufige Fassung)

das Transformationsschema

$code\ (x := e)\ \alpha$ = $code_A\ (x)\ \alpha$; $code_W\ (e)\ \alpha$; **sto**
$code_A\ (x)\ \alpha$ = **lda** $\alpha(x)$

$code_W\ (c)\ \alpha$ = **ldc** repr(c)
$code_W\ (x)\ \alpha$ = $code_A\ (x)\ \alpha$; **ind** **wenn** size(x) = 1
$code_W\ (x)\ \alpha$ = $code_A\ (x)\ \alpha$; **mvs** *size*($\beta(x)$) **wenn** size(x) > 1
$code_W\ (\oplus\ E)\ \alpha$ = $code_W\ (E)\ \alpha$; $mop_{\oplus}$
$code_W\ (E \otimes F)\ \alpha$ = $code_W\ (E)\ \alpha$; $code_W\ (F)\ \alpha$; $dop_{\otimes}$

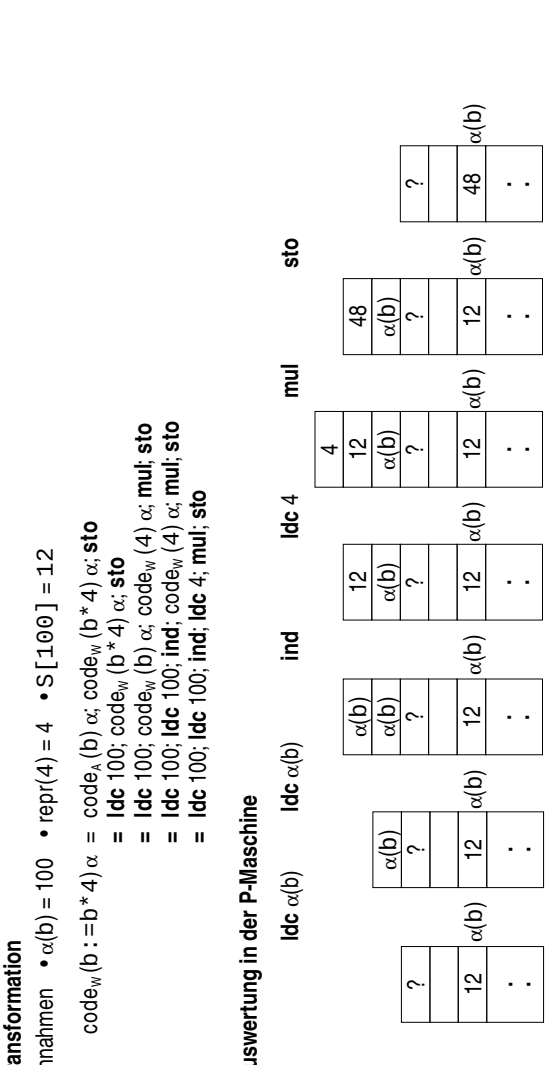
Beispiel: Übersetzung und Ausführung von Zuweisungen

Transformation

Annahmen • $\alpha(b) = 100$ • repr(4) = 4 • `S[100] = 12`

$code_W(b:=b*4)\ \alpha$ = $code_A(b)\ \alpha$; $code_W(b*4)\ \alpha$; **sto**
= **ldc** 100; $code_W(b*4)\ \alpha$; **sto**
= **ldc** 100; $code_W(b)\ \alpha$; $code_W(4)\ \alpha$; **mul**; **sto**
= **ldc** 100; **ind**; $code_W(4)\ \alpha$; **mul**; **sto**
= **ldc** 100; **ind**; **ldc** 4; **mul**; **sto**

Auswertung in der P-Maschine



Güte des Codes

Beispiel 1: `x := y`

code (`x := y`) α = **ldc** α(x);
ldc α(y);
ind;
sto

besserer Code: **ldo** α(y);
sro α(x)

Beispiel 2: `x := x * x`

code (`x := y * y`) α = **ldc** α(x);
ldc α(y);
ind;
ldc α(y);
ind;
mul;
sto

besserer Code: **ldo** α(x);
dpl;
mul;
sro α(x)

Ursache für die geringe Qualität

rekursive Aufteilung ohne Berücksichtigung gleicher Teilausdrücke
auf dem Stapel wird jedes Zwischenergebnis sofort "abgeräumt"
Spezialbefehle wie **dpl** und **sro** werden nicht konsequent ausgenutzt

Befehle

Syntax

C ::= V := E
| C ; C
| **if** E **then** C — Auswahl
| **if** E **then** C **else** C |
| **case** E **of** 0: C_0 | 1: C_1 ... | k: C_k **end**
| **while** E **do** C — Schleifen
| **repeat** C **until** E

for-Schleifen sind ziemlich spezielle **while**-Schleifen
Sprünge werden nicht behandelt

P-Befehle für Kontrollstrukturen

unbedingte, unbedingte und indizierte Sprünge

Befehl	Bedeutung
ujp q	`PC := q`
fjp q	`if S[SP] = false then PC := q end;` `SP := SP - 1`
ijp	`PC := PC + S[SP];` `SP := SP - 1`

Sprungziele: Codeadressen (Zahlen von 0 bis C_{max})
bei der Transformation verwenden wir aber symbolische Namen
(sonst müßte viel mit Codelänge herumgerechnet werden)
praktische Lösung

- nachträgliches Einflicken der Adresse des Sprungzieles, wenn es feststeht
- Assemblierung: systematisches Auflösen symbolischer Marken

Befehlsfolgen, bedingte Befehle

Transformationvon Befehlsfolgen

code(*C* ; *D*) α = code(*C*) α;
code(*D*) α

Transformationvon `if`

code(*C* **if** *E* **then** *C*) α = code(*E*) α;
fjp l;
code(*C*) αl;
l :

code(*C* **if** *E* **then** *C* **else** *D*) α = code(*E*) α;
fjp l;
code(*C*) α;
ujp m;
l : code(*D*) α;
m:

Befehl	Bedeutung
ujp q	`PC := q`
fjp q	`if S[SP] = false then PC := q end;` `SP := SP - 1`

numerische Auswahl

Vereinfachungen

- nur Zahlen als Marken
- vollständigeListe von 0 bis k
- keine Mehrfach-Marken

dies kann man aber leicht erweitern

Transformation

Transformation in geschachteltes **`if`**
Sprungtabelle

code(**case** E **of** 0: C_0 | 1: C_1 ... | k: C_k **end**)
= code(*E*) α;
ijp;
ujp l_0;
...
ujp l_k;
l_0: code(C_0) α;
ujp m;
l_1: code(C_1) α;
ujp m;
...
l_k: code(C_k) α;
m:

Befehl	Bedeutung
ijp	`PC := PC + S[SP];` `SP := SP - 1`

Schleifen

`while`-Schleife

code(**`while`** *E* **`do`** *C*) α = l: code(*E*) α;
fjp m;
code(*C*) α;
ujp l;
m:

`repeat`-Schleife

code(**`repeat`** *C* **`until`** *E*) α = l: code(*C*) α;
code(*E*) α;
fjp l;

`for`-Schleife

code(**`for`** *i* := *E* **`to`** *F* **`do`** *C*) α = code(*i* := *E*;
u := *F*;
`while` *i* <= *u*
`do` *C*;
i := succ(*i*);
`end`) α

code(**`for`** *i* := *E* **`downto`** *F* **`do`** *C*) α
= code(*i* := *E*;
u := *F*;
`while` *i* >= *u*
`do` *C*;
i := pred(*i*);
`end`) α

Übersetzung von Befehlen (Beispiel)

Fakultät iterativ

```
if n > 0
then
   fak:= 1;
   while n > 1
   do
      fak := fak * n;
      n:= n-1
   end;
end
```

Übersetzung

code(**`if`** n>0 **`then`** fak:=1;
`while` n>1 **`do`** fak := fak * n; n:= n-1 **`end`**;
`end`)
= code(n>0); **fjp** L_1;
code(fak:=1; **`while`** n>1 **`do`** fak := fak * n; n:= n-1 **`end`**)
L_1:
= **lod** a_n; **ind**; **ldi** 0; **grt**; **fjp** L_1;
lod a_{fak}; **ldi** 1; **sto**;
code(**`while`** n>1 **`do`** fak := fak * n; n:= n-1 **`end`**)
L_1:
= **lod** a_n; **ind**; **ldi** 0; **grt**; **fjp** L_1;
lod a_{fak}; **ldi** 1; **sto**;
L_2: code(n>1) **fjp** L_1;
code(fak := fak * n; n:= n-1 **`end`**); **ujp** L_2;
L_1:
= **lod** a_n; **ind**; **ldi** 0; **grt**; **fjp** L_1;
lod a_{fak}; **ldi** 1; **sto**;
L_2: **lod** a_n; **ind**; **ldi** 1; **grt**; **fjp** L_1;
lod a_{fak}; **lod** a_{fak}; **ind**; **lod** a_n; **ind**; **mul**; **sto**;
lod a_n; **lod** a_n; **ind**; **ldi** 1; **sub**; **sto**; **ujp** L_2;
L_1:

Darstellung von Datentypen

welche Datentypen gibt es?

T ::= **bool**	primitive Typen
| **int**	
| **char**	
| **real**	
| l: T × r: T	kartesisches Produkt
| l: T + r: T	disjunkte Vereinigung
| [ID_u .. ID_o] **of** T	statische Felder
| [E_u .. E_o] **of** T	dynamische Felder
| **set of** T	Potenzmenge
| **ref** T	Zeiger

Darstellung eines Datentyps im Speicher

seine Werte sind endlich

ihre Größe ist statisch bestimmt (zur Übersetzungszeit)

☞ bei dynamischen Feldern:
Code zur Berechnung der Größe kann statisch bestimmt werden;
er kann bei Anlegen des Feldes ausgeführt werden
darüber hinaus ist die Größe des *Deskriptors* von Feldwerten statisch

Zugriff auf Komponenten einer zusammengesetzten Variablen

V ::= ID	
| V **.** ID	Verbundkomponente
| V **:** s	Projektion einer Vereinigung
| V ^	Inhalt eines Zeigers
| V [E]	Feldindizierung

Adressierung der Komponenten

(relative) Adressen der Komponenten sind statisch bestimmt

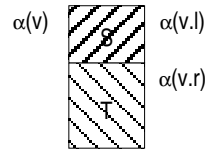
☞ bei dynamischen Feldern:
Adresse der Adresse, in der zur Laufzeit die Feldwerte stehen, ist statisch

kartesisches Produkt

Vereinbarung als Variable

v: l: S × r: T

Abspeicherung



Größe

size(l: *s* × r: *t*) = size(*s*) + size(*t*)

Selektorzugriff

V ::= V **.** ID Verbundkomponente

Adressierung

addieren der lelativen Adresse des Selektorbezeichners

Codierung

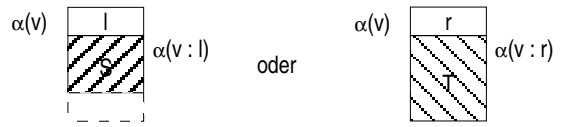
α(l)=0 und α(r)=size(S) sind die Relativadressen

$code_A(v\ .\ s)\ \alpha$ = $code_A(v)\ \alpha$; **inc** α(*s*)

disjunkte Vereinigung

Vereinbarung als Variable
v: l: S + r: T

Abspeicherung



Größe

size(l: *s* + r: *t*) = 1+ max(size(*s*), size(*t*))

Projektion auf Ausgangstyp

V ::= V **:** ID

Adressierung
Überspringen des Variantenfeldes mit Überprüfen der Variante

Codierung

$code_A(v : s)\,\alpha$ = $code_A(v)\,\alpha$; **chv** s; **inc** 1

Befehl	Bedeutung
chv q	`if S[SP] /= q then error("illegal projection")`

Potenzmenge

Vereinbarung als Variable

```
v: set of T
```

Komponententyp T muß eine kleine Kardinalität haben
und sich einfach auf natürliche Zahlen abbilden lassen

Abspeicherung als *Binärwortfolgen*
jedes Bit stellt dar, ob die betreffendeKomponente in der Menge enthalten ist

Größe

size(**set of** *t*) = $2^{size(t)/wordsize}$!

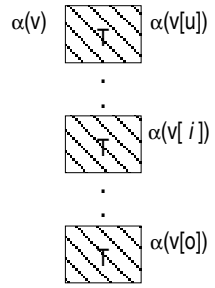
Zugriff auf die Elemente
gibt es eigentlich nicht

E ::= x **in** s

Enthaltensein wird als **and** über Binärwortfolgen implementiert

statische Felder

Vereinbarung als Variable
v: [*u* .. *o*] **of** T



Abspeicherung
als kontinuierliche Sequenz von Worten

Größe
Sei *d* = (o - u + 1) die Komponentenzahl des Feldes und *g* = size(T)

size([*u* .. *o*] **of** *t*) = $d \times g$

Feldindizierung

V ::= V [E]

Adressierung
Addieren der Relativadresse mit Überprüfen des Indexes

Codierung

$code_A(v\,[\,e\,])\,\alpha$ = $code_A(v)\,\alpha$; $code_W(e)\,\alpha$; **chk** u o; **ixa** *g* ; **inc** $-g \times u$

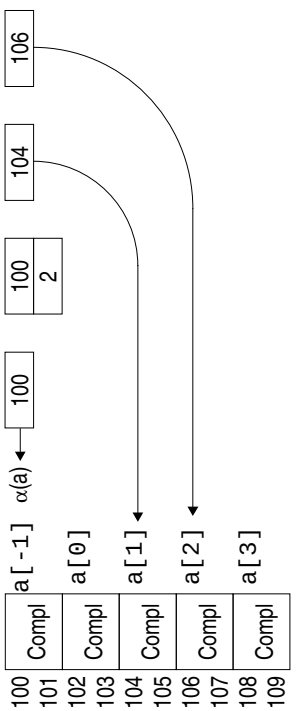
Befehl	Bedeutung
chk p q	`if S[SP] < p or S[SP] > q then error("index error")`
ixa q	`S[SP-1] := S[SP-1] + S[SP]*q; SP := SP-1`

Beispiel zu statischen Feldern

Feldindizierung `a[2]`

Vereinbarung
`var a : [-1 .. +3] of Compl;`
Sei *g* = size(Compl) = 2 ; dann ist *d* = 3 - -1 + 1 = 5 und size(`[-1 .. +3] of Compl`) = $d \times g = 10$

Codierung
$code_A(a[2])\,\alpha$ = $code_A(a)\,\alpha$; $code_W(2)\,\alpha$; **chk** -1 +3; **ixa** *g* ; **inc** $-g \times u$
= **ldc** α(a); **ldc** 2 ; **chk** -1 +3; **ixa** $g \times d$; **inc** $-g \times u$



dynamische Felder

Vereinbarung als Variable (Annahme: *T* ist statisch)
v: [*E* .. *F*] **of** T
die Werte von *E* oder *F* sind *nicht* statisch

Abspeicherung

- ein *Felddeskriptor* wird statisch angelegt (6 Worte)
- die Feldwerte werden dynamisch angelegt (dafür wird Code erzeugt)

Struktur des Felddeskriptors

Selektor	Wert	Definition
fa	fiktive Anfangsadresse	*Adresse* - su
gr	Feldgröße	(ob - un + 1) × size(*T*)
su	Subtrahend zum ersten Element	un × size(*T*)
un	untere Grenze	Wert von *E*
ob	obere Grenze	Wert von *F*

Code zum Anlegen eines dynamischen Feldes

$code_D$(v: [*E* .. *F*] **of** *T*) α = $code_W$(*E*) α; $code_W$(*F*) α; **sad** α(v) size(*T*)

Befehl	Bedeutung

sad p q	`S[p+un]:= S[SP - 1];`
	`S[p+ob]:= S[SP];`
	`S[p+su]:= S[p+un] * q;`
	`S[p+gr]:= S[p+ob] - S[p+un] + 1 * q;`
	`S[p+fa]:= SP - S[p+su] - 2;`
	`SP := SP + S[p+gr] -2;`

Zugriff auf dynamische Felder

Feldindizierung

V ::= V [E]

Codierung

$code_A$(*v* [*e*]) α = $code_A$ (*v*) α; **ind**; $code_W$ (*e*) α; **chd** α(*v*) + 3; **ixa** *g*

Befehl	Bedeutung
ind	`S[SP] := S[S[SP]]`
chd q	`if S[SP] < S[q] or S[SP] > S[q+1]` `then error("index error")`
ixa q	`S[SP-1] := S[SP-1] + S[SP]*q;` `SP := SP-1`

Beispiel: Zugriff auf dynamische Felder

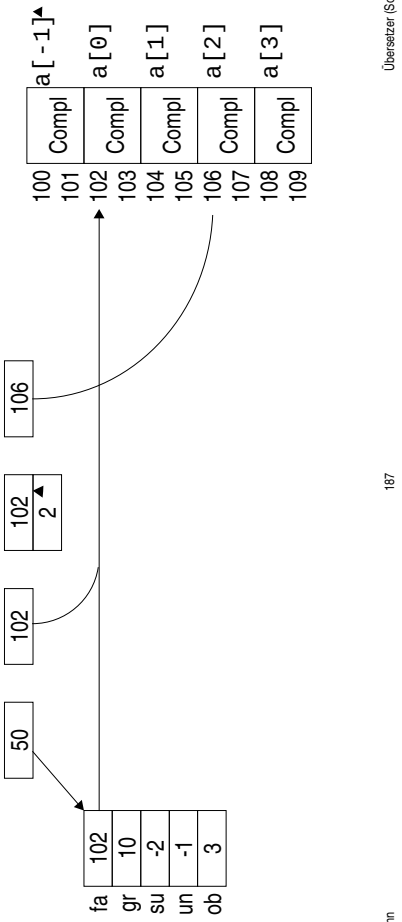
Feldvereinbarung

```
var a : [u .. v] of Compl;   ( * Wert(u) = -1 und Wert(o) = +3 * )
```

Feldzugriff

```
a[i]
```

$code_A$(a [2]) α = $code_A$ (a) α; **ind**; (* Wert(i) = 2 *)
$code_W$ (2) α;
chd α(*v*); **ixa** *g* ;



Mehrfachindizierung bei statischen Feldern

ein mehrdimensionales Feld `var` *v* : [u_1 .. o_1, u_2 .. o_2, ... , u_k .. o_k] `of` *T* **...**

... und ein mehrdimensionales Zugriff *v*[e_1, e_2, ... e_k]
Sei für *i* von 1 bis *k*:

- $d_i = o_k - u_k + 1$ die Spanne der *i*-ten Dimension
- $g^{(i)} = gsize(T) \times \left(\prod_{j=i+1}^{k} d_j \right)$ die Größe der Dimensionen *i* +1 bis *k*
- w_i der Wert des Ausdrucks e_i

die Adresse der Komponente errechnet sich als

$$r = \begin{array}{l} (w_1 - u_1) \times g^{(1)} + (w_2 - u_2) \times g^{(2)} + \ldots \\ (w_{k-1} - u_{k-1}) \times g^{(k-1)} + (w_k - u_k) \times g^{(k)} \end{array}$$

statische Ausdrücke ausmultipliziert ergibt das

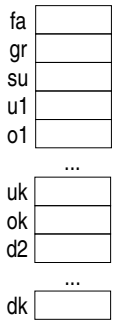
$$r = \underbrace{w_1 \times g^{(1)} + w_2 \times g^{(2)} + \ldots + w_{k-1} \times g^{(k-1)} + w_k \times g^{(k)}}_{g} + \ldots$$
$$\underbrace{\qquad\qquad u_1 \times g^{(1)} + u_2 \times g^{(2)} + \ldots + u_{k-1} \times g^{(k-1)} + u_k \times g^{(k)} \qquad\qquad}$$

die Transformation lautet dann

$code_A$ (*v*[e_1, e_2, ... e_k]) α = $code_A$ (*v*) α;
$code_W$ (e_1) α; **chk** u_1 o_1; **ixa** $g^{(1)}$;
$code_W$ (e_2) α; **chk** u_2 o_2; **ixa** $g^{(2)}$;
...
$code_W$ (e_k) α; **chk** u_k o_k; **ixa** $g^{(k)}$;
inc -*g*

Mehrfachindizierung bei dynamischen Feldern

ein mehrdimensionales dynamisches Feld
`var` *v* : [u_1 .. o_1, u_2 .. o_2, ... , u_k .. o_k] **`of`** *T* **...**
bekommt einen Deskriptor der Länge 3 *k* + 2
für die Spannen der Dimensionen 2 bis *k*



die Transformation lautet dann

$code_A$ (*v*[e_1, e_2, ... e_k]) α
= $code_A$ (*v*) α; **dpl**; **ind**;
$code_W$ (e_1) α; **chd** 3; **ldd** 2 k +3; **mul**;
$code_W$ (e_2) α; **chd** 5; **add**; **ldd** 2 k +4; **mul**;
...
$code_W$ (e_{k-1}) α; **chd** 2 *k*+ 2; **add**; **ldd** 3 k +1; **mul**;
$code_W$ (e_k) α; **add**;
ixa *g*
sli

Befehl	Bedeutung
dpl	`SP := SP + 1;` `S[SP] := S[SP-1]`
chd q	**`if`** `S[SP] < S[S[SP-4]+q]` **`or`** `S[SP] > S[S[SP-4]+q+1]` **`then`** `error("index error")`
ldd q	`SP := SP + 1;` `S[SP] := S[S[SP-3] + q]`
ixa q	`S[SP-1] := S[SP-1] + S[SP]*q;` `SP := SP-1`
sli	`S[SP-1] := S[SP];` `SP := SP - 1`

Beispiel: zweidimensionales dynamisches Feld

ein mehrdimensionales dynamisches Feld **`var`** `a: [-1 .. 2, 1 .. 5] of Compl;`

$code_A$(a[1, 3]) α
= **ldc** α(v); **ldd** 2 k +3; **mul**; **ldc** 3; **add**; **ixa** *g* **sli**
dpl; **chd** 3; **chd** 5;
ind;

Größe von Datentypen (Zusammenfassung)

statische Größe

size(**bool**) = 1 stark vereinfacht!
size(**int**) = 1
size(**char**) = 1
size(**real**) = 1
size(l: *s* × r: *t*) = size(*s*) + size(*t*)
size(l: *s* + r: *t*) = 1+ **max**(size(*s*), size(*t*))
size([*u* .. *o*] **of** *t*) = (o - u + 1) × size(*t*)
size(**set of** *t*) = $2^{size(t)/wordsize}$
size(**ref** *t*) = 1
size([] **of** *t*) = 5 Größe des Feld*deskriptors* ist statisch!

dynamische Größe von Feldern
berechnen durch Code, der den Descriptor füllt und den Platz reserviert

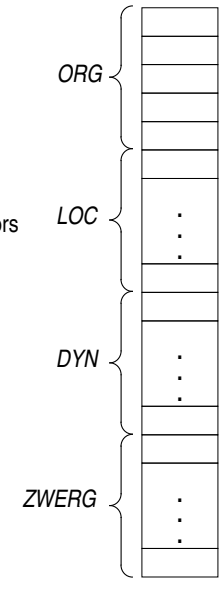
$code_D$(v: [*E* .. *F*] **of** *T*) α = $code_W$(*E*) α; $code_W$(*F*) α; **sad** α(v) size(*T*)

Befehl	Bedeutung

sad p q	`S[p+un]:= S[SP - 1];`
	`S[p+ob]:= S[SP];`
	`SP := SP -2;`
	`S[p+su]:= S[p+un] * q;`
	`S[p+gr]:= (S[p+ob] - S[p+un] + 1) * q;`
	`S[p+fa]:= SP - S[p+su];`
	`SP := SP + S[p+gr];`

Speicherverwaltung für lokale Variablen

statisch bekannte Eigenschaften von Variablen
- Name
- Typ
- Dimension (bei Feldvariablen)
- Selektoren (bei Verbundvariablen)
- Größe ihrer Repräsentation im Speicher
  (bei *dynamischen* Feldern: Größe ihres Descriptors
  und die Formel zur Berechnung der Größe)



Speicherbelegung für eine einzelne Prozedur
4 organisatorische Zellen
statische lokale Variablen
dynamische lokale Variablen
Stapel für Ausdrucksauswertung

Die Adreßumgebung α wird also wie folgt definiert
Für eine Vereinbarungsliste
var v_1 : t_1; v_2 : t_2; ... v_k : t_k;

$$\alpha(v_i) = 4 + \sum_{j=1}^{i-1} size(t_j)$$

Für die Selektoren eines Verbundes **record** s_1 : t_1; s_2 : t_2; ... s_k : t_k;**end**

$$\alpha(s_i) = \sum_{j=1}^{i-1} size(t_j)$$

Annahmen
Variablen und Selektoren sind eindeutig benannt

Variablenzugriffe (Zusammenfassung)

Syntax

V ::= ID	
| V . ID	Verbundkomponente
| V : s	Projektion einer Vereinigung
| V ^	Inhalt eines Zeigers
| V [E]	Feldindizierung

Transformation

$$\begin{aligned}
\text{code}_A(x)\,\alpha &= \textbf{ldc}\ \alpha(x)\\
\text{code}_A(v\ .\ s)\,\alpha &= \text{code}_A(v)\,\alpha;\ \textbf{inc}\ \alpha(s)\\
\text{code}_A(v : s)\,\alpha &= \text{code}_A(v)\,\alpha;\ \textbf{chp}\ s;\ \textbf{inc}\ 1\\
\text{code}_A(v\,\hat{}\,)\,\alpha &= \text{code}_A(v)\,\alpha;\ \textbf{chn};\ \textbf{ind}\\
\text{code}_A(v[i])\,\alpha &= \text{code}_A(v)\,\alpha;\ \text{code}_W(e)\,\alpha;\ \textbf{chk}\ u\ o;\ \textbf{ixa}\ g\ ;\ \textbf{inc}\ -g\times u\\
\text{code}_A(v[i])\,\alpha &= \text{code}_A(v)\,\alpha;\ \textbf{ind};\ \text{code}_W(e)\,\alpha;\ \textbf{chd}\ \alpha(v)+3;\ \textbf{ixa}\ g
\end{aligned}$$

P-Code-Instruktionen zur Adreßberechnung

Befehl	Bedeutung
ldc q	`SP := SP + 1;` `S[SP] := q;`
ind	`S[SP] := S[S[SP]];`
ixa q	`S[SP-1] := S[SP-1] + S[SP]*q;` `SP := SP-1`
chp q	**if** `S[SP] /= q` **then** `error("wrong variant")`
chn	**if** `S[SP] = 0` **then** `error("nil pointer")`
chk p q	**if** `S[SP] < p or S[SP] > q` **then** `error("index error")`
chd q	**if** `S[SP] < S[q]` **or** `S[SP] > S[q+1]` **then** `error("index error")`

dynamische Variablen und die Halde (*heap*)

dynamische Variablen

werden erzeugt durch eine Operation wie **`new`**
und freigegeben

- mit einen Befehl wie **`dispose`** oder
- durch *garbage collection* oder
- bei Programmende

Zugriff mit einer Zeigervariablen
sie können nicht auf dem Stapel angelegt werden

die Halde

der Speicherbereich "über" dem Stapel
wird von oben nach unten gefüllt
Verwaltung mit einem zusätzlichen Register NP (*node pointer*)
Vorbeugung bei Speicherüberlauf:

- es muß immer gelten: SP < NP

damit das nicht bei jeder Erhöhungvon NP *und* SP überprüft werden muß,
führt man ein weiteres Register EP (*extreme pointer*) ein
EP enthält den maximalen Wert von SP bei Ausführung der Prozedur
der Wert von EP ist statisch bestimmt:

- Platz für lokale Variablen, plus
- maximaler Platz für die Auswertung von Ausdrücken

dann muß nur bei jeder Erhöhung von NP
auf Speicherüberlauf geachtet werden
(wenn es **`dispose`** gibt,
braucht man eine zusätzliche Freispeicherverwaltung)

Extreme Stapelausdehnung ...

... läßt sich statisch berechnen

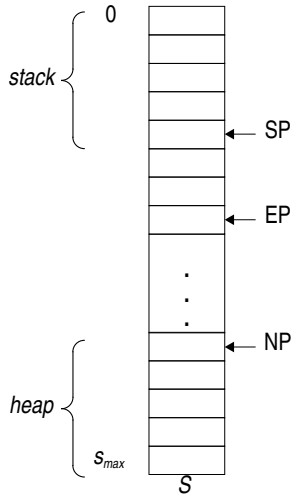
$$\begin{aligned}
\max(x) &= 1\\
\max(c) &= 1\\
\max(E \oplus F) &= 1 + \max(E, F)\\
&\ldots
\end{aligned}$$

Speicherbedarf ist *nicht* symmetrisch
Beispiel

$$\begin{aligned}
&\max(\ ((((1+2)+3)+4)+5)+6\ ) = 2\\
&\max(\ 1+(2+(3+(4+(5+6))))\ ) = 6
\end{aligned}$$

Der Speicherplatzbedarf läßt sich also vorhersagen
das ist eine Art abstrakte Interpretation
statt des Codes werden nur die Veränderungen des Registers *SP* interpretiert

die P-Maschine mit Halde



der Befehl new

Befehl	Bedeutung

new q	`NP := NP - q;`
	if `NP <= EP`
	then `error("storage overflow")`
	`else S[S[SP]] := NP;`
	`SP := SP -1`

chn	**if** `S[SP] = 0` **then** `error("nil pointer")`

(funktioniert so nur für statische Typen)

Prozeduren und Parameterübergabe

was ist bei der Transformation von Prozeduraufrufen zu beachten?

allgemeiner Ablauf

- Sprung zur Prozedurmarke
- Speichern der Rücksprungadresse (alter Wert von PC)

Parameterübergabe

- Parameter liegen oben auf dem Stapel
- Nach Rückkehr liegt dort das Ergebnis (bei Funktionen)

lokale Variablen

- werden in einem eigenen Datenbereich (*frame*) für die Prozedur angelegt
- nicht-lokale Variablen müssen zugreifbar bleiben

Rekursion

- während eine Prozedur aktiv ist, kann sie sich selbst aufrufen
- mehrere *Inkarnationen* können gleichzeitig existieren
- Jede Inkarnation braucht eigene Instanzen ihrer Variablen

Konsequenzen

- bei jedem Eintritt werden neue Instanzen auf dem Stapel angelegt und bei Austritt wieder freigegeben
- der Stapel ist ein Stapel von Prozedurrahmen, jeder Rahmen wird wiederum selbst als Stapel verwaltet (wie bisher)
- Variablenadressen sind nicht vollständig statisch bekannt, Adressen *lokaler* Variablen sind relativ zu einem Register FP (*frame pointer*) nach der Formel

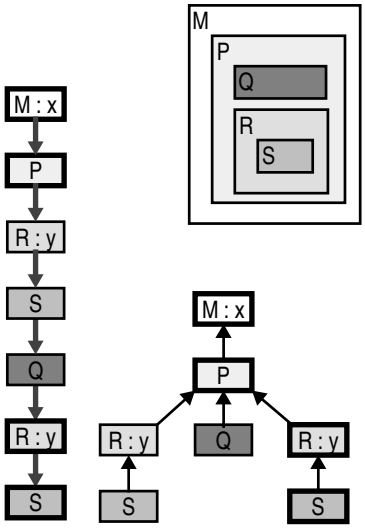
  $$FP + \alpha(v)$$

  Adressen *nichtlokaler* Variablen werden ähnlich bestimmt

statische Vorgänger

Blockschachtelung

```
program M
  var x
  proc P
    proc Q
      R

    proc R
      var y
      proc S
        x := y
        Q
      S
    R
  P
```



Aufruffolge

Geschichte der Prozeduraufrufe
an einem Punkt der Programmausführung
Verkettung dynamischer Vorgänger (oder Nachfolger)

Aufrufbaum

Protokoll aller Prozeduraufrufe bei einer Programmausführung

statischer Vorgängerbaum

Verweis auf die Vereinbarungen nicht-lokaler Variablen

nicht-lokaler Zugriff auf Variablen

Implementierung

in jedem Rahmens wird ein Verweis auf den statischen Vorgänger gespeichert
(als Verweis auf dessen FP, mit relativer Adresse 0)
beim Laden und Speichern wird diesem Verweis nachgegangen
wie weit, ergibt sich aus der Differenz der Schachtelungstiefen

Transformation des Variablenzugriffs

die zusätzliche Umgebung λ
ordnet jedem Bezeichenr *x* die aktuelle Vereinbarungstiefe zu

$$\text{code}_A\,(x)\,(\alpha.\,\lambda)\,l \;=\; \textbf{lda}\; l\text{–}\lambda(x)\;\alpha(x)$$

Befehl	Bedeutung
lod d q	`SP := SP + 1;` `S[SP] := S[`*base*`(d, FP) + q];`
lda d q	`SP := SP + 1;` `S[SP] :=` *base*`(d, FP) + q;`

mvs q	**for** `i := q-1` **down to** `0` **do**
	`  S[SP + i] := S[S[SP] + i];`
	`SP:= SP + q -1`

str d q	`S[`*base*`(d, FP) + q] := S[SP];` `SP := SP - 1;`
base`(d, a) =` **if** `d = 0` **then** `a` **else** *base*`(d-1, S[a])`	

Alternative

ein globaler *Rahmenvektor* enthält die Zeiger auf die statischen Vorgänger

Beispiel: nicht-lokaler Zugriff

Blockschachtelung

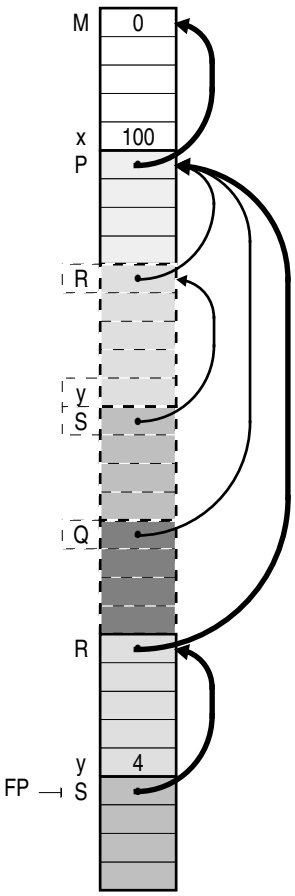
```
program M
  var x
  proc P
    proc Q
      R

    proc R
      var y
      proc S
        x := y
        Q
      S
    R
  P
```

Es sind $\lambda(x) = 0$ und $\lambda(y) = 2$

code(x := y) $\alpha\,\lambda$ 4
= **lda** α(y) 1;
ind;
str α(x) 3

base(3,FP) = FP_M

base(1,FP) = FP_R



Parameterübergabe in Programmiersprachen

… P(*A*) … ⇒ **proc** P (**?**f: *T*) **is** *C* **end**

Kopiermechanismen (f ist eine lokale Variable)

Wertparameter (*call by value*)

- der Wert des Ausdrucks *A* wird vor Ausführung von *C* an f zugewiesen

Ergebnisparameter (*call by result*)

- der Wert von f wird nach Ausführung von *C* an die Variable *A* zugewiesen

Wert-Ergebnisparameter (*call by value result*)

- der Wert der Variablen *A* wird vor Ausführung von *C* an f zugewiesen und nach Ausführung von *C* an A zurücküberwiesen

Funktionsergebnisse werden ähnlich wie Ergebnisparameter behandelt

- der Wert des fiktiven Parameters ERG_P wird während der Ausführung von *C* gesetzt und danach als Wert der Funktion P abgeliefert

Definitorische Mechanismen (f wird eine Aliasbezeichnung)

Konstantenparameter

- f ist eine Konstantenbezeichnung für den Wert des Ausdrucks *A*

Variablenparameter (*call by reference*)

- f ist eine Variablenbezeichnung für die mit *A* bezeichnete Variable

Prozedurparameter

- f ist eine Prozedurbezeichnung für die mit *A* bezeichnete Prozedur

Exoten

Namensparameter (*call by name*)

- bei jeder Benutzung von f in *C* wird der Ausdruck *A* neu ausgewertet

verzögerte Parameter (*call by need, lazy evaluation*)

- bei der ersten Benutzung von f in *C* wird der Ausdruck *A* ausgewertet; bei jeder weiteren Benutzung wird dieser Wert dann wieder benutzt

Implementierung der Parameterübergabe

Grundsatz

auf dem Stapel zwischen den Rahmen von Aufrufer Q und Aufgerufenem P

proc Q **is** … P(*A*) … ⇒ **proc** P (**?**f: *T*) **is** *C*

Wert-Ergebnisparameter

[Wert von *A* auf den Stapel legen]	f wird als Adresse von *A* benutzt [Wert von f an *A* zuweisen]

wenn f ein *dynamisches Feld* ist: (nur als Wertparameter)

Deskriptor von *A* auf den Stapel legen	Wert von *A* kopieren f wird als Adresse von *A* benutzt

- Weshalb dürfen dynamische Felder keine Ergebnisparameter sein?

Konstantenparameter

wie Wertparameter, aber f darf nicht überschrieben werden

Variablenparameter

Adresse von *A* auf den Stapel legen	f ist die *indirekte* Adresse von *A*

Prozedurparameter

Deskriptor von *A* auf den Stapel legen	f ist die *indirekte* Adresse von *A*

Namensparameter

Code für *A* wird angelegt	und bei jeder Benutzung von f ausgeführt

verzögerte Parameter

Code für *A* wird angelegt	bei der ersten Benutzung von f ausgeführt, gespeichert und wieder benutzt

Übergabe von Feldern an "offene"

Aufbau eines Deskriptors für ein statisches Feld

code(*a*) α λ lev
= **ssb** *lwb*(τ(*a*)) *upb*(τ(*a*));
sss *size*(*elemtype*(τ(*a*)))
lda (l - λ(*a*) α(*p*)
ssa

Kopieren des Deskriptors eines dynamischen Feldes (oder Feldparameters)

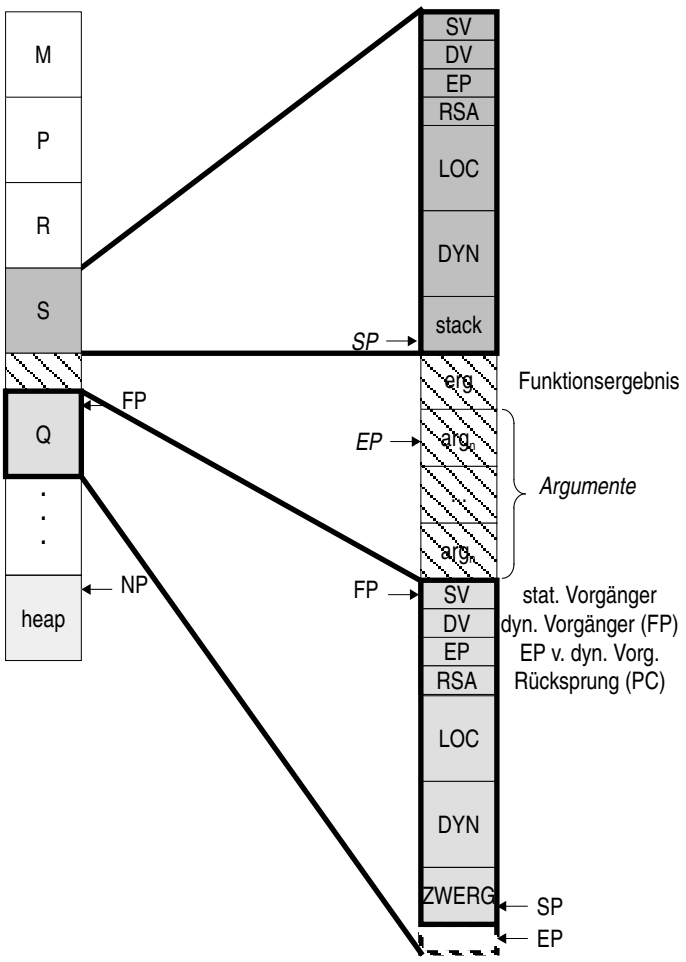
code(*a*) α λ lev
= **lda** (l - λ(*a*) α(*p*)
mvs 5

Kopieren der Werte eines dynamischen Feldparameters (im Rumpf)

$code_{DP}$(*f*: [] **of** *T*) α λ lev
= **mda** α(*f*)

Befehl	Bedeutung
ssb u o	S[SP+un]:= u; S[SP+ob]:= o;
sss g	S[SP+su]:= S[SP+un] * g; S[SP+gr]:= (S[SP+un] - S[SP+ob] + 1) * g SP := SP + 4;
ssa	S[SP-4]:= S[SP] - S[SP-2];
mda q	**for** i := 1 **to** S[FP+q+gr] **do** S[SP+i] := S[S[FP+q+fa]+S[FP+q+su]] **end**; S[FP+q] := SP +1 - S[FP+q+su]; SP := SP + S[FP+q+gr]

vollständige Speicherorganisation der P-Maschine



Transformation des Prozeduraufrufs

Schritte in der Umgebung des Aufrufs

- *Platz fürs Funktionsergebnis reservieren* **isp**
- Argumente auswerten
- statischen Vorgänger setzen und die Register SP und EP retten **mst**
- Rücksprungadresse (PC) retten, FP setzen und zum Unterprogramm springen **cup**
- SP setzen
- Platz für dynamische Felder reservieren und Werte kopieren
- EP setzen

Transformation

code($p(e_1, ..., e_k)$) $\alpha\,\lambda$ lev
= **isp** size(restype(p));
code$_X$(e_1) $\alpha\,\lambda$ lev
...
code$_X$(e_k) $\alpha\,\lambda$ lev
mst lev - λ(p)
cup α(p)

(X = A für Referenzparameter, X= W für Wertparameter)

Befehl	Bedeutung
isp p	`SP := SP + p;`

mst d	`S[SP+1] := base(d, FP);`
	`S[FP+2] := FP;`
	`S[FP+3] := EP;`

cup q	`S[SP+4] := PC;`
	`FP := SP + 1;`
	`PC := q`

Transformation der Prozedurvereinbarung

Schritte in der Umgebung der Vereinbarung

- SP setzen **isp**
- Platz für dynamische Wertparameter (Felder) reservieren; Werte kopieren
- Platz für dynamische lokale Variablen (Felder) reservieren
- EP setzen **sep**
- *Rumpf ausführen*
- Register restaurieren und zurückspringen **ret**

Transformation

- *extreme* ist dder maximale Platzbedarf für Zwischenergebnisse

code(**proc** p(P): T **let** D **in** C) $\alpha\,\lambda$ lev
= α(p): **isp** *size*(D) Platz für lokale Variablen
code$_{DA}$(P) $\alpha\,\lambda$ lev+1 Kopieren dynamischer Parameter
code$_{DD}$(D) $\alpha\,\lambda$ lev+1 Allokieren dyn. lokaler Variablen
sep *extreme*(p)
code(C) $\alpha\,\lambda$ lev+1
l: **ret** *size*(P)

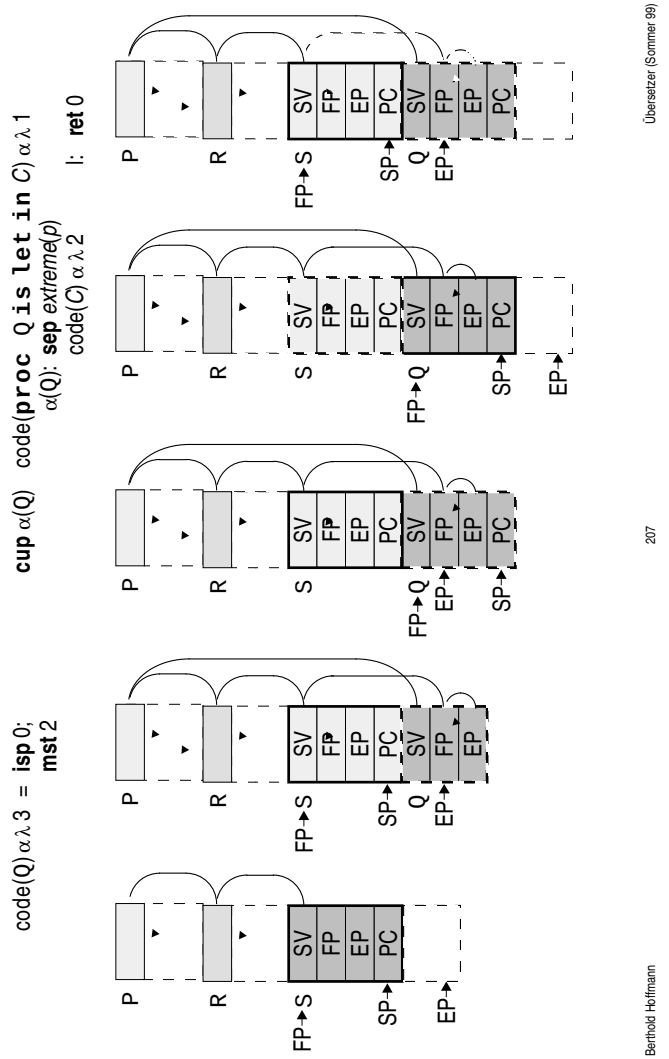
Umgang mit dem Rückgabewert (neuer Befehl **return** E)

code(**return** E) $\alpha\,\lambda$ lev: (hier nur für *size*(T)=1
= code$_W$(E) $\alpha\,\lambda$ lev; **sto** α(ERG$_p$); **ujp** l

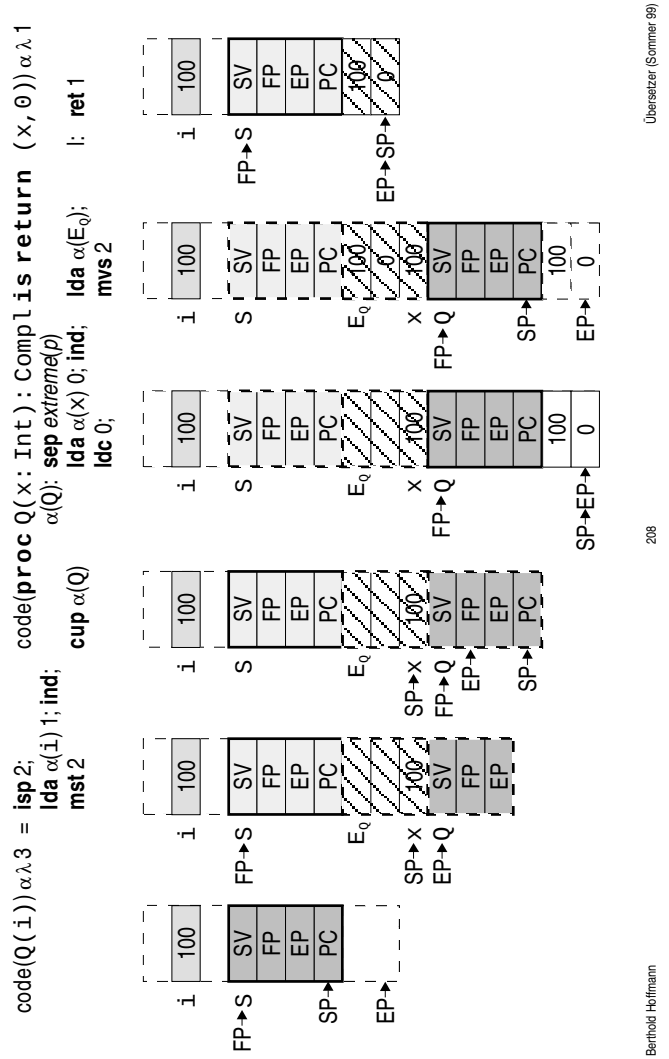
Befehl	Bedeutung
isp p	`SP := SP + q;`
sep p	`EP := SP + p;` `if EP >= NP then error ("store overflow")`

ret p	`SP := FP - p - 1;`
	`PC:= S[FP+3];`
	`EP  := S[FP+2];`
	`if EP >= NP then error ("store overflow")`
	`FP:= S[FP+1];`

Beispiel: Aufruf einer Prozedur (ohne Parameter)



Beispiel: Aufruf einer Prozedur (mit statischem Wertparameter und Ergebnis)



Beispiel: Aufruf einer Prozedur (mit dynamischem Wertparameter)

```
code(Q(a)) α λ 3 = isp 0;
                   lda α(a) 1;
                   mvs 5;
                   mst 2;
                   cup α(Q);

code(proc Q(x: [] Compl) : is ... x[1]... ) α λ 1
      α(Q): mvd α(x);
            sep extreme(p);
                   lda α(x) 0; ind;
                   ldc 1;
                   ixa 2;
                l: ret 5
```


Beispiel: Aufruf einer Prozedur (mit Variablenparameter)

```
code(Q(i)) α λ 3 = isp 0;
                   lda α(i) 1;

code(proc Q(x: ref T) is ... x := ...) α λ 1
      α(Q): sep extreme(p)
                   lda α(x) 0;
                   ldc / mvs ;
                   sto;
                l: ret 1

mst 2
cup α(Q)
```


Prozeduren als Werte und Parameter

Was beschreibt eine Prozedur zur Laufzeit ?

ihre Codeadresse

ihre statische Umgebung

- Prozedurdesktiptoren sind Doppelworte

Transformation eines Prozedurarguments

vereinbarte Prozedur

$code_P(p)\ \alpha\ \lambda\ l$ = **ldc** $\alpha(p)$; — Codeadresse von p
lda $(l - \lambda(p))$ 0 — statischer Vorgänger von p

formaler Prozedurparameter ($\alpha(p)$ ist relative Adresse des Deskriptors)

$code_P(p)\ \alpha\ \lambda\ l$ = **lda** $(l - \lambda(p))\ \alpha(p)$; — Adresse des Deskriptors von p
mvs 2 — Deskriptorwert

Befehl	Bedeutung
lda d q	`SP := SP + 1;` `S[SP] := base(d, FP) + q;`

mvs q	`for i := q-1 down to 0 do`
	`  S[SP + i] := S[S[SP] + i];`
	`SP:= SP + q -1`

base(d, a) = **if** d = 0 **then** a **else** *base*(d-1, S[a])	

Transformation des Aufrufs einer formalen Prozedur

Transformation

$code(p(e_1, ..., e_k))\ \alpha\ \lambda$ lev
= **isp** *size*(*restype*(*p*));
$code_X(e_1)\ \alpha\ \lambda$ lev
...
$code_X(e_k)\ \alpha\ \lambda$ lev
msf lev - $\lambda(p)\ \alpha(p)$
cpi $\alpha(p)$

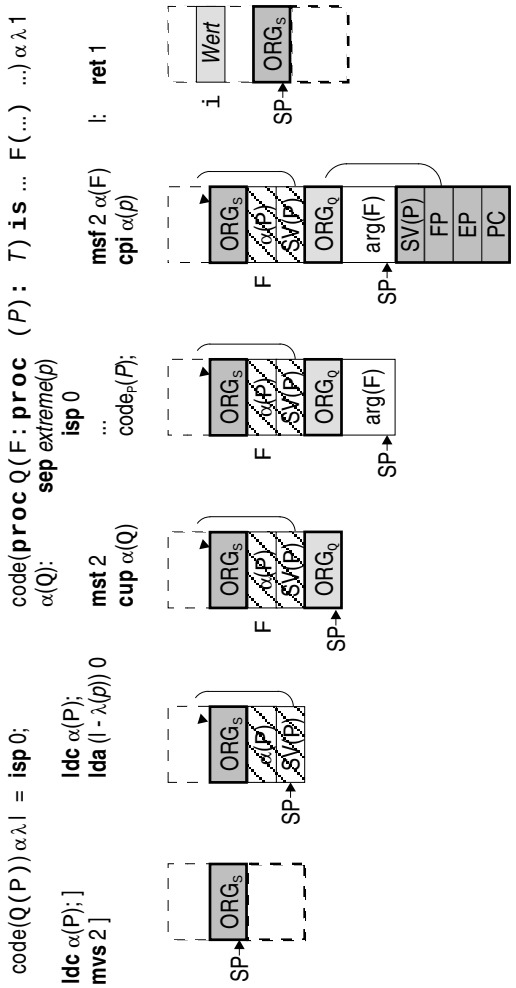
P-Befehle

Befehl	Bedeutung
isp p	`SP := SP + q;`

msf d q	`S[SP+1] := S[base(d, FP) + q + 1];`
	`S[FP+2] := FP;`
	`S[FP+3] := EP;`

cpi d q	`S[SP+4] := PC;`
	`FP := SP + 1;`
	`PC := S[base(d, S[FP+2]) + q]`

Beispiel: Aufruf einer Prozedur (mit Prozedurparameter)



das Hauptprogramm

Aufbau

M ::= **proc** main **is let** *D* **in** *C*

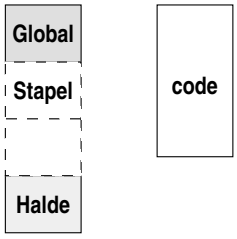
Transformation

- *extreme* ist dder maximale Platzbedarf für Zwischenergebnisse

code(**proc** main **is let** *D* **in** *C*) $\alpha_0\,\lambda_0$

= **isp** *size*(*D*) Platz für lokale Variablen
$code_D(D)\,\alpha\,\lambda\,1$ Allokieren dyn. lokaler Variablen
sep *extreme*(main)
code(*C*) $\alpha\,\lambda\,1$
l: **stp**

Organisation von Speicher und Code



Moduln (statisch á la Modula und Oberon)

sind Verbunde, die auch Prozeduren (und Funktionen) enthalten

module *M* **imports** $i_1 \ldots i_k$
var v_1: ... v_i: ...
proc p_1 ...
...
proc p_n ...
end

Speicherorganisation

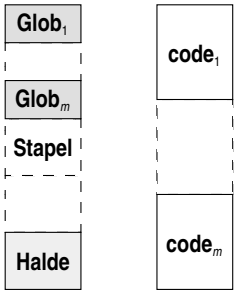
Code der Moduln wird statisch zusammengebunden
Globale Datenbereiche werden nacheinander im Speicher angelegt
(globale) Prozeduren und Variablen können absolut adressiert werden

Transformationen

- α ordnet Moduln den Anfang des Datenbereichs zu
- γ ordnet Moduln den Anfang des Prozedurcodes zu

$code_A(M.v)\,\alpha\gamma$ = **lda** $\alpha(M)+\alpha(v)$

$code_A(m.p)\,\alpha\gamma$ = **lda** $\gamma(m)+\alpha(p)$



Klassen und Objekte

Moduln können beliebig oft instanziiert werden

generic class *c* **imports** $i_1 \ldots i_k$
var v_1: ... v_i: ...
proc p_1 ...
...
proc p_m ...
end

```
o1 := new c;      -- Klasseninstanzen (Objekte)
o2 := new c;
```

"naive" Speicherorganisation

Objekte werden dynamisch auf der Halde angelegt
sie werden als Verweise auf das Produkt ihrer Eigenschaften dargestellt

Transformationen

wie bei Zeigern auf Verbunde

$code_A(o1.v)\,\alpha\gamma$ = **lda** $\alpha(o1)$ l - $\lambda(o1)$; **inc** $\alpha(v)$

$code_A(o2.p)\,\alpha\gamma$ = **lda** $\gamma(c) + \alpha(p)$; **cum**;

Bemerkung

die Darstellung ist *hochgradig redundant*,
weil alle Objekte Verweise auf die *gleichen* Methoden enthalten

Speicherorganisation für Klassen und Objekte

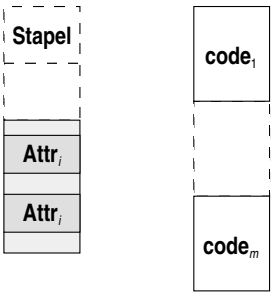
Verbesseung (bei statischer Bindung von Methoden an Objekte)
jedes Objekt erhält einen Verweis auf die *Methodentabelle* seiner Klasse
im Objekt selber werden nur die (dynamischen) Attribute abgespeichert

Transformation

$code_A(o1.v)\alpha\gamma$ = **lda** $\alpha(o1)$ l - $\lambda(o1)$; **ind**; **inc** $\alpha(v)$

$code_A(o2.p)\alpha\gamma$ = **lda** $\gamma(c)$ + $\alpha(p)$; **cum**;

Die Instruktion **cum** ruft die Methode auf ,
deren Adresse oben auf dem Stapel liegt



Einfachvererbung (mit statischer Bindung)

Klassen können Eigenschaften von einer Oberklasse erben
Methoden können in der Unterklasse *überschrieben* werden (*overriding*)
die geerbten Eigenschaften (Attribute und Methoden)
werden in die Objekte der Klasse kopiert (und ggf. überschrieben)

```
generic class s inherits
var v1: ... vi: ...
proc p1 ...
...
proc pn ...
generic class c inherits
var vl+1: ... vk| : ...
override proc pi ...
proc pn+1 ...
...
proc pm ...
end
o1 := new s;        -- Klasseninstanzen (Objekte)
o2 := new c;
```

Methoden können in der Unterklasse *überschrieben* werden (*overriding*)
die geerbten Eigenschaften (Attribute und Methoden)
werden in die Objekte der Klasse kopiert (und ggf. überschrieben)

Einfachvererbung (mit dynamischer Bindung)

Untertypen
Die Klasse *c* wird als Untertyp von *s* aufgefaßt
Objekte haben einen (statischen) prinzipiellen Typ (z. B. *s*)
und einen (dynamischen) aktuellen Typ (z. B. *c*)
c-Objekte können immer an *s*-Variablen zugewiesen werden

```
o1 := new s;        -- Klasseninstanzen (Objekte)
o2 := new c;
o1 := o2            -- statisch: s;  dynamisch:  c
```

umgekehrt geht das nicht (Weshalb?)

dynamische Bindung
der aktuelle Typ (d. h. die aktuelle Unterklasse) des Objektes bestimmt,
welche Methode an einem Namen gebunden wird

```
o1.m
```

ruft (in diesem Beispiel) `c.m` oder `s.m` auf

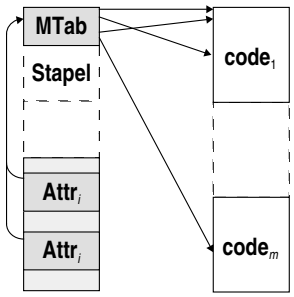
casting

```
o1 := (s)o2         -- statisch und dynamisch: s
```

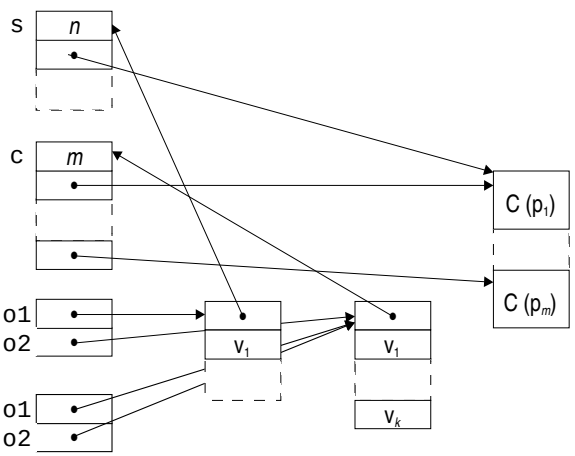
die Attribute werden kopiert , der Methodenzeiger bleibt

Implementierung der dynamischen Bindung

Methodentabellen bei Einfachvererbung
jede Klasse erhält eine *Methodentabelle*



Objekte verweisen auf die Methodentabelle ihres aktuellen Typs



dynamisches Laden von Klassen

wie in Java

die Methodentabellen werden über den Klassennamen erreicht (*hashing*)
die Methodentabelle und der Code werden als Halde verwaltet
(Speicherplatz-Freigabe)

Mehrfachvererbung

casting wird komplizierter

...

Synthese allgemein

Teilaufgaben

Zwischencode: