7. Kapitel

Abbildungsphase
Teil 1

Transformation

Kapitel 7: Transformation

0. Einbettung
1. Typabbildung
2. Speicherabbildung
3. Abbildung der Operatoren
4. Abbildung der Ablaufsteuerung
5. Speicherorganisation und Prozeduraufruf

7. Die Synthesephase

Aufgabe: attributierter Strukturbaum $\rightarrow$ ausführbarer Maschinencode

Problem:

- außer bei Codeerzeugung für die abstrakte Quellsprachenmaschine (QM), eine Kellermaschine, sind alle Aufgaben ,,guter'' Codeerzeugung NP-vollständig
  - Qualität also nur näherungsweise erreichbar

Zerlegung der Synthese:

- Abbildung, d.h. Transformation/Opimierung: Code für abstrakte Zielmaschine ZM (ohne Ressourcenbeschränkung) herstellen und optimieren, Repräsentation als Zwischensprache IL
- Codeerzeugung: Transformation $IL \rightarrow$ symbolischer Maschinencode
  - unter Beachtung Ressourcenbeschränkungen
- Assemblieren/Binden: symbolische Adressen auflösen, fehlende Teile ergänzen, binär codieren

7. Zwischensprache *IL*

2 Klassen von Zwischensprachen:

- Code für Kellermaschine mit Halde, z.B. Pascal-P, ..., JVM, CLR (.net)
  - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
  - Datentypen und Operationen auf Daten entsprechen weitgehend der QM, zusätzlich Umfang und Ausrichtung im Speicher berücksichtigen
- Code für RISC-Maschine mit unbeschränkter Registerzahl und (stückweise) linearem Speicher
  - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
  - Datentypen entsprechen Zielmaschine einschl. Umfang und Ausrichtung im Speicher
  - Operationen entsprechen Zielmaschine (Laufzeitsystem berücksichtigen!)
  - aber noch keine konkreten Befehle, keine Adressierungsmodi
  - Vorteil: fast alle Prozessoren auf dieser Ebene gleich

Kellermaschinencode gut für (Software-)Interpretation, schlecht für explizite Codeerzeugung, RISC-Maschine: umgekehrt

7. Zwischensprache *IL* II

Im folgenden nur Code für RISC-Maschine mit unbeschränkter Registerzahl betrachtet,

drei Darstellungsformen:

- keine explizite Darstellung: *IL* erscheint nur implizit bei direkter Codeerzeugung aus AST: höchstens lokale Optimierung, z.B. Einpaßübersetzung
- Tripel-/Quadrupelform: Befehle haben schematisch die Form
  - $t_1 := t_2 \tau t_3$ oder
  - $m: t_1 := t_2 \tau t_3$
    - analog auch für Sprünge
- SSA-Form (Einmalzuweisungen, *static single assignment*): wie Tripelform, aber an jedes t_i kann nur einmal zugewiesen werden (gut für Optimierung)

7. Zwischensprache *IL* III

Gesamtprogramm eingeteilt in Prozeduren,

Prozeduren unterteilt in Grundblöcke, oder erweiterte Grundblöcke

- Grundblock: Befehlsfolge maximaler Länge mit:
  wenn ein Befehl ausgeführt wird, dann alle genau einmal, also
  - Grundblock beginnt mit einer Sprungmarke,
  - enthält keine weiteren Sprungmarken
  - endet mit (bedingten) Sprüngen
  - enthält keine weiteren Sprünge
  - entspricht einem Block im Flußdiagramm (dort nicht maximal)
  - Unterprogrammaufrufe zählen nicht als Sprünge!
- erweiterter Grundblock: wie Grundblock, aber kann mehrere bedingte Sprünge enthalten: ein Eingang, mehrere Ausgänge

.NET: Goals

- language independence
- but platform dependent
  - the libraries reflect Windows functionality
- location independence: the target are web-services
- reflective programming
- versioning
- dynamic class loading, but exactly one class loader

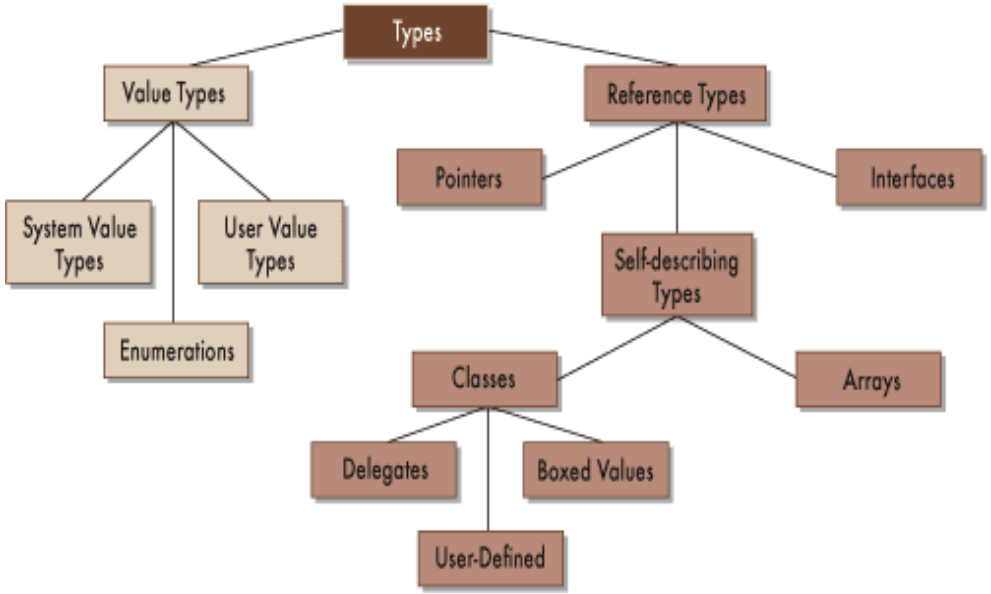
.NET – Literatur

W. Beer, D. Birngruber, H. Mössenböck, A. Wöß: Die .NET-Technologie. dpunkt.verlag 20
Michael Stal: C#- und .NET-Tutorial, iX 12/2001 - 02/2002.
J. Gough: Stacking Them Up: A Comparison of Virtual Machines. ACSAC 2001.
J. Gough: Compiling for the .NET Common Language Runtime (CLR). Prentice-Hall, 2002.
Mono Projekt, Open Source implementation of the .NET framework, .NET and C#, http:/

.NET: Language Independence

- make programming languages interoperable by translating them all into the same intermediate lang
- properties of CIL
  - common type system CTS
  - minimal type system CLS (common language specification)
    - to be supported by all languages
  - common language runtime system CLR
    - stack machine architecture, automatic garbage collection, versioning, no "DLL hell", securi
      - graphic user interfaces (Windows Forms)
      - web interfaces (Web Forms)
      - data base interface (ADO.NET)
      - container classes (Collections)
      - multithreading (Threads)
      - reflexion
      - …
- **but CIL is not directly executable, must be compiled to machine language!**

Common Type System (CTS) I



CTS:

- set of types that **may** be supported by any .NET language, not a minimal but a maxima
- all types consist of objects (no “primitive values types”)

distinguish

- value types: enumerated types, integers, floats, user defined struct types
- reference types: arrays, classes, delegates (sets of function pointers, pointers)
  - boxing: convert value type to reference type
  - unboxing: convert reference type to value type

Common Type System (CTS) II

Common Language Specification (CLS): minimal type system
the common denominator for interoperating languages

- subset of CTS, must be supported by all compilers if their language would like to in
- all exported types and interfaces must obey the 41 CLS rules, e.g.
  - rule 15: the types of array elements must conform to CLS. Arrays must have a fi

The Types of CTS

CIL	in system lib	explanation
bool	System.Boolean	1 Byte, false = 0, true $\neq 0$
char	System.Char	16 Bit Unicode character
string	System.String	Unicode string
object	System.Object	managed heap pointer
typedref	System.TypedReference	pair (pointer,type)
float32	System.Single	IEEE float. pt. 32 Bit
float64	System.Double	IEEE float. pt. 64 Bit
int8	System.SByte	2 complement inter 8 Bit
int16	System.SByte	2 complement inter 8 Bit
int32	System.Int32	2 complement inter 32 Bit
int64	System.Int64	2 complement inter 64 Bit
unsigned int8	System.Byte	unsigned integer 8 Bit
unsigned int16	System.Uint16	unsigned integer 16 Bit
unsigned int32	System.Uint32	unsigned integer 32 Bit
unsigned int64	System.Uint64	unsigned integer 64 Bit
native int	System.IntPtr	signed mach. dep. int
native unsigned int	System.UIntPtr	unsigned mach. dep. int

CLR: Common Language Runtime

CLR
is the runtime environment for .NET-applications
uses abstract stack machine

- that only processes instructions of the intermediate code CIL

CIL Example

```
public static Point operator+(Point op1, Point op2) {
      return new Point(op1.x+op2.x,op1.y+op2.y);
  }
.method public hidebysig specialname static
    valuetype ComplexNumbers.Point  op_Addition(valuetype ComplexNumbers.Point op1,
                              valuetype ComplexNumbers.Point op2) cil managed
{  // Code size       40 (0x28)
  .maxstack  4
  .locals ([0] valuetype ComplexNumbers.Point CS$00000003$00000000)
  IL_0000:  ldarga.s   op1
  IL_0002:  call       instance float64 ComplexNumbers.Point::get_x()
  IL_0007:  ldarga.s   op2
  IL_0009:  call       instance float64 ComplexNumbers.Point::get_x()
  IL_000e:  add
  IL_000f:  ldarga.s   op1
  IL_0011:  call       instance float64 ComplexNumbers.Point::get_y()
  IL_0016:  ldarga.s   op2
  IL_0018:  call       instance float64 ComplexNumbers.Point::get_y()
  IL_001d:  add
  IL_001e:  newobj     instance void ComplexNumbers.Point::.ctor(float64,float64)
  IL_0023:  stloc.0
  IL_0024:  br.s       IL_0026
  IL_0026:  ldloc.0
  IL_0027:  ret
} // end of method Point::op_Addition
```

Comparable Java Bytecode

```
public static Point plus(Point p1, Point p2) {
  return new Point(p1.getX() + p2.getX(), p1.getY() + p2.getY());
}
```

```
public static Point plus(Point arg0, Point arg1)
Code(max_stack = 8, max_locals = 2, code_length = 26)
0:    new               <Point> (9)
3:    dup
4:    aload_0
5:    invokevirtual     Point.getX ()D (10)
8:    aload_1
9:    invokevirtual     Point.getX ()D (10)
12:   dadd
13:   aload_0
14:   invokevirtual     Point.getY ()D (11)
17:   aload_1
18:   invokevirtual     Point.getY ()D (11)
21:   dadd
22:   invokespecial     Point.<init> (DD)V (12)
25:   areturn
```

note: Java bytecode has typed operators (dadd), hence it can be interpreted

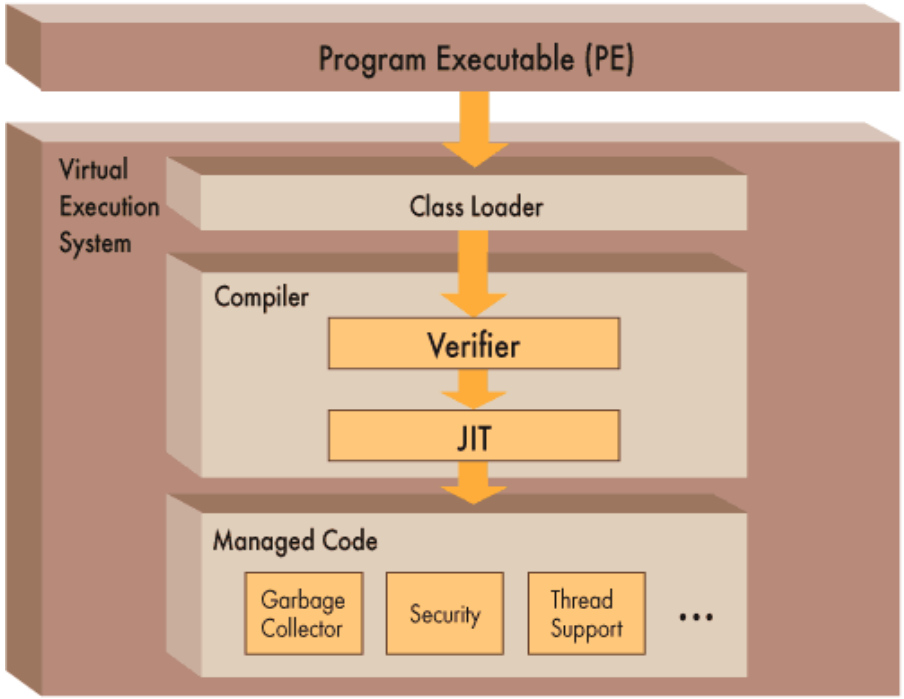
CIL operators are typefree (add), hence it must be compiled before execution

.NET-runtime system CLR

CIL-Code is not interpreted but compiled to machine code by the VES (virtual execution sys

CLR contains

- just-in-time compiler (JIT)
- garbage collector
- class loader (for loading assemblies)
- code inspector (verifier): checking type safety of CIL programs



Managed Code and Data

managed code: execution under supervision of CLR
unmanaged code: other code

- interoperability between managed and unmanaged code:
  - link existing DLLs with P/Invoke (platform invoke)
  - COM/COM+-linking via COM-Interop-mapping

managed data: data under supervision of the garbage collector (not persistent)
unmanaged data (unsafe code): data manually allocated and deallocated by the programm

- managed code may contain managed but also unmanaged data

Code Checking

problem: unsafe code, especially pointer arithemtic

- code checking by verifier and class loader:
  - verifier is conservatively checking that the program is obeying all rules about st
    - program only accesses storage allocated for it
    - program accesses objects only via their interfaces
  - class loader checks that all initial values are of admissible type.

verifier may be shut down for dealing with arbitrary pointer arithemtic and function poi

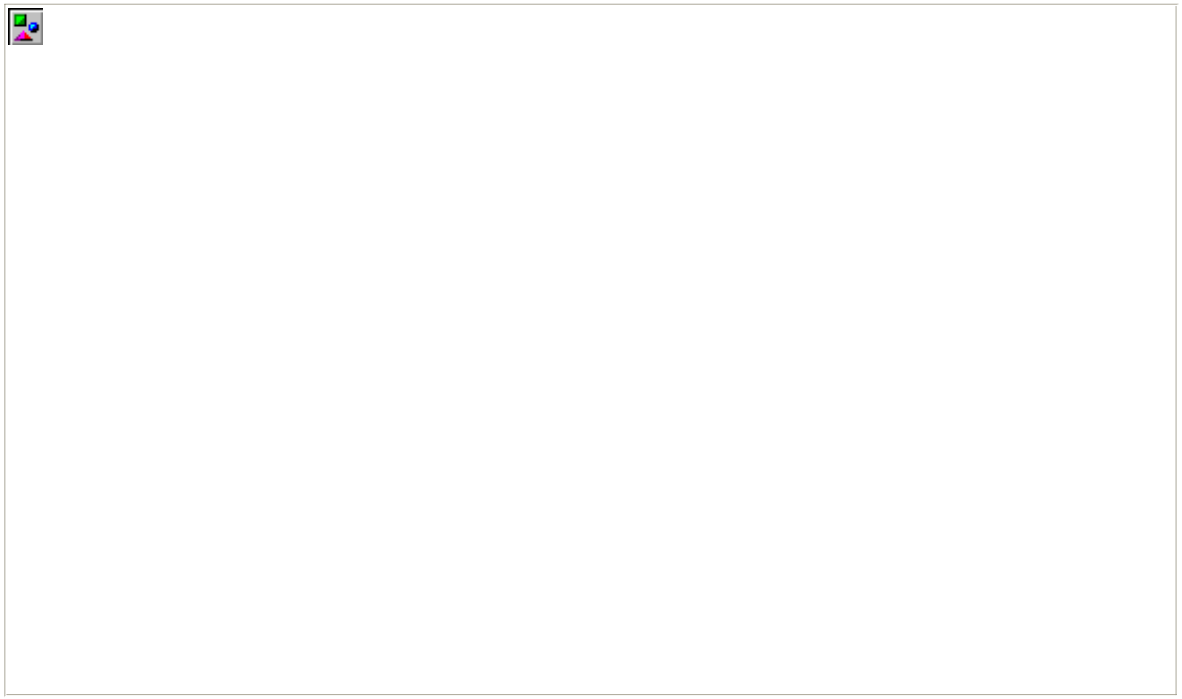
- ANSI C and multiple inheritance can be supported

Assemblies

assembly: .NET-component

- logical unit similar to DLL or EXE
- elementary unit with respect to installation, configuration, security, safety, version

during loading the VES activates a runtime host for supervising the compilation and exec



Assemblies: the Content

an assembly contains

- one or more modules (program executables, PE): executable code for the types wit
- version number (strong name) specified by **`.assembly`**:
  - name
  - version: (Major.Minor.Build.Revision)
  - language (en-US, de-DE, ...)
  - public key of producer
- manifest information, belonging to a module or globally to an assembly:
  - descrition of the modules and ressources of the assembly
  - import interface: informs about references to imported assemblies
  - export interface: exported types and ressources
- specification of the CLR version to be used for execution
  - useful but dangerous: a MS compiler could specify that a given assembly ma

Assemblies: reflection and administration (versioning)

assembly contains explicit meta data (generated by compiler front-end):

- definition tables: description of the elements contained in the module
- reference tables: description of foreign elements, not defined but used in a module
- manifest (table)

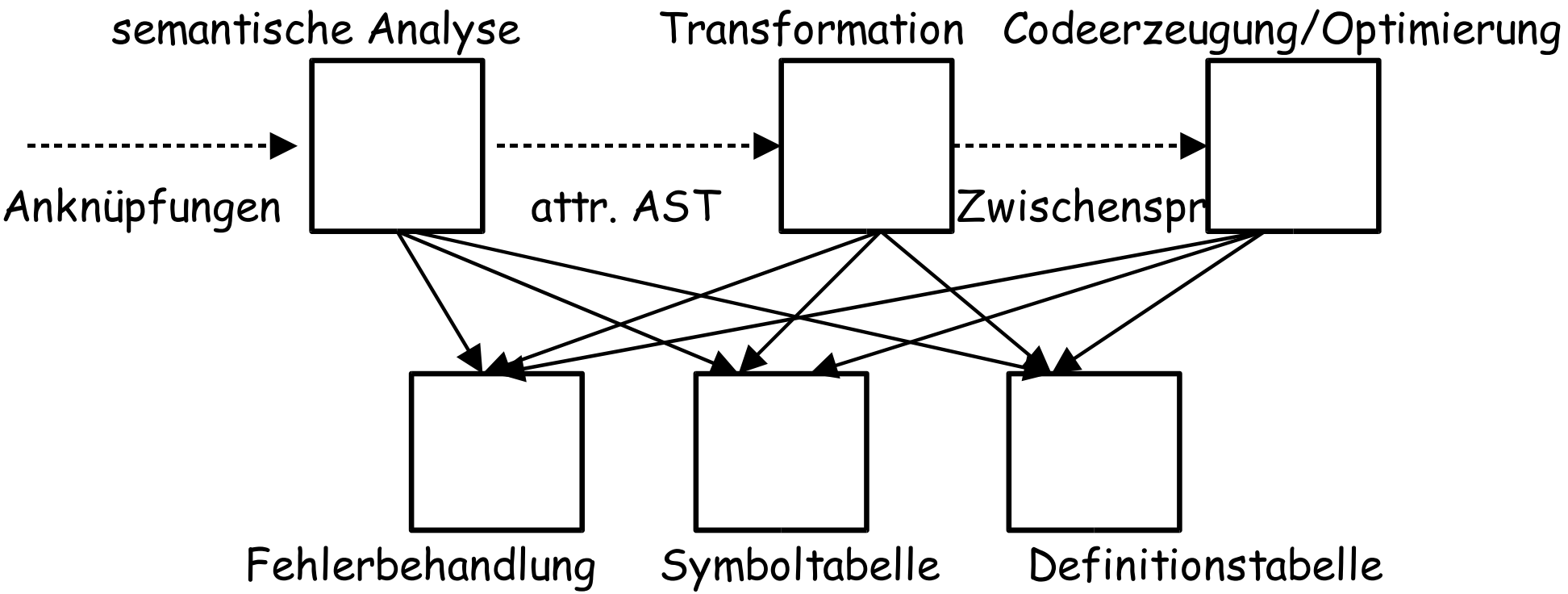
dynamic queries with reflection mechanisms possible

- meta data + reflection imply
  - no IDL necessary
  - no registry for assemblies, assemblies may be anywhere in the file system
    - private assemblies in local directories of the application
    - common assemblies mostly in global directories (global assembly cache)

installation requires signature with private key

- avoid naming conflicts by help of strong names

7. Transformationsphase



7. Transformationsphase - Aufgaben

Definition der abstrakten Zielmaschine

(Speicherlayout, Befehlssatz (Laufzeitsystem)), dann:

- Typabbildung,
- Operatorabbildung,
- Ablaufabbildung.

7. Abstraktion der Zwischensprache

Problem Abstraktionsniveau:

Übersetzung vs. Laufzeitsystem

Portabilität des Übersetzers vs. Effizienz der übersetzten Programme

Beispiele:

E/A-Routinen gewöhnlich im Laufzeitsystem,

Indexrechnung wird vollständig übersetzt (?),

Prozeduraufrufe werden gewöhnlich auf parameterlose Prozedurrufe reduziert,

Speicherzuteilung und Speicherbereinigung gewöhnlich im Laufzeitsystem,

Ausnahmebehandlung mit Unterstützung des Laufzeitsystems

Kapitel 7: Transformation

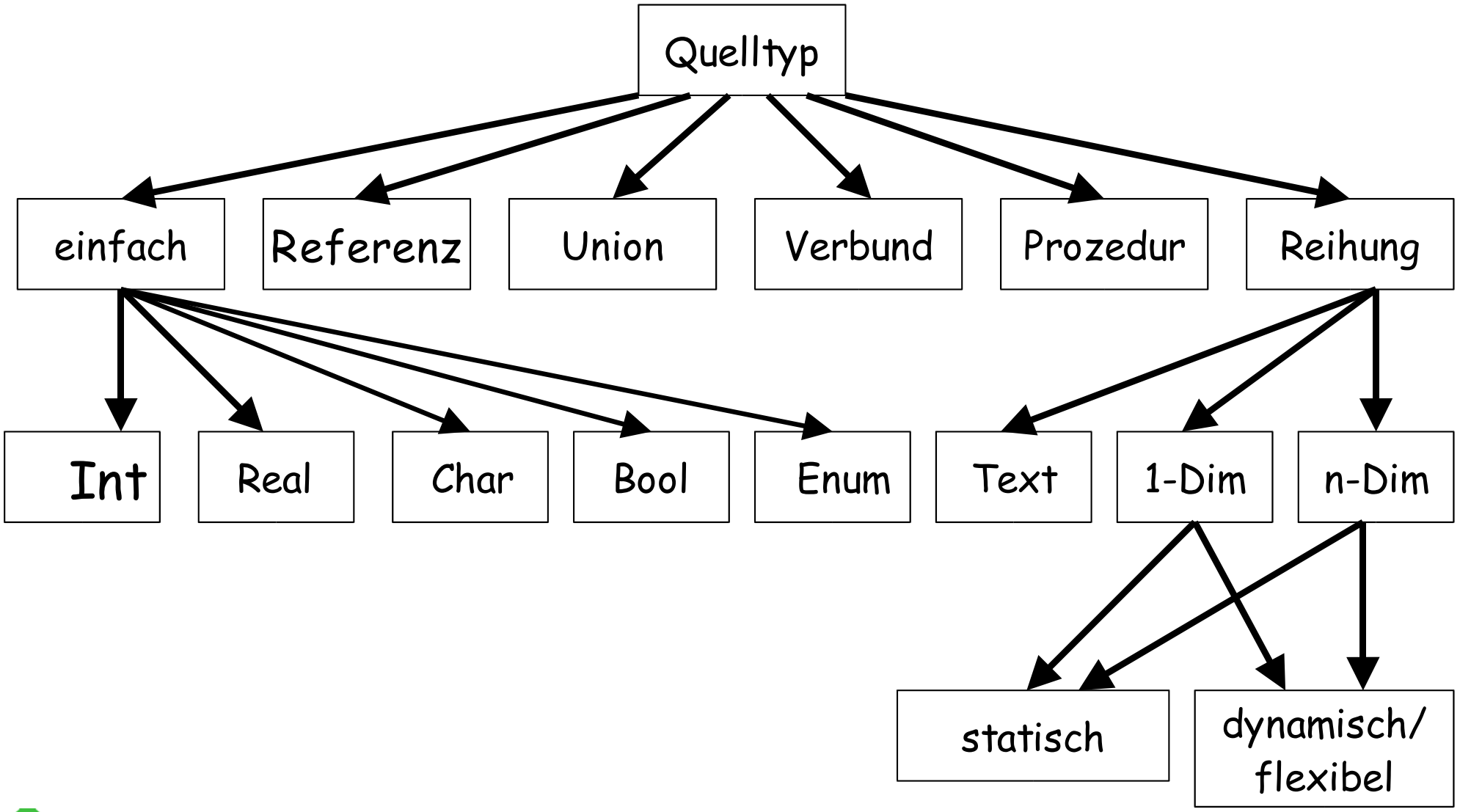
0. Einbettung
1. Typabbildung
2. Speicherabbildung
3. Abbildung der Operatoren
4. Abbildung der Ablaufsteuerung
5. Speicherorganisation und Prozeduraufruf

7.1 Typabbildung

Datentypen sind

- Einfach:
  - Aufzählungstyp, Referenz(!), `bool`, `char`, `int`, `unsigned`, `flt`, ...
  - Aufgabe: Quellsprachentypen auf adressierbaren Speicherbereich mit Ausrichtung abbilden, Minimalgröße gewöhnlich ein Byte
- zusammengesetzt:
  - Reihungen: unterscheide
    - Reihung statisch fester Länge (statische R.)
    - Reihungslänge bei Vereinbarung fest (dynamische R.)
    - Reihungslänge durch Zuweisung änderbar (flexible R.)
    - Texte (`array[*](char)`)
  - Verbunde
  - Vereinigungstypen, einschl. Verbunde mit Varianten
  - gepackte zusammengesetzte Typen, einschl. BCD-Zahlen
  - oo-Objekte, Schachteln im Keller wie Verbunde behandeln!

7.1 Klassifikation von Typen



7.1 Einfache Datentypen

Unterscheide Abbildung auf 1,8,16,32,64 Bit (80-Bit Gleitpunktzahlen?), Ausrichtung

Aufzählungstyp: Werte durch ganze Zahlen codieren

- bei Bool
  - Festlegung der Codierung für true, false
    1. 0 false, 1 true
    2. 0 false, ≠ 0 true

       C Interpretation von `int` in if-Anweisungen
    3. 0 true, ≠ 0 false

       C exit code für Programmterminierung
  - vermeide Abbildung auf 1 Bit (auch Bool mindestens 8 Bit)
- bei char
  - Festlegung der Codierung: ASCII, ISO 8859-*, EBCDIC, Unicode

7.1 Allgemeines zur Typcodierung

- bei allen Typen: auf Kompatibilität mit Betriebssystem achten, wegen Systemaufrufen
  - daher gewöhnlich die Konventionen des C-Übersetzers nutzen, mit dem das BS geschrieben ist
- bei Ausrichtung auf die Geschwindigkeit der Speicherlogik achten
- *big /little endian* beachten (erstes Byte höchst-/geringst-wertig)

7.1 Reihungen

Festlegung zeilenweise/spaltenweise Speicherung

Zerlegung in Deskriptor und Datensatz

- Deskriptor: enthält alle Info für
  - Speicherabbildungsfunktion $\text{adr}(a[i_1,...,i_n]) = \text{adr}(a[0,...,0]) + (...(i_1*(og_1-ug_1+1) + i_2)*(og_2-ug_2+1) + ... i_{n-1}*(og_{n-1}- ug_{n-1}) + i_n)*d$
  - Test der Grenzen
- Deskriptor hat feste Länge
- Deskriptor und Datensatz getrennt im Speicher (außer eventuell bei statischen Reihungen)
  - Abbildung also auf zwei Speicherobjekte
  - $\text{adr}(a[0,...,0])$ heißt virtuelle Anfangsadresse

Untergrenze ug_1
Obergrenze og_1
...
Untergrenze ug_n
Obergrenze og_n
El.umfang d
$\text{adr}(a[0,...,0])$

7.1 Texte

Eigentlich eindimensionale Reihung von Zeichen

Sonderbehandlung:

- C Konvention: Abschluß mit \0
- Sonst: Deskriptor wie bei Reihungen (speichert Länge)
- Wegen Betriebssystemrufen (C Funktionen) oft beides
- Ausrichtung wie Zeichen

Problem Unicode:

- Bei UTF-8: Länge erforderlich, da nicht aus Anzahl Bytes herleitbar
- Bei UTF-16: Länge = Anzahl Bytes / 2

7.1 Referenzen

- Wie elementare Typen behandeln
- Länge der Speicherobjekte definiert maximale Größe des Adreßraums
- 16-bit, 32-bit, 64-bit Referenzen?

7.1 Verbunde

- Folge (oder Menge ?) von Elementen
- Ausrichtung ist maximale Ausrichtung der Elemente
- Länge ist Summe der Länge der Elemente plus Verschnitt wegen Ausrichtung
- variante Verbunde wie Vereinigungstypen behandeln

7.1 Objekte

allgemein: Objekte wie Verbunde behandeln

- wegen Polymorphie ist zusätzlich Typkennung erforderlich
  - entweder bei Referenz (weniger empfehlenswert)
  - oder als nulltes Attribut (Objekt mit selbstidentifizierendem Typ)
- Folge (oder Menge ?) von Attributen?
- bei Objekten mit Oberklassen: ein oder mehrere Objekte (mit Rückverweis?)
- Methoden im Objekt oder in der Klasse vermerken (Sprungleiste)?
  - oft benutzt, eigentlich nicht nötig
- Sonderaufgabe: Berechnung der Reihenfolge von Initialisierungen (einschl. gemeinsamer Klassenattribute)

7.1 Prozedurtypen

- Referenz auf Prozedur: behandeln wie Referenzen
- gebundene Prozedur (*closure*): Verbund aus Prozedurreferenz und Umgebungszeiger
- gebundene Prozedur mit gebundenen Parametern: gebundene Parameter wie zusätzlichen Verbund behandeln
  - Vorsicht, wenn gebundene Parameter per Referenzaufruf übergeben werden oder Prozeduren sind: Dann muß der Kontext der Bindung beim späteren Aufruf noch verfügbar sein, er muß zurückgehalten werden, selbst, wenn es sich um eine Schachtel auf dem Keller handelt, siehe später ,,Rückhaltestrategie''

7.1 Vereinigungstypen (*union*)

Vereinigungstypen, variante Verbunde, polymorphe Typen

- In typsicheren Sprachen: Variante wird als dynamischer Typ gemerkt: Typkennung für dynamischen Typ
- Länge ist definiert durch längstes Element
- Achtung Rekursion: Elemente können wieder Vereinigungstypen sein. Bei Abbildung beachten (siehe *ADT* Speicherabbildung: `growth_points` ist Keller)
- Vorsicht: die Typkennung ist nicht universell, sondern abhängig vom Übersetzerlauf

7.1 Gebrauch Quelltypen

in der semantischen Analyse:

- Test auf Wohlgeformtheit des Programms (ist abgeschlossen)
- Operatoridentifikation (ist abgeschlossen)

in der Transformationsphase

- Speicherabbildung
- induzierte Typanpassungen

in der Optimierung:

- Unterscheidung von Werten mit gleicher Speicherdarstellung
- Alias-Analyse

zur Laufzeit:

- Typkennungen bei Vereinigungstypen, polymorphen Typen, ...
- Speicherdarstellung aus Speicherabbildung muß für Testhilfen und Speicherbereinigung bekannt sein

Kapitel 7: Transformation

0. Einbettung
1. Typabbildung
2. Speicherabbildung
3. Abbildung der Operatoren
4. Abbildung der Ablaufsteuerung
5. Speicherorganisation und Prozeduraufruf

7.2 Berechnung Relativadressen in Schachteln, Verbunden, ...

ADT Speicherabbildung, Grundideen:

- bei Schachtelung ist Umfang/Ausrichtung eines Objekts q alloziert in Objekt p stets bei Allokation in p bekannt, daher:
  1. öffne ein oder mehrere zunächst leere Gebiete
  2. füge zu offenem Gebiet Objekte bekannten Umfangs/Ausrichtung hinzu und liefere Relativadresse zurück
  3. schließe Gebiet, Gebiet wird Block b festen Umfangs und Ausrichtung, das ein Objekt repräsentiert, b kann nun einem anderen Gebiet zugefügt werden

7.2 Berechnung Relativadressen in Schachteln, Verbunden, ...

Erweiterung bei Überlagerung von Alternativen, z.B. Pascal-Verbunde mit Varianten (*mark-dispose*-Verfahren):

- offene Gebiete besitzen Keller von Marken (= Relativadressen)

  2a. bei Beginn erste Alternative: anfängliche Relativadresse und Alternativenlänge 0 in den Keller

  2b. bei Ende Alternative: Maximum der Alternativenlängen ersetzt bisherige Alternativenlänge im Keller, wieder bei anfänglicher Relativadresse beginnen

  2c. bei Ende letzte Alternative: anfängliche Relativadresse um maximale Alternativlänge erhöhen

7.2 *ADT* zur Speicherabbildung

```
class Area {
private ListofAreas blocks;   // contained blocks
private int offset;           // offset in the block | ´m contained in
private int current_offset;   // offset of the next block to add
private int align;
private Direction d;          // up or down
private Strategy s;           // aligned or packed
private StackofInt growth_points;
private StackofInt maximum_offset;

Area ( Direction d; Strategy s; int align ){
this.d = d;
this.S = s;
this.align = align;
this.current_offset =0;
this.current_growth_point = 0;
this.blocks = new ListofAreas();
this.growth_points = new StackofInt(); // stack of marked offsets
this.maximum_offset = new StackofInt();
this.maximum_offset.push(0);}
...
```

7.2 Fortsetzung

```
...
public void add_block (Area block ) {
   int maximum;
   block.offset = current_offset;
   current_offset = f(current_offset, block.size, align, s, d);
   maximum = maximum_offset. pop();
   maximum_offset.push(current_offset > maximum?
      current_offset:maximum);
   align = g(block.align, align, s);
   blocks.append(block);
}
public void mark (){
   growth_points.push(current_offset);
   maximum_offset.push(0);
}
public void back () {
   current_offset = growth_points.top();
}
public void combine () {
   current_offset = maximum_offset.pop();
   growth_points. pop ();
}
```

7.2 zum Algorithmus Speicherabbildung

Idee des Verfahrens auch für andere Zwecke einsetzbar, z.B.

- Konstruktion der Grundblöcke in der Zwischensprache:
  1. Gebiete für Grundblöcke öffnen
  2. erzeugte Befehle auf offene Gebiete verteilen
  3. nach Zufügen des abschließenden Sprungs Gebiet schließen

Algorithmus unterstellt, daß alle allozierten Objekte auch tatsächlich benötigt werden

- das ist bei Prozedurschachteln nicht sicher
  - z.B. nicht benutzte lokale Variable
  - z.B. Variable, denen kein Speicher, sondern nur Register zugeordnet werden
- Lösung: bei der Speicherabbildung nur symbolische Relativadressen vergeben, numerische Werte erst nach Optimierung und Registerzuteilung bestimmen

Kapitel 7: Transformation

0. Einbettung
1. Typabbildung
2. Speicherabbildung
3. Abbildung der Operatoren
4. Abbildung der Ablaufsteuerung
5. Speicherorganisation und Prozeduraufruf

7.3 Abbildung der Operationen

Prinzip: jede Maschinenoperation hat nur ein Ergebnis

- Arithmetische Operationen: 1-1 Zuordnung entsprechend Speicherabbildung
  Vorsicht: Prüfung Überlauf bei ganzzahliger Multiplikation erforderlich! meist sehr aufwendig!
- Maschinenoperationen mit mehreren Ergebnissen:
  - Operation mit anschließender Projektion, z.B. **divmod**
  - Operationen, die zusätzlich Bedingungsanzeige setzen:
    Zusätzlicher cmp-Befehl, falls Bedingung benötigt
- logische Operationen und Relationen: Unterscheide, ob Ergebnis zur Verzweigung benutzt oder abgespeichert wird
- Speicher-Zugriffe: Zugriffspfad explizit mit Adreßarithmetik codieren, Basisadressen sind Operanden, woher bekannt? (Konventionen festlegen)
  Achtung: Indexrechnung ganzzahlig, Adreßarithmetik vorzeichenlos!
- Typanpassungen: Dereferenzieren, deprozedurieren, Anpassung `int` $\rightarrow$ `flt` usw. explizit, Uminterpretation Bitmuster implizit

7.3 Zuweisung $x := a$

Wert in Register laden:

```
LD >Adresse a<  <Register>
```

rechte Seite unter Adresse abspeichern:

```
ST >Register< <Adresse x>
```

Adresse ist `Basisregister(+ Offset)`

- Informationen aus Typabbildung berechenbar
- kann durch Optimierung verändert werden, daher vorläufig nur symbolisch

7.4 Abbildung Ablaufsteuerung

kann als Quell-Quelltransformation beschrieben werden (unter Einsatz von Sprüngen)

- aber tatsächlich Erzeugungsverfahren für Einzelbefehle und Grundblöcke der Zwischensprache

Einzelfälle:

- Sprung
- bedingte Anweisung
- Fallunterscheidung
- Typ-Fallunterscheidung
- Schleife
- Zählschleife
- Prozeduraufruf
- Ausnahmebehandlung

7.4 Sprunganweisung

goto M;

```
JMP M
```

beendet Grundblock

7.4 Bedingte Anweisung

if B then S else S´ end;

```
B
JMP ZERO MS´
S
JMP Ende
MS´:
S´
Ende:
...
```

Beachte: Sprünge mit erfüllter Bedingung oft schneller!

3 Grundblöcke: `B; JMP ZERO MS´`, `S; JMP Ende`, `S´`

7.4 Fallunterscheidung

case expr
    when x_1 S´;
    when x_2 S´´; . . .
    default S;

Einfache Übersetzung-Kaskade von bedingten Anweisungen

```
e:=expr
if e = x_1 then S´ else
 if e = x_2 then S´´ else
  if . . .
  else S
   . . .
  end;
 end;
end;
```

- $2n+1$ Grundblöcke, beginnend mit
- `e:=expr; e = x_1; JMP NOTEQUAL MS´´`

7.4 Fallunterscheidung mit Sprungleiste

Abbildung von x_1 x_2 . . . in die ganzen Zahlen muß eindeutig sein.

```
JUMP IND Sprungtabelle + expr
Sprungtabelle:
M1, Sonst, Sonst, Sonst, M2, Sonst, . . .
M1:
S´
JMP Ende
M2:
S´´
JMP Ende
. . .
Sonst:
S
Ende:
. . .
```

Problem bei großen Lücken in der Tabelle

- $n+3$ Grundblöcke, einschl. Sprungtabelle (Sonderfall)

7.4 Typ-Fallunterscheidung

Fallunterscheidung über dem Eintrag, der den dynamischen Typ eines Objekts kennzeichnet.

Bsp. Ada, Sather(-K).

Implementierung polymorpher Aufrufe/Objektzugriffe erzeugt Typ-Fallunterscheidung implizit

Behandlung wie gewöhnliche Fallunterscheidung

Vorsicht mit Sprachen, bei denen die Typkennung nicht gespeichert wird - sie sind nicht typsicher!!

- Bsp. Variante Verbunde in Pascal, erzeugte Variante wird nicht gemerkt.

7.4 Anfangsgesteuerte Schleife

while B loop S end;

```
Anfang:
B
JMP ZERO Ende
S
JMP Anfang
Ende:
...
```

```
            JMP Anfang
 weiter: S
Anfang: B
            JMP NONZERO weiter
    ...
```

- 2 Grundblöcke, Fassungen unterscheiden sich in Anzahl ausgeführter Sprungbefehle (Anz. Sprünge im Code gleich), Anordnung rechts oft günstiger
- aber: Grundblöcke beliebig im Code plazierbar, dann beide Fassungen äquivalent

7.4 Endegesteuerte Schleife

loop S until B end;

```
Anfang:
   S
   B
   JMP ZERO Anfang
   ...
```

ein Grundblock

7.4 Zentralgesteuerte Schleife

loop S; exit when B; S´; exit when B´; S´´...end;

```
Anfang:
   S
   B
   JMP NON ZERO Ende
   S´
   B´
   JMP NON ZERO Ende
   S´´
   ...
   JMP Anfang
Ende:
   ...
```

- n+1 Grundblöcke

7.4 Zählschleife

for *i* := *a* step *s* until *e* do *S*

Annahme: Schrittweite *s* (samt Vorzeichen) statisch bekannt

Standardübersetzung (entspricht C, C++, ...):

```
i := a;
while i <= e      -- bei s < 0: i >= e
loop S; i := i+s end;
```

Alternative:

Ziel: _o ist der letzte Wert <= e, für den die Schleife ausgeführt wird, bei *s*=1 sinnvoll, da keine Div- oder Modulo-Operation und e = _o

```
if a <= e then
   i:=a; _x:=a mod s; _y:=e mod s;
   _o:=(_y >= _x)? e-(_y-_x) : e-(_x+s-_y);
        loop S;
           if i = _o then break else i := i+s end;
        end;
end
```

Alternative funktioniert immer! Auch bei *e = maxint* ! Aber Vorsicht, wenn <= in Vergleich auf 0 übersetzt werden muß

7.4 Ausnahme behandeln oder ignorieren

Sprachen wie Fortran erlauben beides

Zwei Rücksprungadressen: Adresse a für Ausnahme, Adresse a+Sprungbefehlsgröße für normales Ende

Ignorieren der Ausnahme: Sprung in Ausnahmebehandlung wird durch nop ersetzt

7.4 Beispiel

```
c=0;
                s1: ST    >c<   0
                s2: LD    <x>
if x > 0 {      s3: GT    0
                s4: JMP FALSE u1

a=2;            t1: ST    >a<   2
                t2: LD    <a>
b=a*x+1;        t3: LD    <x>
                t4: MUL   t2    t3
                t5: ADD   t4    1
                t6: ST    >b<   t5
a=2*x;          t7: LD    <x>
                t8: MUL    2    t7
                t9: ST    >a<   t8
```

```
c=a+1+b;        t10: LD   <a>
                t11: ADD t10    1
                t12: LD   <b>
                t13: ADD t11   t12
                t14: ST   >c<  t13
}               t15: JMP u1

x=c;            u1: LD    <c>
                u2: ST     x    u1
```

Kapitel 7: Transformation

0. Einbettung
1. Typabbildung
2. Speicherabbildung
3. Abbildung der Operatoren
4. Abbildung der Ablaufsteuerung
5. Speicherorganisation und Prozeduraufruf

7.5 Speicherorganisation und Prozeduraufruf

Aufgaben:

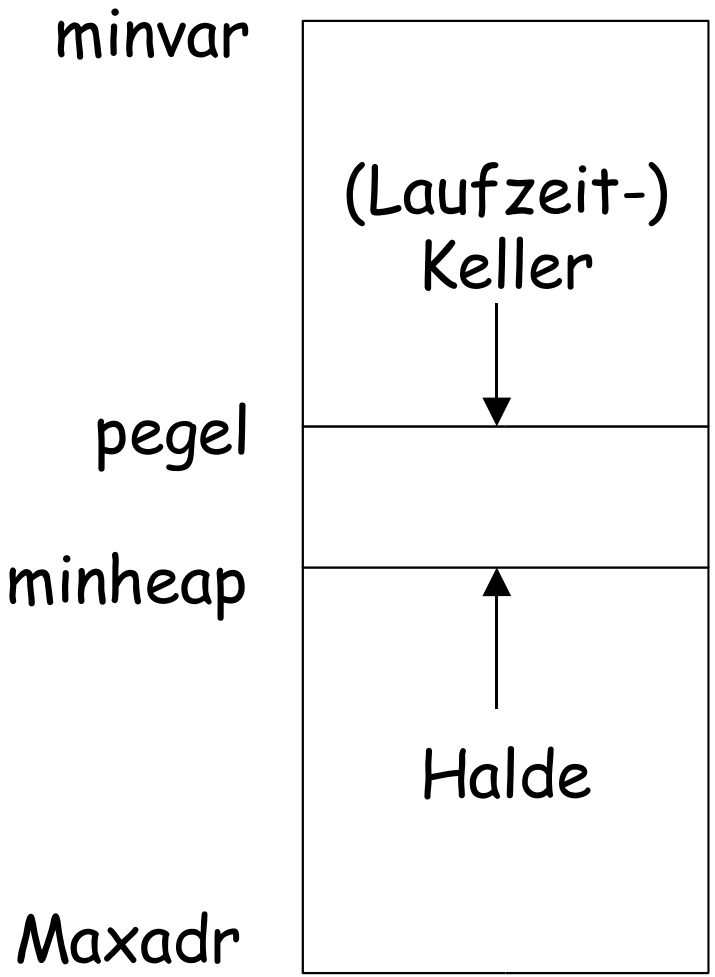
- alle Programmvariablen allozieren
- statische/dynamische/anonyme Allozierung unterscheiden
- Vorbereitung auf dynamische Prozedurschachtelung einschl. Rekursion

Verfahren:

- Unterscheide (Laufzeit-)Keller für dyn. Variable und Halde für Variable mit unbekannter Lebensdauer (anonyme Objekte)
- Keller eingeteilt in Schachteln (*activation record*)
  - unterste Schachtel für statische Variable
- Schachtel enthält Prozedurparameter, lokale Variable, Rückkehradresse, Verweis stat./dyn. Vorgänger, sonstige organisatorische Info, Hilfsvariable für Zwischenergebnisse
- Schachtel besteht aus statischem Teil (Länge dem Übersetzer bekannt) und dynamischem Teil (für dynamische Reihungen)

Erweiterung für mehrfädige Programme: mehrere Keller, Kaktuskeller

7.5 Speicherorganisation



Einteilung in zwei Speicherbereiche

Anordnung hardware-abhängig

garantiere Invariante:

minheap > pegel

7.5 Basisadressen

minvar Basis statischer Variabler
(Beginn Keller)

schachtel Basis lokaler Variabler eines Unterprogramms
(Beginn UP-Schachtel)

pegel Kellerpegel

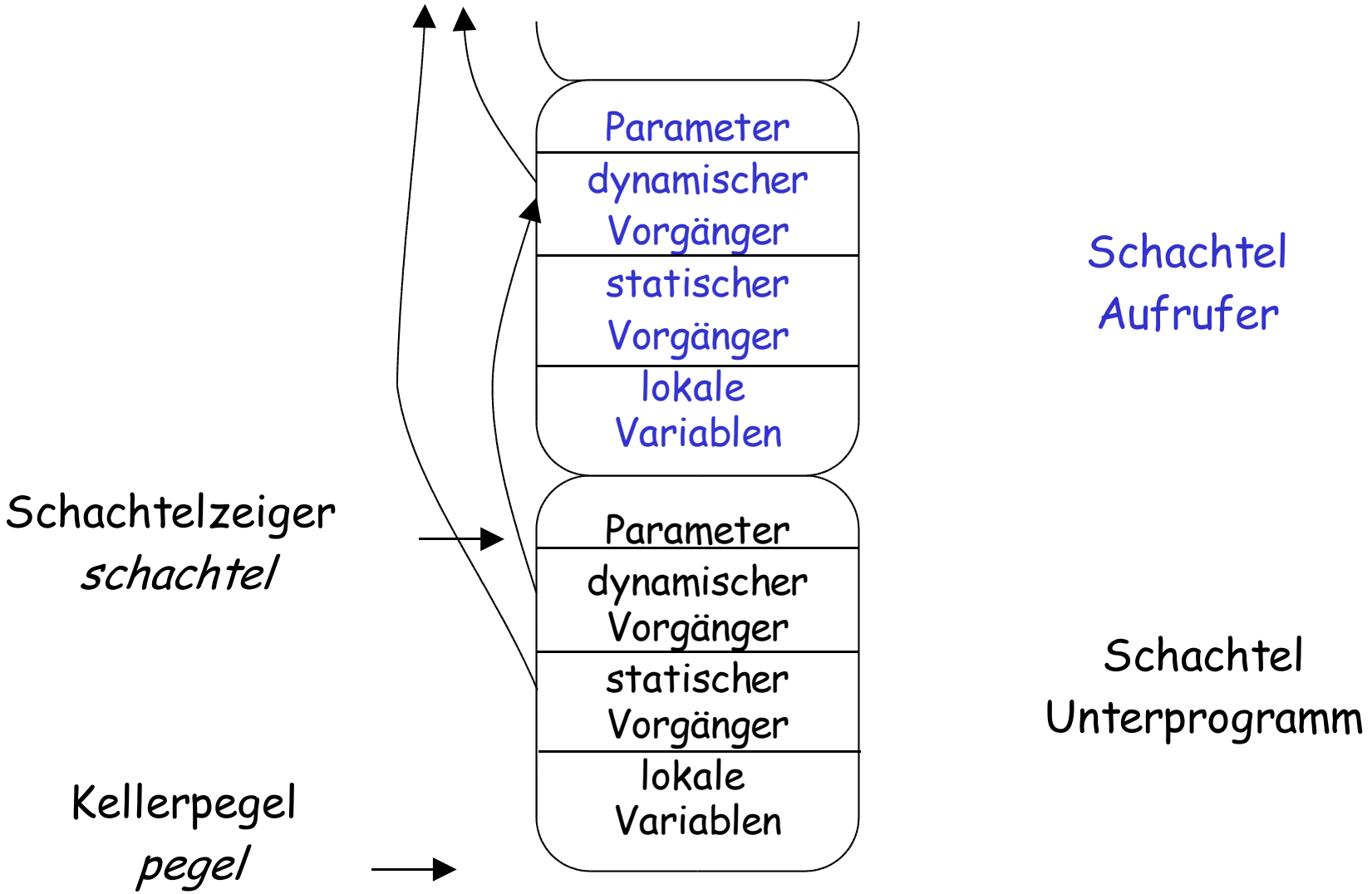
Adressen statischer Variabler v mit Rel.Adr. r_v:

$minvar + r_v$

Adressen dynamischer Variabler v mit Rel.Adr. r_v:

$schachtel + r_v$

7.5 Laufzeitkeller



7.5 Schachtel

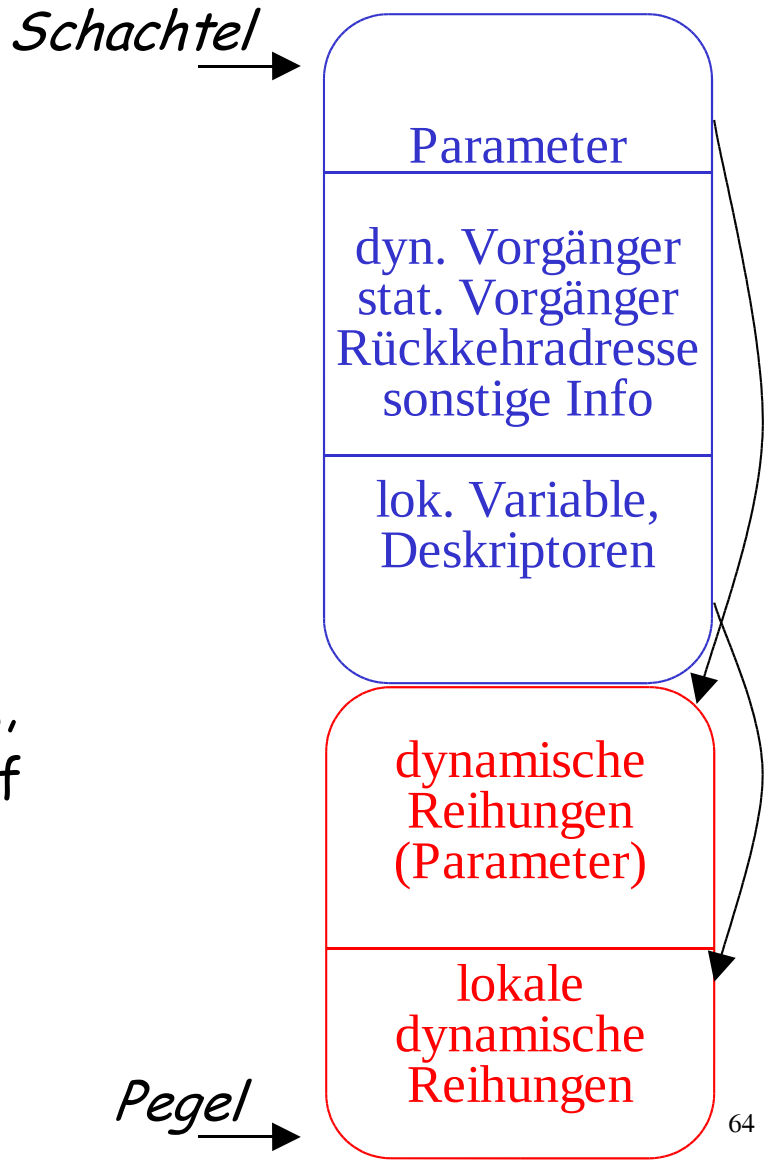
Einteilung statisch/dynamisch

Reihungsdeskriptoren im statischen Teil

Problem: (statischen) Teil der Schachtel sollte der Aufrufer reservieren

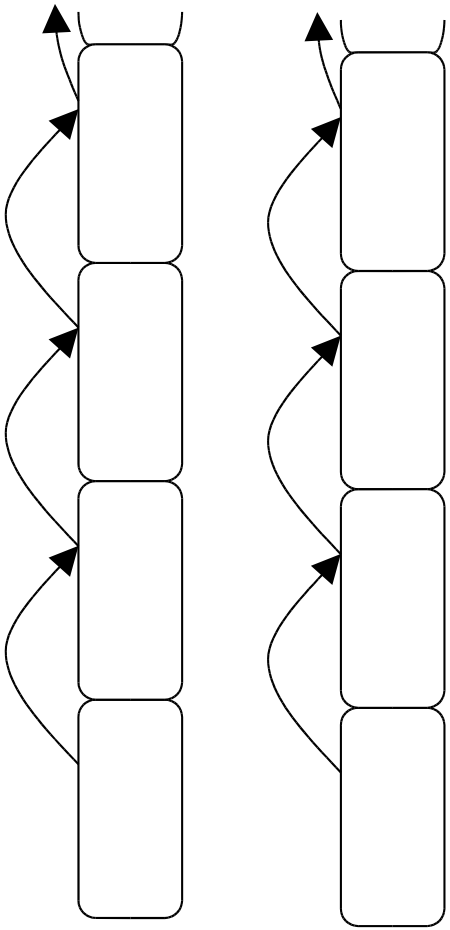
- dazu: Länge muß bekannt sein
- Länge nicht bekannt bei indirektem Aufruf (Prozedurvariable, formale Prozedur, Polymorphie)
- Lösung: 2 unterschiedliche Aufrufformen, 2 Eingänge für direkten/indirekten Aufruf

sonstige organisatorische Info:
Kennzeichnung der Prozedur (für Testhilfe/Speicherbereiniger)

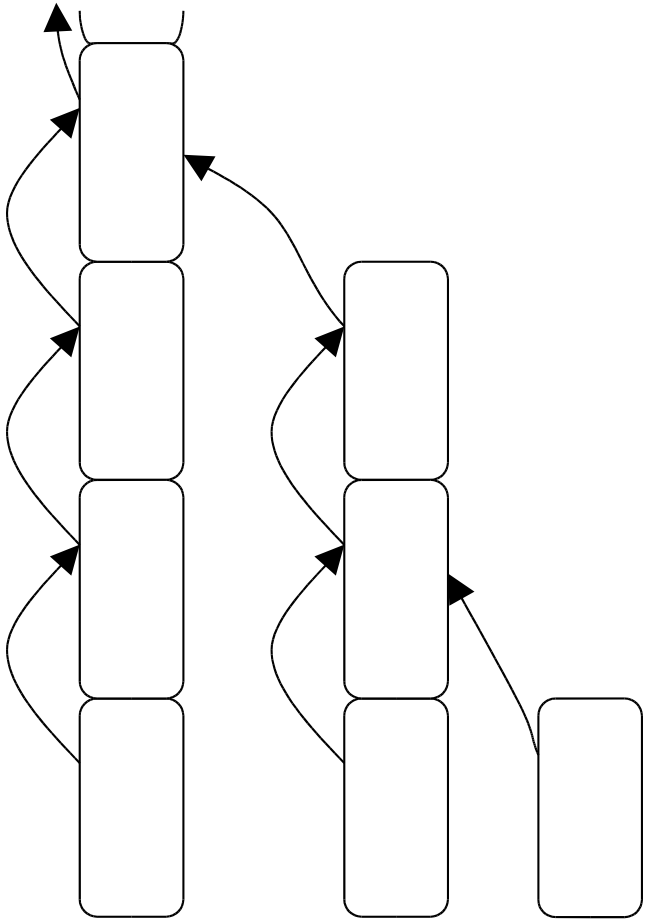


7.5 mehrere Keller

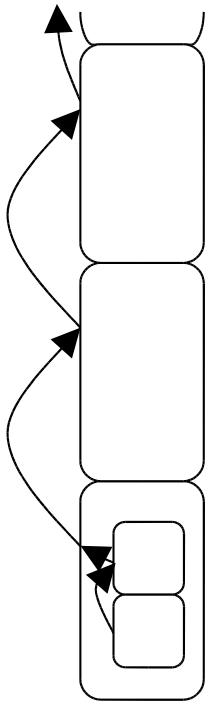
Disjunkte Keller



Kaktuskeller
(auf der Halde)



Keller rekursiv
(max. Kellergröße?)



7.5 Prozeduraufruf Aufgaben

1. Zustand sichern
2. Neue Prozedurschachtel generieren
3. Kellerpegel setzen
4. Statischen und dynamischen Vorgänger eintragen
5. Parameterübergabe
6. Unterprogrammsprung
7. Rücksprungadresse sichern (bei Rekursion/geschachteltem Aufruf)
8. Prozedurrumpf ausführen
9. Rücksprung
10. Ergebnisrückgabe
11. Kellerpegel zurücksetzen, Zustand wiederherstellen

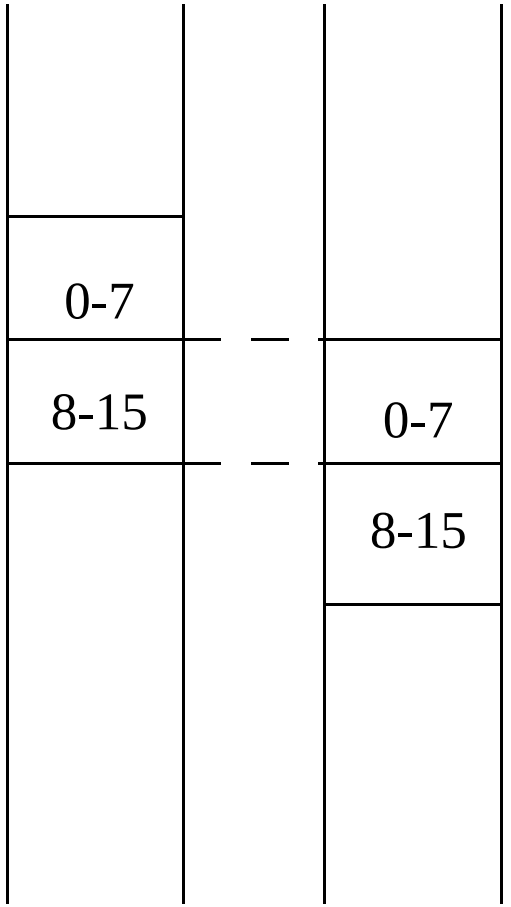
Reihenfolge teilweise veränderbar

7.5 Zustand sichern

Alle Register

Auswahl bestimmter Register

Spezialfall Registerfenster (*SPARC, Itanium*)



Probleme:

- Register sichern sehr zeitaufwendig
  (n bzw. $2n$ Speicherzugriffe)
- bei automatischer Speicherbereinigung:
  enthält die Sicherung Referenzen?

7.5 Geschachtelte Prozeduren

Prozedur p innerhalb einer Prozedur p' deklarierbar, z.B. in Pascal, Modula

Prozedur p' ist statischer Vorgänger sv von Prozedur p

Lokale Variablen v' von p' auch in p gültig

Addressierung:

$$adresse(v) = speicher[speicher[schachtel] + r_{sv}] + r_v$$
$$= << schachtel > + r_{sv} > + r_v$$

Achtung:

- Statischer Vorgänger unnötig, wenn alle Prozeduren auf Schachtelungstiefe 1, z.B. C oder oo-Sprachen
  - aber Vorsicht bei inneren Klassen in Java
- Laufzeitkeller unnötig für Sprachen ohne Rekursion, z.B. ursprüngliches Cobol oder Fortran (nur statische Variable)

7.5 Dynamischer und Statischer Vorgänger

dynamischer Vorgänger: Verweis auf Schachtel Aufrufer statischer Vorgänger: Verweis auf Schachtel der statisch umfassenden Prozedur

- statischer Vorgänger überflüssig, wenn alle Prozeduren auf Schachtelungstiefe 1 (C, C++)

Vorgänger eintragen, um bei Rückkehr und bei Zugriff auf globale Größen die richtige Schachtel zu finden

Implementierung:

- Ausprogrammieren
- Spezialbefehl, z.B. auf MC 680x0 für dynamische Vorgänger

7.5 Rücksprungadresse

Rücksprungadresse in den Keller:

- Ausprogrammieren (`Intel`)
- Spezialbefehl (680x0)
- in speziellem Register (RISC), von dort abholen

7.5 Parameter-, Ergebnisübergabe

- Wertaufruf
- Wert-Ergebnis-Aufruf
- Referenzaufruf
- Namensaufruf (wie gebundene Prozedur behandeln)
- Ergebnisaufruf

Alternativen

- Aufrufendes Programm oder Unterprogramm berechnet Parameter
- Aufrufendes Programm oder Unterprogramm speichert Resultat
- Spezialfall: Parameter oder Resultat im Register bei Ergebnisaufruf

7.5 Wert-/Ergebnisaufruf

Parameter ist lokale Variable

- bei Wertaufruf vom Aufrufer initialisiert
- bei Ergebnisaufruf nach Prozedurende Wert vom Aufrufer abgeholt
  - Prozedur kann Ergebnis selbst abspeichern, wenn sie zusätzlich die Adresse des Ergebnisparameters kennt

Funktionsergebnisse wie Ergebnisparameter behandeln

Aus Effizienzgründen bei geringer Parameterzahl (n=1): Argument und/oder Ergebnis in Register übergeben (nicht möglich bei gebundenen Prozeduren oder polymorphem Aufruf!)

daher: zuerst neue Schachtel anlegen, dann erst Argumente berechnen

- bei Wertübergabe von Reihungen wird der dynamische Teil der neuen Schachtel um die Reihung verlängert

7.5 Referenzaufruf

- Parameter ist lokale Variable, wird vom Aufrufer mit Adresse des Arguments initialisiert
  - wenn Argument ein Ausdruck (keine Variable) ist: Argument an neue Hilfsvariable beim Aufrufer zuweisen, Adresse Hilfsvariable übergeben
  - Schutz vor unzulässiger Zuweisung an den Ausdruck, z.B. in Fortran
- alle Zugriffe auf den Parameter in der Prozedur haben eine zusätzliche Indirektionsstufe: der Wert des Parameters (Adresse) kann explizit weder gelesen noch überschrieben werden

- Referenzaufruf nur möglich, wenn Aufrufer und Aufgerufener im gleichen Adreßraum
  - im verteilten System simuliert durch Übergabe einer Lese- und einer Schreibprozedur (*get*, *set*)

7.5 Vorsichtsmaßnahmen

Zuerst Schachtel einrichten, dann Argumente berechnen

- Argumentberechnung kann zu weiteren Aufrufen führen!
- (Kollision mit der Halde vermeiden)

Schachtel des Aufgerufenen erst nach Abholen der Ergebnisse streichen

- (Kollision mit der Halde vermeiden)

7.5 Rückhaltestrategie

Annahme: eine gebundene Prozedur p besitzt gebundene Parameter aus der Prozedur p', in der gebunden wird (also aus der Schachtel von p'). p werde erst nach Verlassen von p' aufgerufen. Bei

- Wertaufruf: kein Problem
- Ergebnisaufruf, Referenzaufruf: Schachtel von p' muß erhalten bleiben, um auf den Parameter zugreifen zu können: Rückhaltestrategie (retention strategy)
- Namensaufruf oder gebundener Parameter ist lokale Prozedur von p' : ebenfalls Schachtel zurückhalten.

Implementierung: Schachteln gehen wie beim Kaktus-Keller auf die Halde.

notwendig in: Simula 67, funktionalen Sprachen, bei Strömen in Sather-K. Die Vermeidung der Situation führt zu Beschränkungen in vielen anderen Sprachen.

7.5 Polymorpher Unterprogrammsprung

Wenn alle Untertypen zur Übersetzungszeit bekannt:

- typecase auf ersten Parameter,

Bei getrennter Übersetzung oder dynamischem Nachladen:

- Eintrag in Haschtabelle mit Typeintrag liefert richtige Unterprogrammadresse:
- je eine Haschtabelle für jeden Vererbungsbaum, Tabelle wird vom Binder konstruiert
  - also: alle polymorphen Aufrufe mit zusätzlicher Indirektionsstufe belastet

Trick: Merke Typeintrag und Adresse des jeweils zuletzt aufgerufenen Unterprogramms; teste, ob nächster Aufgerufene denselben Typeintrag besitzt und springe dann direkt, sonst über Haschtabelle