



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

7.0 Die Synthesephase

Aufgabe: attributierter Strukturbaum → 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/Optimierung**: Code für abstrakte Zielmaschine ZM (ohne Ressourcenbeschränkung) herstellen und optimieren, Repräsentation als **Zwischensprache IL**
- **Codeerzeugung**: Transformation IL → symbolischer Maschinencode
 - unter Beachtung Ressourcenbeschränkungen
- **Assemblieren/Binden**: symbolische Adressen auflösen, fehlende Teile ergänzen, binär codieren

7.0 Zwischensprache *IL*

2 Klassen von Zwischensprachen:

- Code für **Kellermaschine** mit Heap, 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

7.0 Zwischensprache *IL* II

Im Fall „Code für RISC-Maschine mit unbeschränkter Registerzahl“ drei Darstellungsformen:

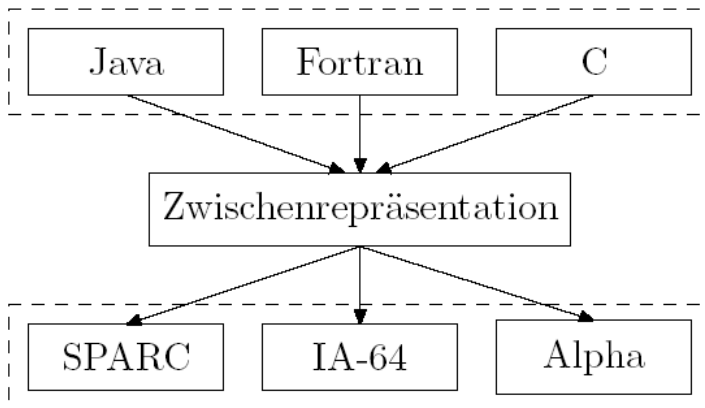
- **keine explizite Darstellung:** *IL* erscheint nur implizit bei direkter Codeerzeugung aus AST: höchstens lokale Optimierung, z.B. Einpaßübersetzung
- **Tripel-/Quadrupelform:** Befehle haben schematisch die Form
 - $t_1 := t_2 \text{ † } t_3$ oder
 - $m: t_1 := t_2 \text{ † } t_3$
 - analog auch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

7.0 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

7.0 UNCOL (Wdh)



Idee 1961 von T. B. Steel, Jr.
UNCOL: UNiversal Computer Oriented
Language.
Niemals direkt realisiert

Beste heutige Annäherung an UNCOL: .NET CLI/CLR
Argumente

- Ablaufsteuerung in allen Sprachen fast gleich
- Anzahl der Basistypen gering und ähnlich
- OO-Eigenschaften durch Verbunde simulierbar

7.0 .NET: Ziele

- Unabhängigkeit von der Quellsprache
- Aber Plattform abhängig
 - Die Bibliotheken stellen Windows Funktionen bereit
- Verteiltes Ablaufmodell: Die Zielanwendung sind Web-Dienste
- Unterstützung von Selbstauskünften (Reflection)
- Versionierung
- Dynamisches Laden von Klassen, aber nur ein Klassenlader (in Java sind Klassenlader benutzerdefinierbar)

7.0 .NET: Sprachunabhängigkeit

- Erlangung der Interoperabilität von Programmiersprachen durch Übersetzung in eine gemeinsame Zwischensprache CIL (common intermediate language, auch MSIL)
- Eigenschaften der CIL
 - Common Type System CTS
 - Typsystem CLS (Common Language Specification): Untermenge des CTS, die von allen Sprachen unterstützt wird
 - Common Language Runtime System CLR
 - Kellermaschinen Architektur, automatische SB, Versionierung, keine “DLL Hölle”, Sicherheitsmechanismen, OO-Klassenbibliotheken
- **Aber: CIL nicht direkt ausführbar, muß erst in Machinensprache übersetzt werden, weil die Operationen typfrei sind** (z.B. `add` statt `dadd` für `double`)
Standard verbietet sogar die Interpretation!

7.0 CIL Beispiel



```
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
```

7.0 Beispiel in 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
```

Hinweis: Java Bytecode hat getypte Operatoren (dadd), kann somit einfach interpretiert werden.
CIL Operatoren sind typfrei (add) und müssen zunächst übersetzt werden.

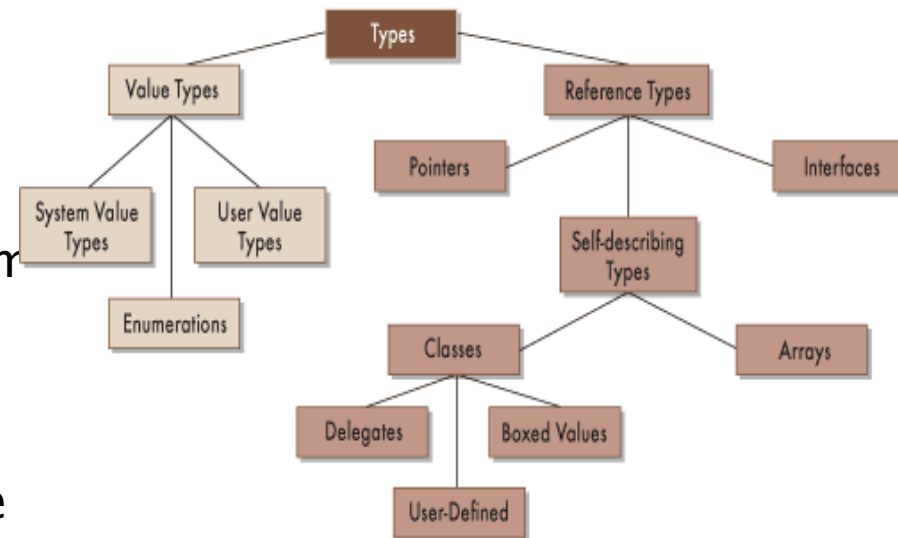
Gemeinsames Typsystem (CTS) I

CTS:

- Menge von Typen, die von irgendwelchen .NET-Sprachen benutzt werden können; Menge nicht minimal, sondern maximal
- Alle Typen umfassen (nur) Objekte, d.h. keine “primitive values types”

unterscheide

- Werttypen: Aufzählungstypen, ganze benutzerdefinierte Verbunde
- Referenztypen: Klassen, Reihungen, Delegierte (Mengen von Funktionszeigern)
 - schachteln (boxing): konvertiere Werttyp in Referenztyp
 - offenlegen (unboxing): konvertiere Referenztyp in Werttyp



Gemeinsames Typsystem (CTS) II

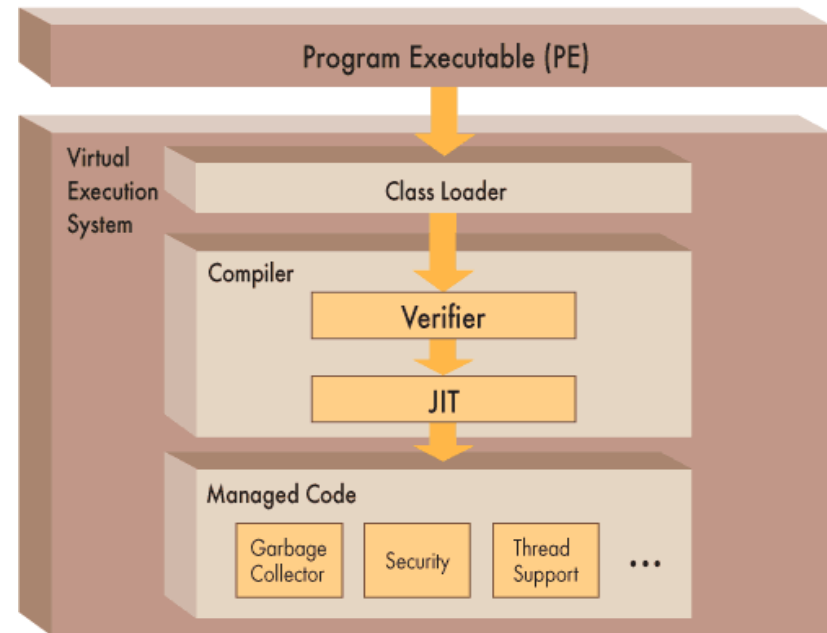
Minimales Typsystem (common language specification) CLS:

der gemeinsame Nenner bei der Zusammenarbeit von Sprachen

- Teilmenge der CTS, muß von allen Übersetzern unterstützt werden, wenn ihre Quellsprache mit anderen CLS-konformen Sprachen zusammenarbeiten soll
- Alle exportierten Typen und Schnittstellen müssen den 41 CLS Regeln gehorchen, z.B.
 - Regel 15: die Typen von Reihungselementen müssen CLS-Typen sein. Reihungen haben eine feste Anzahl von Stufen; Indizes starten bei 0. ...

.NET-Laufzeitsystem CLR

- CIL-Code wird nicht interpretiert, sondern durch VES (virtuelles Ausführungssystem) vor oder während der Ausführung in Maschinensprache übersetzt
- CLR enthält
 - just-in-time Übersetzer (JIT)
 - Speicherbereiniger
 - Klassenlader (um Päckchen zu laden)
- Code-Inspektor (verifier): zur Prüfung der Typ- und Zugriffssicherheit von CIL Programmen



Managed Code und Daten

- managed code: Ausführung unter Aufsicht der CLR
- unmanaged code: anderer Code
- Interoperabilität von managed und unmanaged Code:
 - Binde existierende DLLs mit P/Invoke (platform invoke)
 - COM/COM+-Binden via COM-Interop-mapping
- managed data:
Daten unter Aufsicht des Speicherbereinigers (nicht persistent)
- unmanaged data (für unsicheren Code):
Daten manuell alloziert und freigegeben (vom Programmierer)
- managed code kann managed und unmanaged Daten bearbeiten

Code Prüfung

Problem: unsicherer Code, insbesondere Verweis-Arithmetik

- Code Prüfung durch Verifikator und Klassenlader:
 - Verifikator prüft konservativ, daß das Programm alle Regeln bezüglich Speicherzugriff einhält:
 - Das Programm greift nur auf Speicher zu, der für das Programm alloziert wurde.
 - Das Programm greift auf Objekte nur via deren Schnittstellen zu.
 - Der Klassenlader prüft, daß alle initialen Werte vom passenden Typ sind.

Der Verifikator kann abgeschaltet werden, um mit beliebiger Verweis-Arithmetik und Funktionszeigern umzugehen

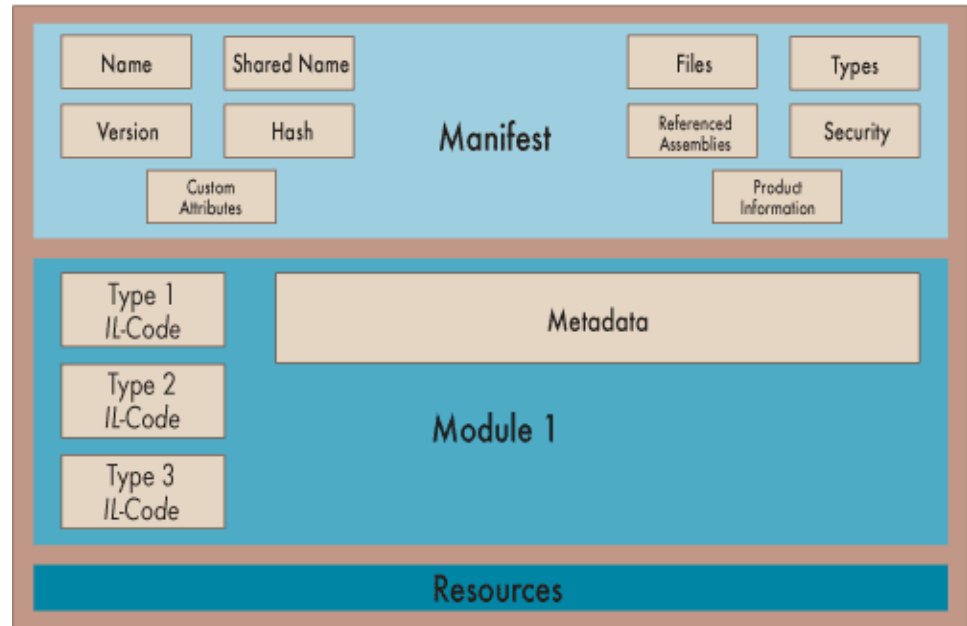
- ANSI C und Mehrfachvererbung können so unterstützt werden.
- Ab Version 3 wird Mehrfachvererbung direkt unterstützt (?).

Päckchen (Assemblies)

Päckchen (assembly):

.NET-Komponente

- logische Einheit ähnlich einer DLL oder einer EXE-Datei
- elementare Einheit bezüglich Installation, Konfiguration, Versionierung, Sicherheit, Laufzeitverwaltung
- Beim Laden aktiviert das VES einen Laufzeitwirt, um die Übersetzung und Ausführung eines Päckchens zu überwachen



Der Inhalt von Päckchen

Ein Päckchen enthält

- einen oder mehrere Module (program executables, PE): ausführbarer Code für die Typen in den Moduln
- Versionsnummer (starker Name) spezifiziert durch .assembly:
 - Name
 - Version: (Major.Minor.Build.Revision)
 - Sprache (en-US, de-DE, ...)
 - öffentlicher Schlüssel des Herstellers
- „manifeste“ Information über einen Modul oder global über das Päckchen:
 - Beschreibung der Moduln und Ressourcen des Päckchens
 - Import Schnittstelle: Referenzen auf importierte Päckchen
 - Export Schnittstelle: exportierte Typen und Ressourcen
- Spezifikation der zur Ausführung benötigten CLR Version

Päckchen: Reflexion und Verwaltung (Versionierung)

Ein Päckchen enthält explizite (vom Übersetzer erzeugte) Metadaten:

- Definitionstabellen: Beschreibung der Elemente in einem Modul
- Verweistabellen: Beschreibung fremder Elemente, die im Modul verwandt, aber nicht definiert sind.
- Manifeste Daten
Dynamische Abfragen mit Reflexionsmechanismen möglich
- Metadaten + Reflexion bedeuten
 - keine IDL nötig
 - Keine Registratur für Päckchen nötig, Päckchen können irgendwo im Dateisystem liegen
 - private Päckchen in lokalen Verzeichnissen einer Anwendung
 - gemeinsame Päckchen meist in globalen Verzeichnissen (global assembly cache)
- Installation verlangt Signatur mit privatem Schlüssel:
- Hilfe zur Vermeidung von Namenskonflikten: starke Namen

7.0 SSA-Form

Zwischensprachen können die SSA-Eigenschaft haben:

- Jede Variable hat genau eine statisch bekannte Definitionsstelle.

- Beispiel:

```
A = 3;
```

```
A1 = 3;
```

```
B = 4;
```

```
B = 4;
```

```
A = A+B;
```

```
A2 = A1+B;
```

- **Problem bei Steuerfluß-Vereinigung:**

```
if (A<B) M = A; else M = B;
```

Lösung:

```
if (A<B) M1 = A; else M2 = B;
```

```
M3 =  $\varphi$ (M1, M2);
```

- **φ -Funktionen** schalten eines ihrer Argumente, je nach tatsächlichem Steuerfluß, als Ergebnis durch.

7.0 SSA-Form II

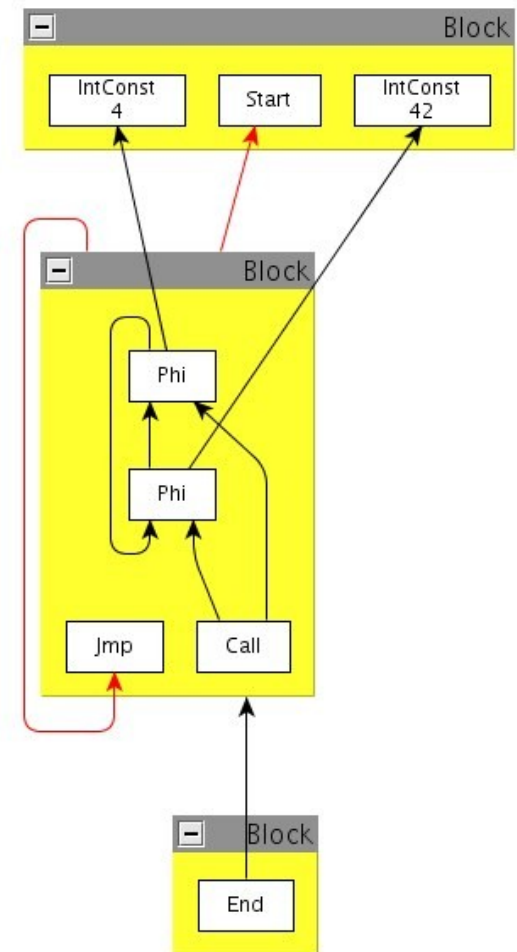
Zwei Varianten:

- Tripelform (siehe letzte Folie)
- Graphbasierte Form
 - Knoten symbolisieren Werte
 - Kanten je nach Richtung:
Entweder Datenfluß oder
Datenabhängigkeit
 - Variable „zu Werten abstrahiert“
 - Beispiel:

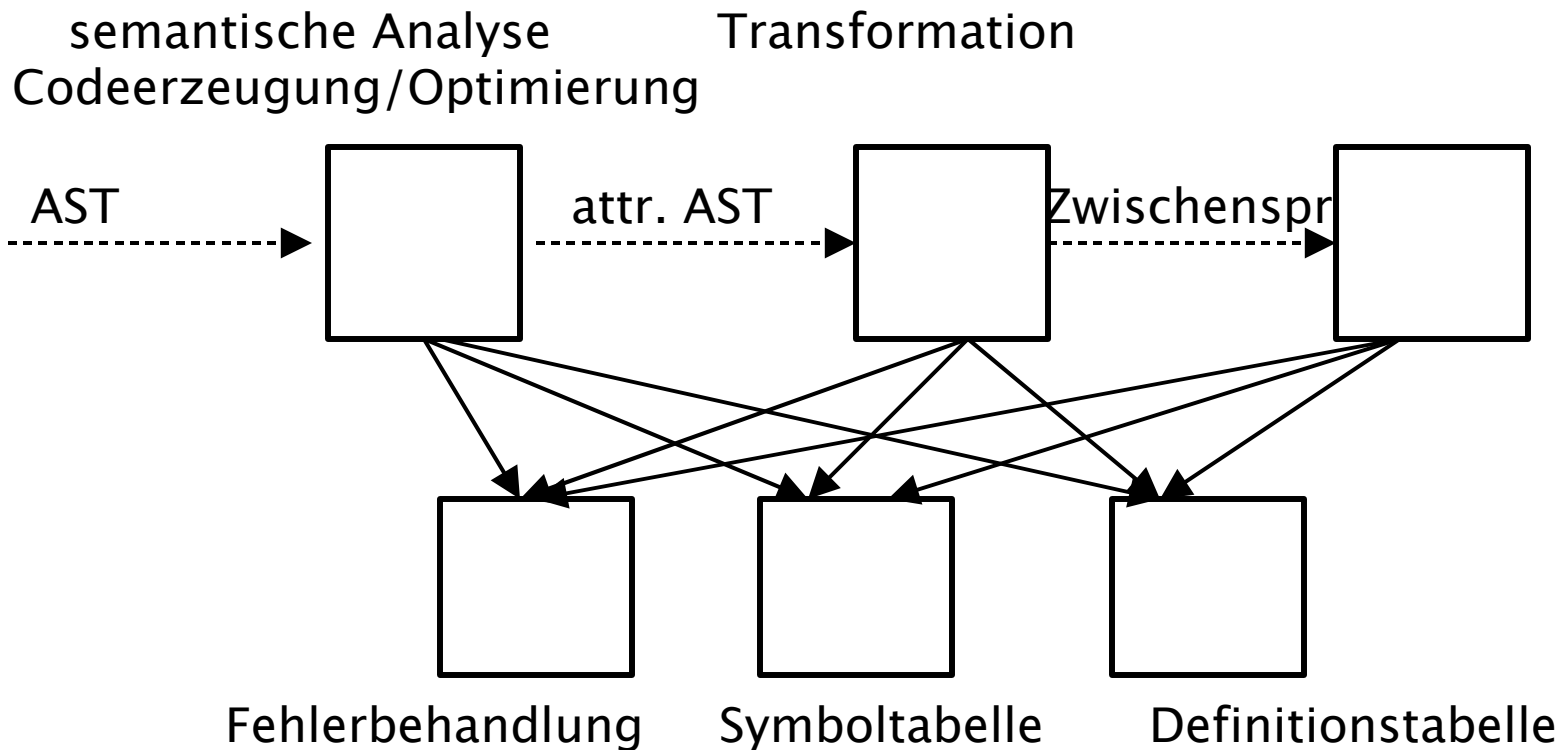
```
while (1) {  
    t = A; A = B; B = t;  
    printf("%d %d\n", A, B);  
}
```

Grafik zeigt Datenabhängigkeitsgraph
rot = Steuerabh., schwarz = Datenabh.

- **Problem:** Alle ϕ -Funktionen müssen simultan ausgewertet werden! (Übung)



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

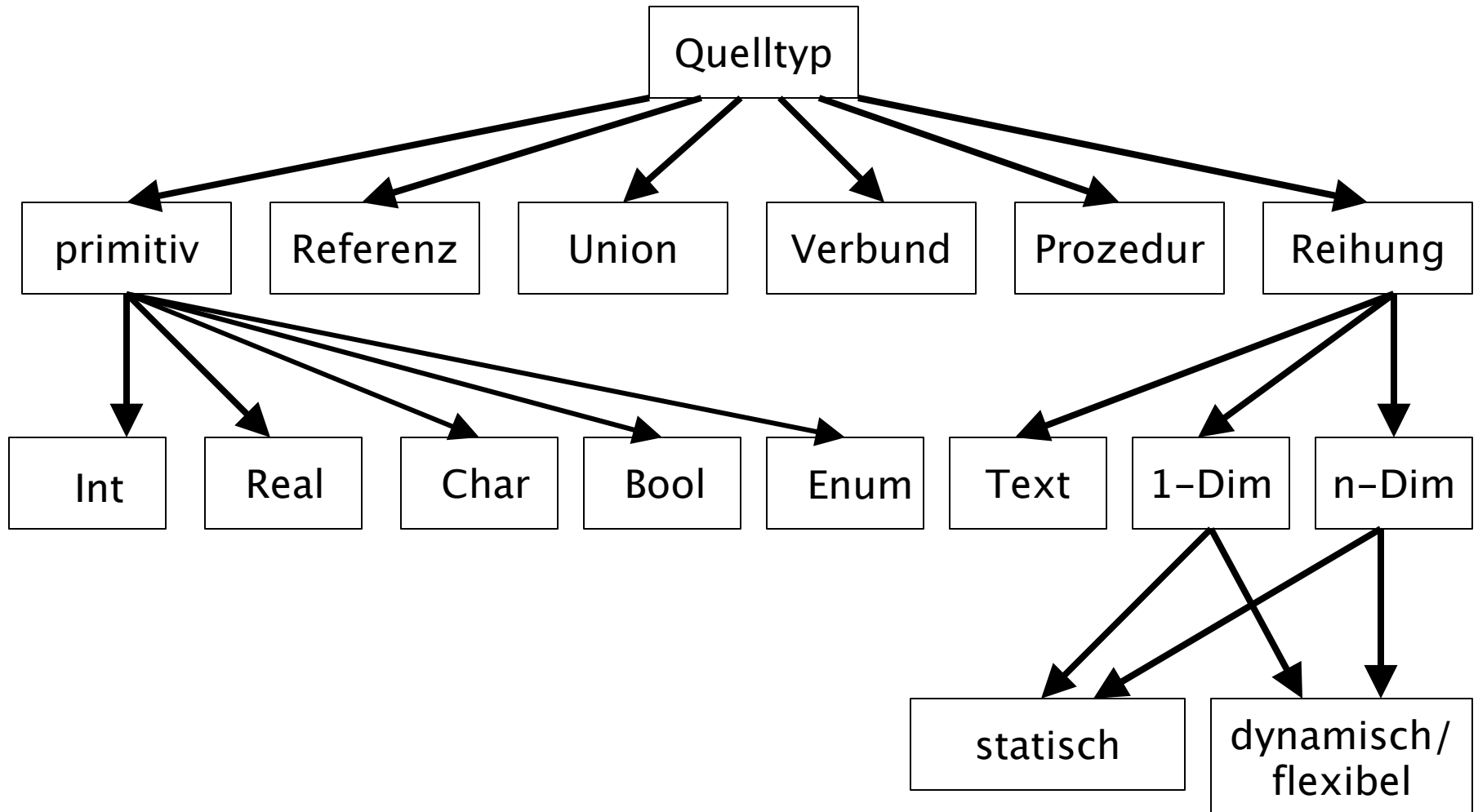
- 0. Einbettung
- 1. Typabbildung
- 2. Speicherabbildung
- 3. Abbildung der Operatoren
- 4. Abbildung der Ablaufsteuerung
- 5. Speicherorganisation und Prozeduraufruf

7.1 Typabbildung

Datentypen sind

- **Einfach:**
 - Aufzählungstyp, Referenz(!), `bool`, `char`, `int`, `unsigned`, `float`, ...
 - 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, $\neq 0$ true
 - C Interpretation von `int` in if-Anweisungen
 3. 0 true, $\neq 0$ 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, UTF-8, UTF-16

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

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

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 Referenzen definiert maximale Größe des Adreßraums
- 16-bit, 32-bit, 64-bit Referenzen?

7.1 Verbunde

- Verbunde heissen auch record oder struct
- Folge (oder Menge ?) von Elementen
- Ausrichtung des Verbundes ist maximale Ausrichtung der Elemente
- Länge ist Summe der Länge der Elemente plus Verschnitt wegen Ausrichtung
- variante Verbunde (variant record) wie Vereinigungstypen (union) behandeln

7.1 Objekte

allgemein: Objekte wie Verbunde behandeln

- bei 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 oder Sprungtabelle)?
 - oft benutzt, aber besser in Klasse als im Objekt
- 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,
                      block.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 zu 1 Zuordnung entsprechend Speicherabbildung
Vorsicht: Bei manchen Sprachen Prüfung auf 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` → `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 >Adresse x< Register
```

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;

...

M:

JMP M

...

M:

beendet Grundblock

7.4 Bedingte Anweisung

if B then S else S' end;

B

JMP ZERO Then

S

JMP Ende

Then:

S'

Ende:

...

Beachte: Sprünge mit erfüllter Bedingung oft schneller!

3 Grundblöcke: B; JMP ZERO Then, 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 günstiger, wenn Sprünge mit erfüllter Bedingung schneller sind
- **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 ( $i \leq e$ ) { // bei  $s < 0$ :  $i \geq e$   
     $S$ ;  $i = i + s$ ;  
}
```

„Richtiger“: $_o$ ist der letzte Wert $\leq e$, für den die Schleife ausgeführt wird, bei $s == 1$ sinnvoll, da keine Div- oder Mod-Operation und $e = _o$

```
if ( $a \leq e$ ) {  
     $i = a$ ;  $\_x = a \% s$ ;  $\_y = e \% s$ ;  
     $\_o = (\_y \geq \_x) ? e - (\_y - \_x) : e - (\_y + s - \_x)$ ;  
    while (true) {  
         $S$ ; if ( $i == \_o$ ) break; else  $i = i + s$ ;  
    }  
}
```

Funktioniert immer, auch bei $e = \text{maxint}$!

Aber Vorsicht, wenn $a \leq e$ 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 $a + \text{Sprungbefehlsgröße}$ für normales Ende

Ignorieren der Ausnahme: Sprung in Ausnahmebehandlung wird durch `nop` ersetzt

7.4 Beispiel

<code>c=0;</code>	<code>s1: ST >c< 0</code>	<code>c=a+1+b;</code>	<code>t10: LD <a></code>
	<code>s2: LD <x></code>		<code>t11: ADD t10 1</code>
<code>if x > 0 {</code>	<code>s3: GT 0</code>		<code>t12: LD </code>
	<code>s4: JMP FALSE u1</code>		<code>t13: ADD t11 t12</code>
<code>a=2;</code>	<code>t1: ST >a< 2</code>		<code>t14: ST >c< t13</code>
	<code>t2: LD <a></code>	<code>}</code>	<code>t15: JMP u1</code>
<code>b=a*x+1;</code>	<code>t3: LD <x></code>		
	<code>t4: MUL t2 t3</code>	<code>x=c;</code>	<code>u1: LD <c></code>
	<code>t5: ADD t4 1</code>		<code>u2: ST x u1</code>
	<code>t6: ST >b< t5</code>		
<code>a=2*x;</code>	<code>t7: LD <x></code>		
	<code>t8: MUL 2 t7</code>		
	<code>t9: ST >a< t8</code>		

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:

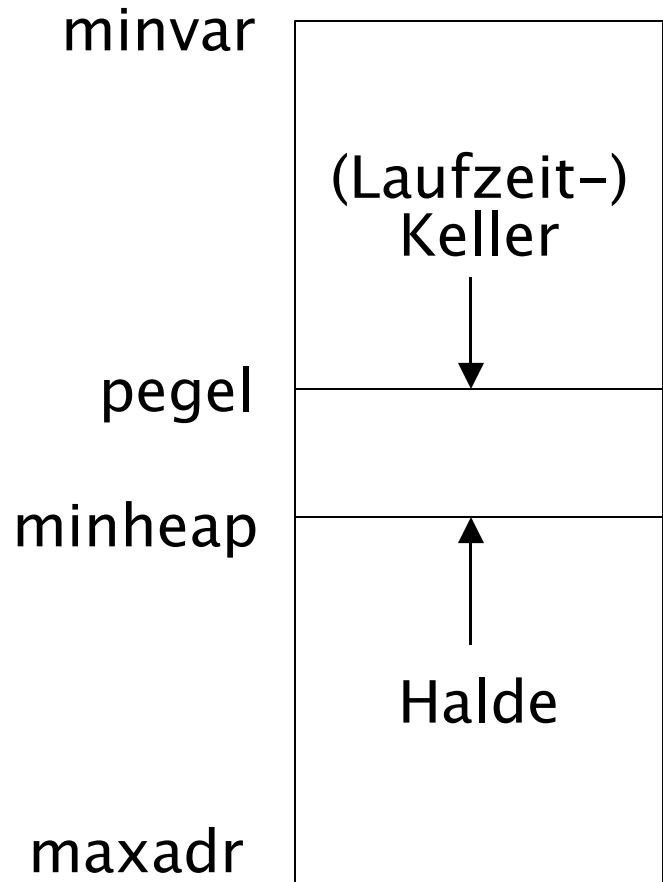
- alle Programmvariablen allozieren
- statische/dynamische/anonyme Allokierung unterscheiden
- Vorbereitung auf dynamische Prozedurschachtelung einschl. Rekursion

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:
 $\text{minheap} > \text{pegel}$

7.5 Basisadressen

minvar Basis statischer Variabler
(Beginn Keller)

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

pegel Kellerpegel

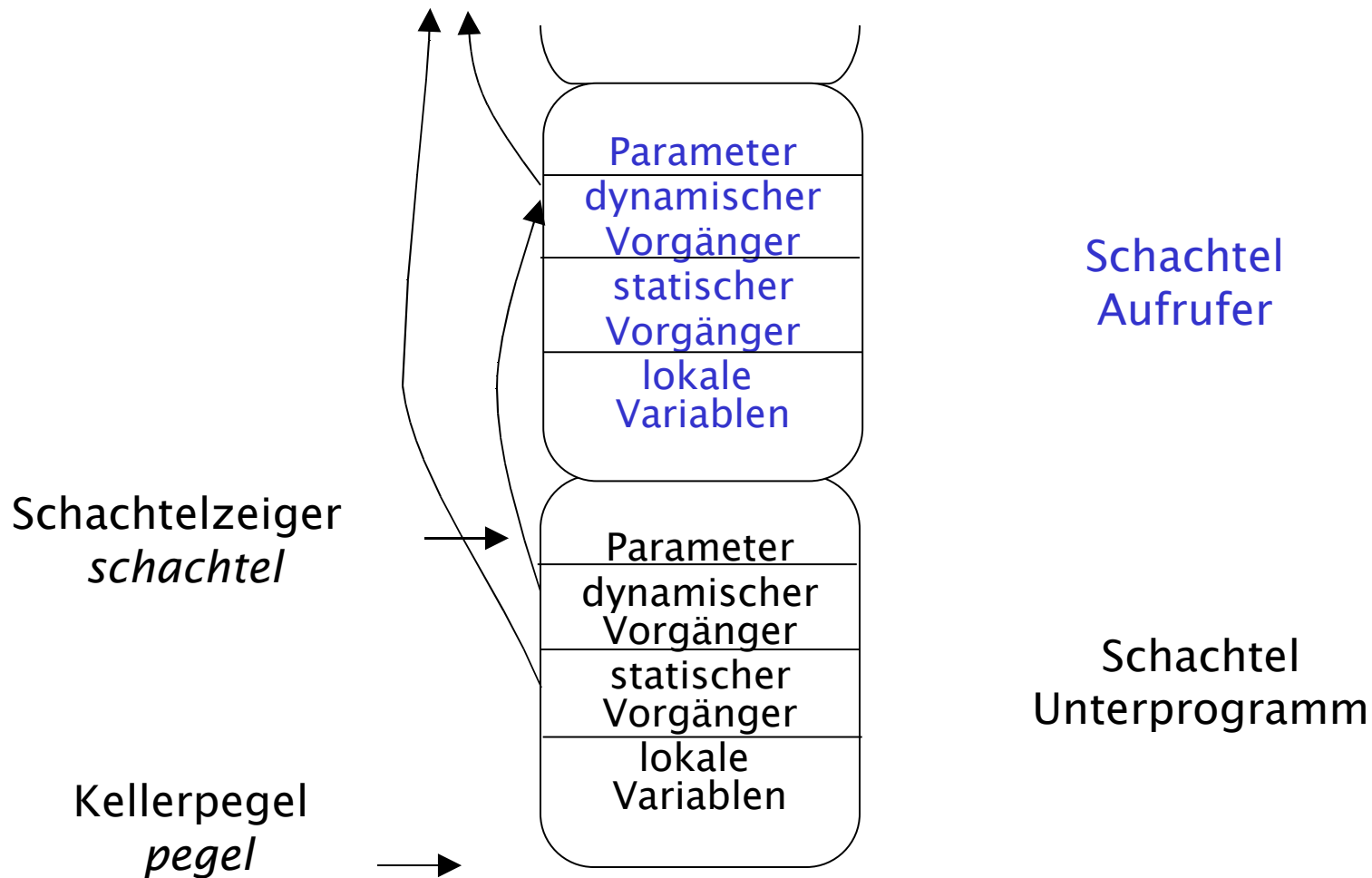
Adressen statischer Variabler v mit Rel.Adr. r_v :

$$\textit{minvar} + r_v$$

Adressen dynamischer Variabler v mit Rel.Adr. r_v :

$$\textit{schachtel} + r_v$$

7.5 Laufzeitkeller



7.5 Schachtel

Einteilung **statisch/dynamisch**

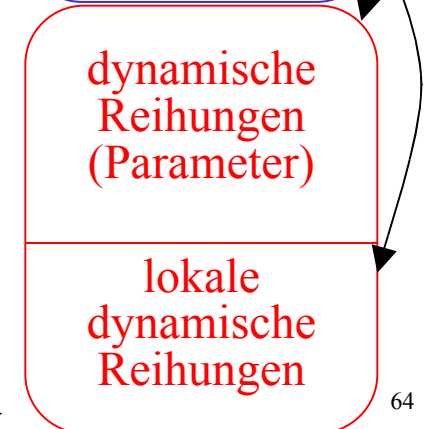
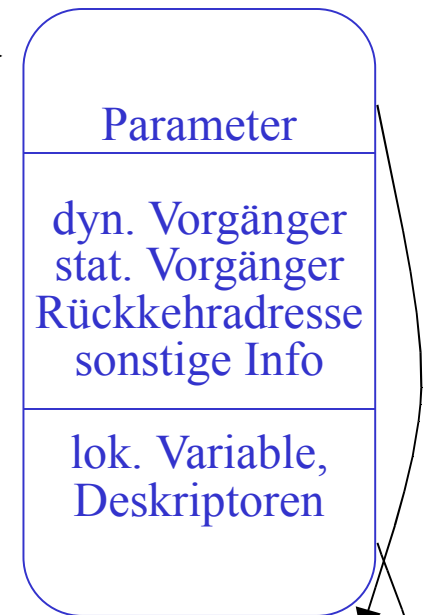
Reihungsdeskriptoren im statischen Teil

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)

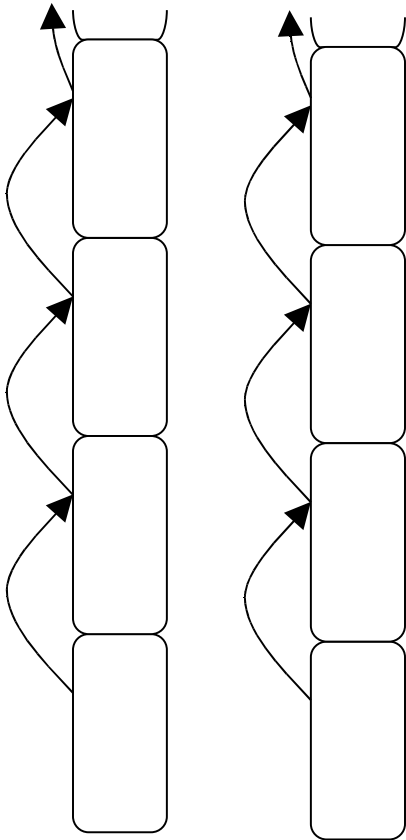
Schachtel →



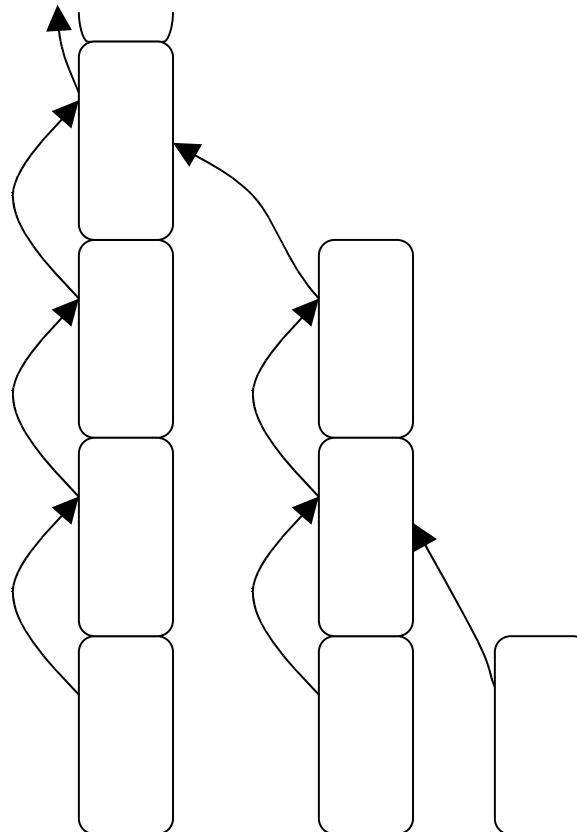
Pegel →

7.5 mehrere Keller

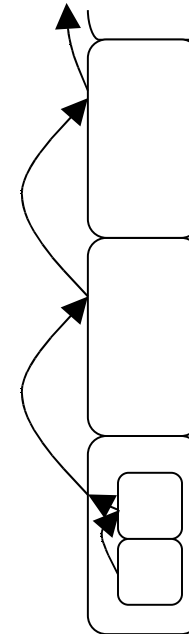
Disjunkte Keller



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. Unterprogramm sprung
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?

0-7		
8-15		0-7
		8-15

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:

$$\begin{aligned} \text{adresse}(v) &= \text{speicher}[\text{speicher}[\text{schachtel}] + r_{sv}] + r_v \\ &= \langle\langle \text{schachtel} \rangle + r_{sv} \rangle + r_v \end{aligned}$$

7.5 Dynamischer und Statischer Vorgänger

- dynamischer Vorgänger: Verweis auf Schachtel Aufrufer
- statischer Vorgänger: Verweis auf Schachtel der statisch umfassenden Prozedur
 - Statischer Vorgänger unnötig, wenn alle Prozeduren auf Schachtelungstiefe 1, z.B. C oder viele OO-Sprachen
 - aber Vorsicht bei inneren Klassen in Java
 - Vorgänger eintragen, um bei Rückkehr und bei Zugriff auf globale Größen die richtige Schachtel zu finden
- Laufzeitkeller unnötig für Sprachen ohne Rekursion, z.B. ursprüngliches Cobol oder Fortran (nur statische Variable)

Implementierung:

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

7.5 Rücksprungadresse sichern

Rücksprungadresse in den Keller :

- Spezialbefehl (680x0, IA-32)
- in speziellem Register (RISC), von dort abholen

Bei spezieller Anordnung der Schachtelemente:

Ausprogrammieren nötig, da Spezialbefehle diese Anordnung nicht berücksichtigen

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: Argument und/oder Ergebnis in Register übergeben (unabhängig von Systemaufrufkonvention; 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 Unterprogramm sprung

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