# Verifikation von Übersetzern

**Wolf Zimmermann**

**Verifikation von Übersetzern**



## Lernziele

- Kenntnis der Methoden der Übersetzerverifikation
- An Hand einfacher Beispiele selbstständig Übersetzer verifizieren können.

1. Einleitung
2. Grundlagen und Semantik von Programmiersprachen
3. Korrektheit der Speicherabbildung
4. Korrektheit der Zwischencodeerzeugung
5. Korrektheit der Codeerzeugung und der Assemblierung



# Kapitel 1
# Einleitung

**Wolf Zimmermann**

**Verifikation von Übersetzern**



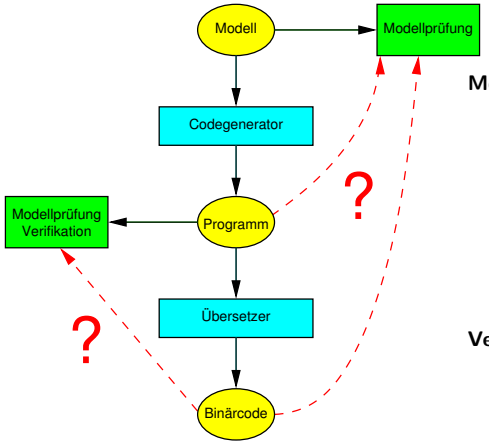
## Inhalt

### Ziele

- Kennenlernen der grundsätzlichen Vorgehensweise bei der Verifikation von Übersetzern
- Verständnis und Beurteilung von Korrektheitsbegriffe
- Skizze einer Verifikation

1. Einleitung
2. Korrektheit von Übersetzern
3. Architektur verifizierter Übersetzer
4. Ausblick auf den Rest der Vorlesung

## 1.1 Korrektheit von Übersetzern




**Modellbasierte Entwicklung**

- Ausgangspunkt: Modell des zu erstellenden Systems
- Überprüfung von Eigenschaften mit Modellprüfung
- Codegenerierung (meist C-Code)
- Erfüllt C-Code geprüfte Eigenschaften?
- Erfüllt Binärcode geprüfte Eigenschaften?

**Verifikation**

- Verifikation des Programms
- Validierung von Eigenschaften (z.B. Schutz, Absturzsicherheit)
- Gilt das auch für den Binärcode?



## Ziele und Randbedingungen

### Ziele (Verifix)

Methoden zur Konstruktion von verifizierten Übersetzern

- von realistischen Programmiersprachen
- in Maschinensprachen handelsüblicher Prozessoren,
- die Code derselben Qualität wie unverifizierte Übersetzer erzeugen können.

### Grundannahmen

- Hardware ist korrekt
- Betriebssystem ist korrekt



## Ja, Übersetzer haben Fehler!

### Fehler in Übersetzern

- Borland Pascal Compiler Bug List enthält ca. 50 Fehler im Übersetzer
- Java Compiler Bug Database from Sun enthält ca. 90 Fehler im Übersetzer

### Borland Pascal Compiler Bug List: Bug # 539

- Funktionsergebnisse werden in Aktivierungsverbund gespeichert
- Falls Funktionen keine anderen Funktionen aufrufen, dann wird das Resultat (4 Bytes) im Register eax statt im Aktivierungsverbund gespeichert
- Bei Rückkehr wurde nicht berücksichtigt, ob Optimierung statt gefunden hat

$\Rightarrow$ Kellerzeiger wurde manchmal um 4 Bytes zu viel dekrementiert



## Ja, Übersetzer haben Fehler!

### Java Bug Database: Bug # 4078305 (behoben im Release 1.2fcs)

```
class B {
   static int foo = 2
   static {
      System.out.println(``Initializing B'');
   }
}
class MAIN {
   public static void main(String[] args) {
      int x = B.foo;
   }
}
```

```
class B { ...}
class MAIN {
   public static void main(String[] args) { }
}
```

- x wird nie benutzt

$\Rightarrow$ Anweisung kann gestrichen werden

- Nur korrekt, wenn rechte Seite der Zuweisung keine Nebenwirkungen hat
- Hier ist Nebenwirkung statische Initialisierung

## 1.2 Korrektheit von Übersetzern

**Häufige Definition**

Übersetzung ist semantikerhaltend

**Problem**

**Definition ist so einfach wie unklar**

**1. Versuch**

- (Imperative) Programme definieren Zustandsübergänge
- Diese müssen erhalten bleiben

**Beispiel 1.1: Korrekte Übersetzung**

```
(1) int i=24;
(2) int j=36;
(3) while (i!=j) {
(4)    if (i>j) i=i-j;
(5)    else j=j-i;
(6)    printf("i=%d",i);
(7)    printf("  j=%d\n",j);
(8) }
```

```
(1) printf("i=24  j=12\n");
(2) printf("i=12  j=12\n");
```

- Die Schleife (3)–(8) und die bedingte Anweisung werden im rechten Programm nicht erhalten
- ⇒ Keine Semantikerhaltung?
- Die Ausgabe der beiden Programme ist jeweils identisch



## Was heißt eigentlich Korrektheit?

**2. Versuch: Erhaltung beobachtbaren Verhaltens**

- Programme interagieren bei ihrer Ausführung mit der Umwelt (**beobachtbares Verhalten**)
- ☞ Lesen von Standardeingabe, Schreiben in Standardausgabe, Lesen von einer Datei, Schreiben in eine Datei, Lesen von Sensordaten, Schreiben von Sensordaten
- ⇒ Es gibt beobachtbare Zustände
- Korrektheit = Erhaltung der beobachtbaren Zustandsübergänge

**Beispiel 1.2: Erhaltung beobachtbaren Verhaltens**

```
(1) void f(int n) {
(2)   int m=100000000;
(3)   double a[m][m];
(4)   if (n==0) return;
(5)   int i,j;
(6)   f(n-1);
(7)   for (i=0;i<m;i++)
(8)     for (j=0;j<m;j++)
(9)       a[i][j]=i+j;
(10)}
(11)int main(void) {
(12)  f(10000000);
(13)  printf("fertig!");
(14)}
```

- Programm endet mit `Segmentation fault (core dumped)`
- Ursache: Platzbedarf $\approx 8 \cdot 10^{12}$ TByte
- ⇒ Sprengt vorhandenen Platz auf praktisch jeder Maschine
- ⇒ Programm kann nicht korrekt übersetzt werden
- ⇒ Es gibt keinen korrekten Übersetzer



## Was heißt eigentlich Korrektheit?

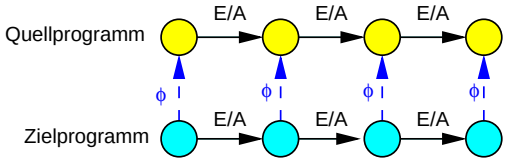
**Determinismus von Programmiersprachen**

Sprachstandards enthalten üblicherweise Freiheitsgrade bei Implementierung

- ☞ Meist bei Ausdrucksauswertung
- In C sind Zuweisungen Ausdrücke
- Aktualisierung des Speichers durch Zuweisung kann zwischen Sequenzpunkten in beliebiger Reihenfolge ausgeführt werden
- ⇒ Übersetzung muss nur eine dieser Reihenfolge implementieren

**Korrekte Übersetzung eines Quellprogramms $P$ in Zielprogramm $Z$**

Jedes beobachtbare Verhalten von $Z$ entspricht einem beobachtbaren Verhalten von $P$ bis auf Verletzung von Ressourcenbeschränkungen






## Diskussion

**Beispiel 1.2: Ein einfacher korrekter C-Übersetzer**

produziert immer folgendes Zielprogramm:

```
int main(void) {
  printf("segmentation fault (core dumped)");
}
```

- Korrekt, aber nutzlos

**Nützliche korrekte Übersetzer**

- Keine a priori Festlegung der Ressourcen möglich
- ⇒ Ingenieursaufgabe

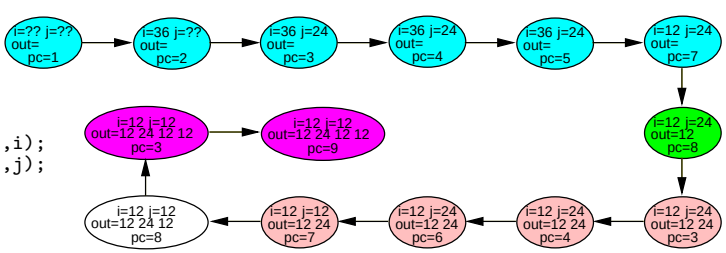
## Formale Definition der Korrektheit: Zustandsübergangssysteme

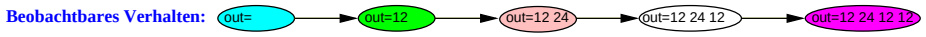
**Definition 1.1 (Zustandsübergangssystem und Lauf)**

Ein **Zustandsübergangssystem** ist ein Tripel $\mathcal{S} \triangleq (Q, I, \rightarrow)$, wobei $Q$ eine Menge von **Zuständen**, $I \subseteq Q$ eine Menge von **Initialzuständen** und $\rightarrow \subseteq Q \times Q$ eine **Zustandsübergangsrelation** ist. Ein Zustand $q' \in Q$ heißt **Nachfolgezustand** eines Zustands $q \in Q$ gdw. $q \rightarrow q'$. Ein Zustand $f \in Q$ heißt **final** gdw. es keinen Nachfolgezustand zu $f$ gibt.
Ein **Lauf** (engl. *run*) ist eine endliche oder unendliche Folge $\langle q_0, q_1, \ldots, q_n, \ldots \rangle$, so dass $q_i \rightarrow q_{i+1}$ für alle $i \geq 0$ gilt. Der **Lauf** heißt **vollständig** gdw. $q_0 \in I$ und der Lauf unendlich oder der letzte Zustand im Lauf final ist.

**Beispiel 1.3: Vollständiger Lauf eines Programms**

```
(1) int i=24;
(2) int j=36;
(3) while (i!=j) {
(4)    if (i>j)
(5)       i=i-j;
(6)    else j=j-i;
(7)    printf("%d ",i);
(8)    printf("%d ",j);
(9) }
```





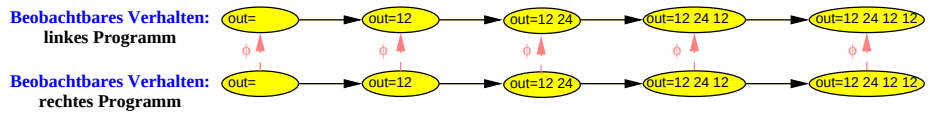

## Formale Definition der Korrektheit: Erhaltung beobachtbaren Verhaltens

**Definition 1.3 (1-1-Simulation)**

Sei $\mathcal{S}_1 \triangleq (Q_1, I_1, \rightarrow_1)$ ein Zustandsübergangssystem. Ein Zustandsübergangssystem $\mathcal{S}_2 \triangleq (Q_2, I_2, \rightarrow_2)$ **1-1-simuliert** $\mathcal{S}_1$ gdw. es eine Funktion $\phi : Q_1 \rightarrow Q_2$ gibt, so dass für jeden vollständigen Lauf $\langle q_0, q_1, \ldots, q_{i-1}, q_i, \ldots \rangle$ von $\mathcal{S}_2$ die Folge $\langle \phi(q_0), \phi(q_1), \ldots, \phi(q_{i-1}), \phi(q_i), \ldots \rangle$ ein vollständiger Lauf von $\mathcal{S}_1$ ist. (**Notation:** $\mathcal{S}_2 \precsim \mathcal{S}_1$)

**Beispiel 1.4: 1-1-Simulation**

```
(1) int i=24;                (1) printf("24 ");
(2) int j=36;                (2) printf("12 ");
(3) while (i!=j) {           (3) printf("12 ");
(4)    if (i>j) i=i-j;       (4) printf("12 ");
(5)    else j=j-i;
(6)    printf("%d ",i);
(7)    printf("%d ",j);
(8) }
```




**Beobachtung**

Die Erhaltung beobachtbaren Verhaltens ist eine 1-1-Simulation.

**Lemma 1.1 (Hinreichende Bedingung für 1-1-Simulation)**

Sei $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$ zwei Zustandsübergangssysteme. Falls es eine Funktion $\phi : Q_1 \rightarrow Q_2$ gibt, so dass $\phi(q) \rightarrow_2 \phi(q')$ für alle $q \rightarrow_1 q'$ gilt, dann gilt $\mathcal{S}_1 \precsim \mathcal{S}_2$.



## Formale Definition der Korrektheit: Beobachtbares Verhalten

**Definition 1.2 (Abstraktion)**

Sei $\mathcal{S}_1 \triangleq (Q_1, I_1, \rightarrow_1)$ ein Zustandsübergangssystem. Ein Zustandsübergangssystem $\mathcal{S}_2 \triangleq (Q_2, I_2, \rightarrow_2)$ heißt **Abstraktion** von $\mathcal{S}_1$ gdw. es eine Funktion $\phi : Q_1 \rightarrow Q_2$ mit den folgenden Eigenschaften gibt:
**i.** $\phi(I_1) \subseteq I_2$
**ii.** Falls $q_1 \rightarrow_1 q_1'$, dann ist entweder $\phi(q_1) = \phi(q_1')$ oder $\phi(q_1) \rightarrow_2 \phi(q_1')$

**Beobachtung**

Das beobachtbare Verhalten ist eine Abstraktion des Zustandsübergangssystems eines Programms



## Formale Definition der Korrektheit: Ressourcenbeschränkungen

**Beobachtung**

- Ressourcenbeschränkungen bedeuten, dass nicht jeder Zustand des "unten" Zustandsübergangssystems auf das zu 1-1-simulierende "obere" Zustandsübergangssystem abgebildet werden kann.

⇒ Abbildungsfunktion ist partiell

- Programme brechen bei Verletzung der Ressourcenbeschränkung ab

⇒ Zustände, die nicht abgebildet werden können, sind Finalzustände

**Definition 1.3 (Eingeschränkte 1-1-Simulation)**

Sei $\mathcal{S}_1 \triangleq (Q_1, I_1, \rightarrow_1)$ ein Zustandsübergangssystem. Ein Zustandsübergangssystem $\mathcal{S}_2 \triangleq (Q_2, I_2, \rightarrow_2)$ **1-1-simuliert eingeschränkt** $\mathcal{S}_1$ gdw. es eine partielle Funktion $\phi : Q_1 \rightarrowtail Q_2$ gibt, so dass für jeden vollständigen Lauf $\langle q_0, q_1, \ldots, q_{i-1}, q_i, \ldots \rangle$ von $\mathcal{S}_2$ eine der beiden folgenden Eigenschaften gilt:
**i.** Die Folge $\langle \phi(q_0), \phi(q_1), \ldots, \phi(q_{i-1}), \phi(q_i), \ldots \rangle$ ist vollständiger Lauf von $\mathcal{S}_1$.
**ii.** Der vollständige Lauf $\langle q_0, q_1, \ldots, q_n \rangle$ von $\mathcal{S}_1$ ist endlich, $q_n$ ist nicht im Definitionsbereich von $\phi$, $q_0, \ldots, q_{n-1}$ ist im Definitionsbereich von $\phi$ und $\langle \phi(q_0), \ldots, \phi(q_{n-1}) \rangle$ ist echter Präfix eines vollständigen Laufs von $\mathcal{S}_2$.
**Notation:** $\mathcal{S}_2 \precnsim \mathcal{S}_1$

## Formale Definition der Korrektheit: Zusammenfassung

**Beobachtung**

Erhaltung beobachtbaren Verhaltens bedeutet eingeschränkte 1-1-Simulation

- Semantik von Programmiersprachen ordnen jedem Programm $\pi$ ein Zustandsübergangssystem $[\![\pi]\!]$ zu.
- Beobachtbares Verhalten $[\![\pi]\!]'$ ist eine Abstraktion von $[\![\pi]\!]$ (**beobachtbare Semantik**)

☞ Zur Definition der Korrektheit von Übersetzern genügt beobachtbare Semantik

- Falls das Programm anhält ist auch der Finalzustand beobachtbar.

**Definition 1.4 (Korrekte Übersetzung, korrekte Übersetzer)**

Sei $Q$ eine Programmiersprache mit beobachtbarer Semantik $[\![\cdot]\!]'_Q$ und $Z$ eine Programmiersprache mit beobachtbarer Semantik $[\![\cdot]\!]'_Z$. Ein Programm $\pi_Z$ der Programmiersprache $Z$ heißt **korrekte Übersetzung** eines Programms $\pi_Q$ der Programmiersprache $Q$ gdw. $[\![\pi_Z]\!]'_Z \precsim\approx [\![\pi_Q]\!]'_Q$ ist.
Ein Übersetzer $\mathfrak{C}$ von $Q$ nach $Z$ heißt **korrekt** gdw. jedes übersetzte Programm $\mathfrak{C}(\pi)$ eine korrekte Übersetzung von $\pi$ ist.



## Diskussion

- Ein korrekter Übersetzer muss nicht jedes Quellprogramm übersetzen können
- Nur für den Fall, dass er tatsächlich ein Zielprogramm definiert, muss sichergestellt werden, dass das beobachtbare Verhalten bis auf Ressourcenbeschränkungen erhalten bleibt
- Die Zuordnung der Semantik und die Abstraktion zum beobachtbaren Verhalten ist zentrale Voraussetzung zum Nachweis der Übersetzerkorrektheit
- Für die bisherigen Überlegungen (und die weiteren in diesem einleitenden Kapitel) genügt es nur zu wissen, dass die Semantik einer Programmiersprache jedem Programm ein Zustandsübergangssystem zuordnet.



## Verwandte Anwendungen

**Codegeneratoren für speicherprogrammierbare Steuerungen**

- Übersetzen Zeitautomaten nach Structured Text
- Zielprogramme dürfen nicht wegen Ressourcenverletzungen abbrechen
- Zeiteigenschaften (Uhrenstände) müssen erhalten bleiben

**Codegeneratoren im Automobilbau**

- Modelle arbeiten mit kontinuierlichen Werten und Uhren
- Steuerung arbeiten mit diskreten Werten und Zeitzuständen

$\Rightarrow$ Keine exakte Zuordnung möglich

$\Rightarrow$ Anwendungsspezifische Toleranzgrenzen



## 1.3 Architektur verifizierter Übersetzer

**Grundidee**

Zerlege Übersetzung einer Quellsprache $Q$ in eine Zielsprache $Z$ in mehrere Übersetzungen sehr nahe beianderliegender Zwischensprachen:
$Q \Mapsto IL_1 \Mapsto IL_2 \mapsto \cdots \mapsto IL_{n-1} \Mapsto Z$

- Naheliegend: Verwende die typischen Zwischensprachen in einem Übersetzer
- Beweistechnisch kann es sinnvoll sein noch zusätzliche Zwischensprachen einzuführen

## Horizontale Dekomposition

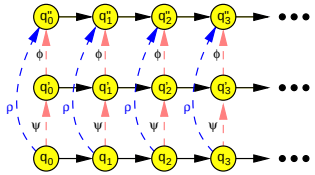
**Satz 1.2 (Horizontale Dekomposition)**

Die beschränkte 1-1-Simulation $\precsim$ auf Zustandsübergangssystemen ist transitiv: Wenn $\mathcal{S}_3 \precsim \mathcal{S}_2$ und $\mathcal{S}_2 \precsim \mathcal{S}_1$, dann gilt auch $\mathcal{S}_3 \precsim \mathcal{S}_1$, wobei $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2, 3$ Zustandsübergangssysteme sind.

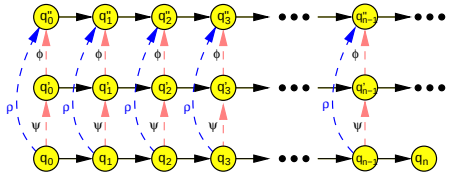
**Beweisskizze**

Sei $\mathcal{S}_3 \precsim \mathcal{S}_2$ mit $\psi : Q_3 \rightarrow Q_2$ und $\mathcal{S}_2 \precsim \mathcal{S}_1$ mit $\phi : Q_2 \rightarrow Q_1$.
**Behauptung:** $\mathcal{S}_3 \precsim \mathcal{S}_1$ mit $\rho \triangleq \phi \circ \psi$

**1. Fall:** $q_i \in \mathrm{DOM}(\psi)$ und $\psi(q_i) \in \mathrm{DOM}(\phi)$ für alle $i$



**2. Fall:** Lauf für $\mathcal{S}_3$ endlich, $q_n \notin \mathrm{DOM}(\psi)$, $q_0, \ldots, q_{n-1} \in \mathrm{DOM}(\psi)$ und $\psi(q_0), \ldots, \psi(q_{n-1}) \in \mathrm{DOM}(\phi)$. Also: $q_0, \ldots, q_{n-1} \in \mathrm{DOM}(\rho)$ und $q_n \notin \mathrm{DOM}(\rho)$



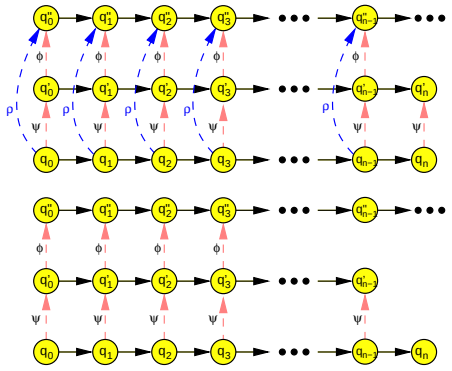
## Horizontale Dekomposition (Forts.)

**Beweisskizze**

**3. Fall:** Lauf für $\mathcal{S}_3$ endlich, $q_0, \ldots, q_n \in \mathrm{DOM}(\psi)$ und $\psi(q_0), \ldots, \psi(q_{n-1}) \in \mathrm{DOM}(\phi)$ und $\psi(q_n) \notin \mathrm{DOM}(\phi)$. Also: $q_0, \ldots, q_{n-1} \in \mathrm{DOM}(\rho)$ und $q_n \notin \mathrm{DOM}(\rho)$

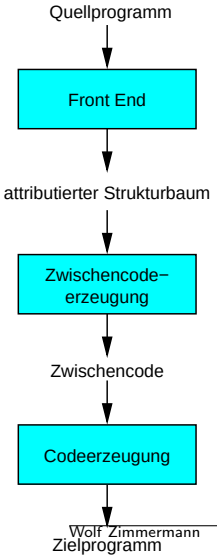
**4. Fall:** Lauf für $\mathcal{S}_3$ endlich, $q_n \notin \mathrm{DOM}(\psi)$, $q_0, \ldots, q_{n-1} \in \mathrm{DOM}(\psi)$, $\psi(q_0), \ldots, \psi(q_{n-2}) \in \mathrm{DOM}(\phi)$ und $\psi(q_{n-1}) \notin \mathrm{DOM}(\phi)$.

- $\Rightarrow$ $\psi(q_{n-1})$ ist final
- $\Rightarrow$ $\langle \psi(q_0), \ldots, \psi(q_{n-1})$ kein **echter** Präfix eines Laufs von $\mathcal{S}_2$
- $\Rightarrow$ $\mathcal{S}_3 \not\precsim \mathcal{S}_2$ **Widerspruch!**





## Architektur korrekter Übersetzer




### Fazit

Für verifizierte Übersetzer kann die klassische Übersetzerbauarchitektur verwendet werden.

- $\Rightarrow$ Keinerlei Einschränkungen in der Konstruktion
- $\Rightarrow$ Korrektheitsbegriff erlaubt Optimierungen
- $\Rightarrow$ Prinzipiell können verifizierte Übersetzer Code mit derselben Qualität erzeugen wie unverifizierte Übersetzer.



## Grundsätzliche Vorgehensweise zur Verifikation von Übersetzern

**Beweisverpflichtungen**

- Korrektheit der Transformationsregeln in Übersetzungsspezifikation
- Korrektheit der Implementierung des Übersetzers (z.B in C)
- Korrektheit der Implementierung des Übersetzers in Binärcode

## Korrektheit der Transformationsregeln

**Beispiel 1.5: Transformationsregel für Schleife**

$\mathcal{S}[\![\mathit{while}(\mathit{expr}, \mathit{stats})]\!] =$

$L_1$ : $\mathcal{S}[\![\mathit{stats}]\!]$
$L_2$ : $\mathcal{E}[\![\mathit{expr}]\!]$
**if** $\mathit{expr.val}$ **then** $L_1$ **else** $L_3$
$L_3$ :

**goto** $L_2$

- $\mathcal{S}$ ist Transformation für Anweisungen
- $\mathcal{E}$ ist Transformation für Ausdrücke
- $\mathit{expr.val}$ ist Register, das den Wert des ausgewerteten Ausdrucks enthält

☞ Jeder Ausdruck bekommt eindeutiges Register, das bei der Attributierung berechnet wird

**Beobachtung**

Transformationsregeln sind Schemata zur Übersetzung von Programmfragmenten

- Direkter Nachweis über beobachtbares Verhalten praktisch unmöglich

⇒ Direkter Nachweis über die Semantik

☞ Simulationsbeweise analog der Berechenbarkeits– oder Komplexitätstheorie



## Simulationsbeweise

**Ausgangspunkt**

Semantik von Quell- und Zielsprache als Zustandsübergangssysteme $\mathcal{S}_i = (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$.

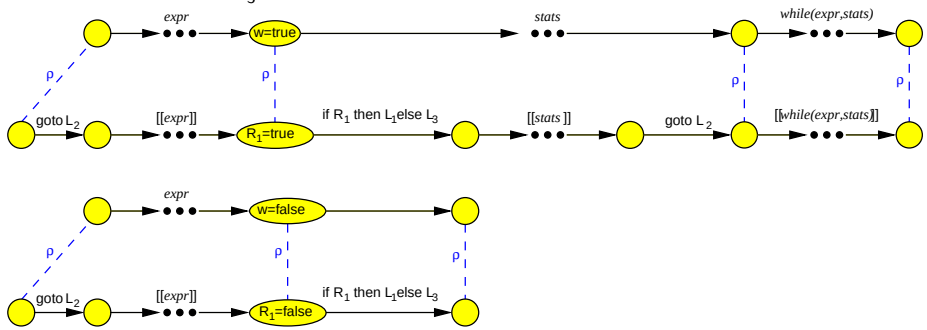
- Relation $\rho \subseteq Q_2 \times Q_1$ an den Grenzen der Transformationsregeln
- Beobachtbares Verhalten muss erhalten bleiben
- Einzelne Simulationen müssen zusammengesetzt werden

**Beispiel 1.5: Transformationsregel für Schleife**

$\mathcal{S}[\![\mathit{while}(\mathit{expr}, \mathit{stats})]\!] =$

$L_1$ : $\mathcal{S}[\![\mathit{stats}]\!]$
$L_2$ : $\mathcal{E}[\![\mathit{expr}]\!]$
**if** $\mathit{expr.val}$ **then** $L_1$ **else** $L_3$
$L_3$ :

**goto** $L_2$






## Vertikale Dekomposition

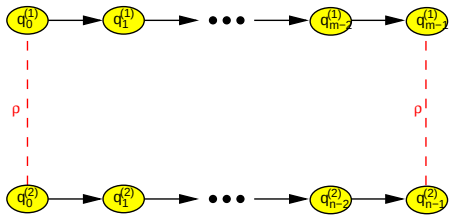
**Definition 1.5 (Urbild– und Bildbereich von Relationen)**

Sei $\rho \subseteq U_1 \times U_2$ eine Relation. Die Menge $\mathrm{DOM}(\rho) \triangleq \{u \in U_1 : \exists u' \in U_2 \bullet (u, u') \in \rho\}$ heißt **Urbildbereich von** $\rho$. Die Menge $\mathrm{RAN}(\rho) \triangleq \{u \in U_2 : \exists u' \in U_1 \bullet (u', u) \in \rho\}$ heißt **Bildbereich von** $\rho$.

**Was muss gelten?**

Gegeben seien zwei Zustandsübergangssysteme $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$ mit jeweils Abstraktionen $\alpha_i$ zu beobachtbarem Verhalten $\mathcal{S}_i' \triangleq (Q_i', I_i', \rightarrow_i')$, $i = 1, 2$.
**Ziel:** Definition einer Simulationsbeziehung zwischen $\mathcal{S}_2$ und $\mathcal{S}_1$, so dass $\mathcal{S}_2' \precsim \mathcal{S}_1'$ mit Funktion $\phi$.
**Vorgehen:** Definition einer Relation $\rho \subseteq Q_2 \times Q_1$ und Zusammensetzen von "Schnappschüssen":

- Lauf $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ von $\mathcal{S}_2$ mit höchstens einem beobachtbaren Zustandsübergang wird auf Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ mit höchstens einem beobachtbaren Zustandsübergang abgebildet
- $q_0^{(2)}, q_{n-1}^{(2)} \in \mathrm{DOM}(\rho)$ sowie $q_i^{(2)} \notin \mathrm{DOM}(\rho)$ für $i = 1, \ldots, n-2$
- $q_0^{(1)}, q_{m-1}^{(1)} \in \mathrm{RAN}(\rho)$ sowie $q_i^{(1)} \notin \mathrm{RAN}(\rho)$ für $i = 1, \ldots, m-2$

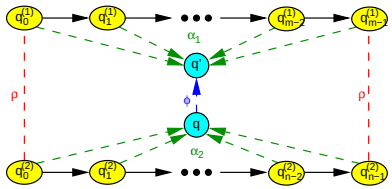





## Vertikale Dekomposition (Forts.)

**1. Fall: Lauf $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ hat keinen beobachtbaren Zustandsübergang.**

Dann hat Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ ebenfalls keinen beobachtbaren Zustandsübergang, $\alpha_2(q_i^{(2)}) = \alpha_2(q_0^{(2)})$ für alle $i = 1, \ldots, n-1$ und $\alpha_1(q_j^{(1)}) = \phi(\alpha_2(q_0^{(2)}))$ für alle $j = 0, \ldots, m-1$

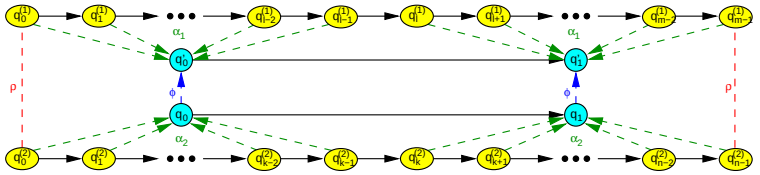


## Vertikale Dekomposition (Forts.)

**2. Fall:** $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ **hat einen beobachtbaren Zustandsübergang** $q_{k-1}^{(2)} \rightarrow_2 q_k^{(2)}$, $q_{n-1}^{(2)} \in \mathrm{DOM}(\rho)$

Dann hat Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ ebenfalls einen beobachtbaren Zustandsübergang $q_{l-1}^{(2)} \rightarrow_2 q_l^{(2)}$ und

$\alpha_2(q_i^{(2)}) = \alpha_2(q_0^{(2)})$ für $i = 0, \ldots, k-1$, $\alpha_2(q_i^{(2)}) = \alpha_2(q_k^{(2)})$ für $i = k+1, \ldots, n-1$,
$\alpha_1(q_j^{(1)}) = \phi(\alpha_2(q_0^{(2)}))$ für $j = 0, \ldots, l-1$ und $\alpha_1(q_j^{(1)}) = \phi(\alpha_2(q_k^{(2)}))$ für $j = k, \ldots, m-1$.

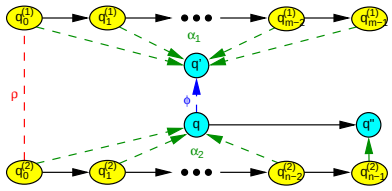




## Vertikale Dekomposition (Forts.)

**3. Fall: Lauf** $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ **hat einen beobachtbaren Zustandsübergang und** $q_{n-1}^{(2)}$ **ist final**

Dann kann auch Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ ohne beobachtbaren Zustandsübergang sein, $\alpha_2(q_i^{(2)}) = \alpha_2(q_0^{(2)})$ für $i = 1, \ldots, n-2$, $\alpha_1(q_j^{(1)}) = \phi(\alpha_2(q_0^{(2)}))$ für $j = 0, \ldots, m-1$ und $\alpha_2(q_{n-1}^{(2)}) \notin \mathrm{DOM}(\phi)$.





## Vertikale Dekomposition: *n*-*m*-Simulation

**Definition 1.6 (*n*-*m*-Simulation)**

Seien $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$, zwei Zustandsübergangssysteme mit jeweils Abstraktionen $\alpha_i$ zu beobachtbarem Verhalten $\mathcal{S}_i' \triangleq (Q_i', I_i', \rightarrow_i')$. Eine Relation $\rho \subseteq Q_2 \times Q_1$ definiert eine ***n*-*m*-Simulation** gdw. $I_2 \subseteq \mathrm{DOM}(\rho)$, $\rho(I_2) \subseteq I_1$ und für alle Läufe $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ von $\mathcal{S}_2$ mit $q_0^{(2)} \in \mathrm{DOM}(\rho)$, $q_{n-1}^{(2)} \in \mathrm{DOM}(\rho)$ bzw. $q_{n-1}^{(2)}$ final sowie $q_i^{(2)} \notin \mathrm{DOM}(\rho)$ für $i = 1, \ldots, n-2$ existiert ein Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ mit $(q_0^2, q_0^{(1)}) \in \rho$ und $q_i^{(1)} \notin \mathrm{RAN}(\rho)$ für $i = 1, \ldots, m-2$, so dass eine der folgenden Bedingungen erfüllt ist:

**i.** Weder in $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ noch in $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ gibt es einen beobachtbaren Zustandsübergang, $(q_{n-1}^{(2)}, q_{m-1}^{(1)}) \in \rho$, $\alpha_2(q_i^{(2)}) = \alpha_2(q_0^{(2)})$ für alle $i = 1, \ldots, n-1$ und $\alpha_1(q_j^{(1)}) = \alpha_2(q_0^{(1)})$ für alle $j = 0, \ldots, m-1$

**ii.** In $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ gibt es einen beobachtbaren Zustandsübergang $q_{k-1}^{(2)} \rightarrow_2 q_k^{(2)}$, in $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ gibt es einen beobachtbaren Zustandsübergang $q_{l-1}^{(1)} \rightarrow_2 q_l^{(1)}$, $(q_{n-1}^{(2)}, q_{m-1}^{(1)}) \in \rho$, $\alpha_2(q_i^{(2)}) = \alpha_2(q_0^{(1)})$ für $i = 0, \ldots, k-1$, $\alpha_2(q_i^{(2)}) = \alpha_2(q_k^{(2)})$ für $i = k+1, \ldots, n-1$, $\alpha_1(q_j^{(1)}) = \alpha_2(q_l^{(1)})$ für $j = 0, \ldots, l-1$ und $\alpha_1(q_j^{(1)}) = \phi(\alpha_2(q_k^{(2)}))$für $j = k, \ldots, m-1$.

**iii.** In $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ gibt es den beobachtbaren Zustandsübergang $q_{n-2}^{(2)} \rightarrow_2 q_{n-1}^{(2)}$, $q_{n-1}^{(2)}$ ist final, $q_{n-1}^{(2)} \notin \mathrm{DOM}(\rho)$, $\alpha_2(q_i^{(2)}) = \alpha_2(q_0^{(2)})$, $\alpha_2(q_{n-1}^{(2)}) \notin \mathrm{DOM}(\phi)$ und $\alpha_1(q_j^{(1)}) = \alpha_2(q_0^{(1)})$ für $j = 0, \ldots, m-1$.



**Satz 1.3 (*n*-*m*-Simulation und eingeschränkte 1-1-Simulation, Gaul/Zimmermann 1997)**

Seien $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$, zwei Zustandsübergangssysteme mit jeweils Abstraktionen $\alpha_i$ zu beobachtbarem Verhalten $\mathcal{S}_i' \triangleq (Q_i', I_i', \rightarrow_i')$ und $\rho \subseteq Q_2 \times Q_1$ eine *n*-*m*-Simulation mit den folgenden Eigenschaften:

**i.** $\phi \triangleq \alpha_1 \circ \rho \circ \alpha_2^{-1} \subseteq Q_2' \times Q_1'$ ist eine injektive partielle Funktion

**ii.** Für jeden unendlichen Lauf $\langle q_i^{(2)} : i \in \mathbb{N} \rangle$ von $\mathcal{S}_2$ mit $q_0^{(2)} \in \mathrm{DOM}(\rho)$ existiert ein unendlicher Lauf $\langle q_i^{(1)} : i \in \mathbb{N} \rangle$ von $\mathcal{S}_1$ mit $(q_0^{(2)}, q_0^{(1)}) \in \rho$ und entweder $q_i^{(2)} \notin \mathrm{DOM}(\rho)$ für $i > 0$, $q_j^{(2)} \notin \mathrm{RAN}(\rho)$ für $j > 0$ oder $(q_i^{(2)}, q_j^{(2)}) \in \rho$ für $i, j > 0$

**iii.** Für jeden Finalzustand $q^{(2)} \in Q_2$ ist jedes $q^{(1)} \in \rho(q^{(2)})$ final in $\mathcal{S}_1$

Dann gilt $\mathcal{S}_2' \precsim \mathcal{S}_1'$.

**Beweis (Skizze)**

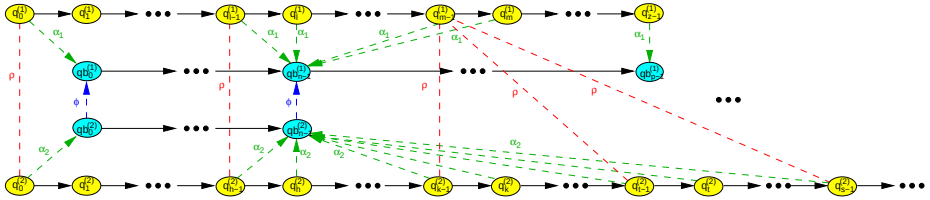
**Vorgehen:**
- Zeige durch Induktion über die Länge der Läufe von $\mathcal{S}_2$, dass mit $\phi$ die Bedingungen für $\mathcal{S}_2' \precsim \mathcal{S}_1'$ für jeden Präfix $\langle qb_0^{(2)}, \ldots, qb_{n-1}^{(2)} \rangle$ eines vollständigen Laufs von $\mathcal{S}_2'$ erfüllt sind
- Zeige durch noethersche Induktion, dass dann auch die Bedingungen für $\mathcal{S}_2' \precsim \mathcal{S}_1'$ für alle vollständigen Läufe von $\mathcal{S}_2'$ erfüllt sind.

**Behauptung 1:** Für jeden Lauf $\langle q_0^{(2)}, \ldots, q_{k-1}^{(2)} \rangle$ von $\mathcal{S}_2$ mit $q_0^{(2)} \in I_2$ und Abstraktion $\langle qb_0^{(2)}, \ldots, qb_{n-1}^{(2)} \rangle$, $q_{k-1}^{(2)} \in \mathrm{DOM}(\rho)$, gibt es einen Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ von $\mathcal{S}_1$ mit $q_0^{(1)} \in I_1$ und Abstraktion $\langle \phi(qb_0^{(2)}), \ldots, \phi(qb_{n-1}^{(2)}) \rangle$

**Behauptung 2:** Die Bedingungen für $\mathcal{S}_2' \precsim \mathcal{S}_1'$ sind für jeden Präfix $\langle qb_0^{(2)}, \ldots, qb_{n-1}^{(2)} \rangle$ eines vollständigen Laufs von $\mathcal{S}_2'$ erfüllt

## Diskussion: Ausschluss der magischen Unendlichkeit

Die Bedingung (ii) schließt die folgende Situation aus:

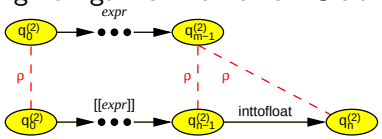




## 0-$m$- und $n$-0-Simulationen

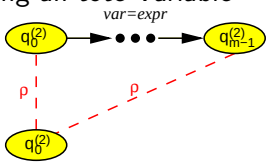
**Beispiel 1.6:** 0-$m$-**Simulation**

Implizite Anpassung von ganzen Zahlen an Gleitkommazahlen




**Beispiel 1.7:** $n$-0-**Simulation**

Streichen von Zuweisung an tote Variable





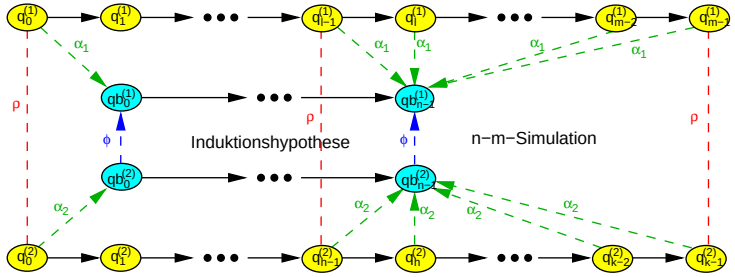

**Lemma 1.4 (Endliche Präfixe)**

Seien $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$, zwei Zustandsübergangssysteme mit jeweils Abstraktionen $\alpha_i$ zu beobachtbarem Verhalten $\mathcal{S}_i' \triangleq (Q_i', I_i', \rightarrow_i')$ und $\rho \subseteq Q_2 \times Q_1$ eine $n$-$m$-Simulation, so dass $\phi \triangleq \alpha_1 \circ \rho \circ \alpha_2^{-1} \subseteq Q_2' \times Q_1'$ eine injektive partielle Funktion ist. Für jeden Lauf $\langle q_0^{(2)}, \ldots, q_{k-1}^{(2)} \rangle$ von $\mathcal{S}_2$ mit $q_0^{(2)} \in I_2$, $q_{k-1}^{(2)} \in \mathrm{DOM}(\rho)$ und Abstraktion $\langle qb_0^{(2)}, \ldots, qb_{n-1}^{(2)} \rangle$ gibt es einen Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ von $\mathcal{S}_1$ mit $q_0^{(1)} \in I_1$, Abstraktion $\langle \phi(qb_0^{(2)}), \ldots, \phi(qb_{n-1}^{(2)}) \rangle$ und $(q_0^{(2)}, q_0^{(1)}), (q_{k-1}^{(2)}, q_{m-1}^{(1)}) \in \rho$.

**Beweis durch Induktion über die Länge $k$**

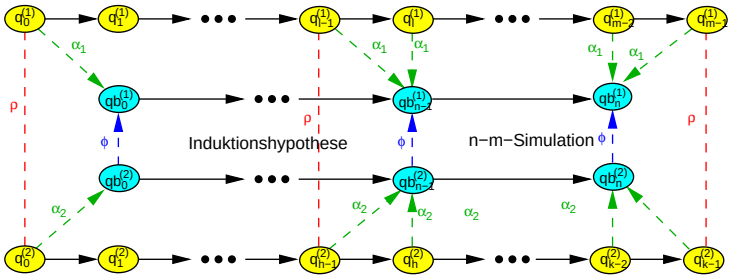
**Induktionsanfang:** $k = 0$. Dann ist $q_0^{(2)} \in I_2$. Wähle ein $q_0^{(1)} \in I_1$, so dass $(q_0^{(2)}, q_0^{(1)}) \in \rho$. Nach Definition ist $\alpha_1(q_0^{(1)}) = \phi(\alpha_2(q_0^{(2)}))$

**Induktionsschritt** Da $q_0^{(2)} \in \mathrm{DOM}(\rho)$ ist, existiert ein $h < k$, so dass $q_{h-1}^{(2)} \in \mathrm{DOM}(\rho)$ und $q_j^{(2)} \notin \mathrm{DOM}(\rho)$ für $j = h, \ldots, k-2$.






## Fortsetzung Beweis Lemma 4



## Beweisskizze von Satz 1.3

**Lemma 1.5 (Unendliche Läufe)**

Seien $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$, zwei Zustandsübergangssysteme mit jeweils Abstraktionen $\alpha_i$ zu beobachtbarem Verhalten $\mathcal{S}'_i \triangleq (Q'_i, I'_i, \rightarrow'_i)$ und $\rho \subseteq Q_2 \times Q_1$ eine *n-m*-Simulation mit den folgenden Eigenschaften:

**i.** $\phi \triangleq \alpha_1 \circ \rho \circ \alpha_2^{-1} \subseteq Q'_2 \times Q'_1$ ist eine injektive partielle Funktion

**ii.** Für jeden unendlichen Lauf $\langle q_i^{(2)} : i \in \mathbb{N}\rangle$ von $\mathcal{S}_2$ mit $q_0^{(2)} \in \mathrm{DOM}(\rho)$ existiert ein unendlicher Lauf $\langle q_i^{(1)} : i \in \mathbb{N}\rangle$ von $\mathcal{S}_1$ mit $(q_0^{(2)}, q_0^{(1)}) \in \rho$ und entweder , $q_i^{(2)} \notin \mathrm{DOM}(\rho)$ für $i > 0$, $q_j^{(2)} \notin \mathrm{RAN}(\rho)$ für $j > 0$ oder $(q_i^{(2)}, q_j^{(2)}) \in \rho$ für $i, j > 0$

Dann gibt es zu jedem unendlichen Lauf $\langle q_i^{(2)} : i \in \mathbb{N}\rangle$ von $\mathcal{S}_2$ mit $q_0^{(2)} \in I_2$ einen unendlichen Lauf$\langle q_i^{(1)} : i \in \mathbb{N}\rangle$ von $\mathcal{S}_1$ mit $(q_0^{(2)}, q_0^{(1)}) \in \rho$, $q_0^{(1)} \in I_1$, so dass $\mathcal{S}'_2 \precsim \mathcal{S}'_1$.

**Beweis (Skizze)**

**1. Fall:** $q_i^{(2)} \in \mathrm{DOM}(\rho)$ für unendlich viele $i$

- Nach Lemma 1.4 sind die Bedingungen für $\mathcal{S}'_2 \precsim \mathcal{S}'_1$ für jeden endlichen Präfix $\langle q_i^{(2)} : i \in \mathbb{N}\rangle$ erfüllt.
- Die Relation *ist Präfix* ist eine vollständige Halbordnung.
- Die Bedingungen sind stetige Funktionen über dieser Halbordnung

$\Rightarrow$ Aussage folgt durch transfinite Induktion

**2. Fall:** $q_i^{(2)} \notin \mathrm{DOM}(\rho)$ für fast alle $i$. Dann folgt die Aussage direkt aus Lemma 1.4 (ii).



## Beweisskizze von Satz 1.3

**Lemma 1.6 (Endliche Läufe)**

Seien $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$, zwei Zustandsübergangssysteme mit jeweils Abstraktionen $\alpha_i$ zu beobachtbarem Verhalten $\mathcal{S}'_i \triangleq (Q'_i, I'_i, \rightarrow'_i)$ und $\rho \subseteq Q_2 \times Q_1$ eine *n-m*-Simulation mit den folgenden Eigenschaften:

**i.** $\phi \triangleq \alpha_1 \circ \rho \circ \alpha_2^{-1} \subseteq Q'_2 \times Q'_1$ ist eine injektive partielle Funktion

**ii.** Für jeden Finalzustand $q^{(2)} \in Q_2$ ist jedes $q^{(1)} \in \rho(q^{(2)})$ final in $\mathcal{S}_1$

Falls ein vollständiger endlicher Lauf $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)}\rangle$ von $\mathcal{S}_2$ mit $q_{n-1}^{(2)} \in \mathrm{DOM}(\rho)$ existiert, gibt es einen vollständigen endlichen Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)}\rangle$ mit $(q_0^{(2)}, q_0^{(1)}), \in (q_{n-1}^{(2)}, q_{m-1}^{(1)}) \in \rho$

**Beweis (Skizze)**

**Annahme:** Es gäbe $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)}\rangle$ von $\mathcal{S}_2$ mit $q_0^{(2)}, q_{n-1}^{(2)} \in \mathrm{DOM}(\rho)$, aber es gibt keinen vollständigen endlichen Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)}\rangle$ von $\mathcal{S}_1$ mit $(q_0^{(2)}, q_0^{(1)}), (q_{n-1}^{(2)}, q_{m-1}^{(1)}) \in \rho$.

- Sei $q_0^{(1)} \in \rho(q_0^{(2)})$. Nach Definition 1.6 ist $q_0^{(1)} \in I_1$.
- Aus Lemma 1.4 folgt, dass es einen Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)}\rangle$ von $\mathcal{S}_1$ mit $(q_{n-1}^{(2)}, q_{m-1}^{(1)}) \in \rho$ gibt.
- Mit (ii) folgt, dass $q_{m-1}^{(1)}$ final in $\mathcal{S}_1$ ist

$\Rightarrow$ $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)}\rangle$ ist vollständig **Widerspruch!**



## Programmprüfung und Validierung von Übersetzungen

### Ziel

Verifikation der Implementierung eines Übersetzers

### Problem

Praktisch nicht durchführbar

- Weder für generiertem noch manuell entwickeltem Übersetzer
- Auch bei Übersetzergeneratoren nicht machbar
- Ursache ist Umfang und Komplexität des Programms



## Programmprüfung und Validierung von Übersetzungen

**Satz 1.7 (Programmprüfung, Blum 1989)**

Sei `T f(U x)` eine unverifizierte Funktion mit Vorbedingung $P(x)$ und Nachbedingung $Q(x, f(x))$. Weiterhin sei `bool check_f(U x,T y)` ein verifiziertes Prädikat mit Vorbedingung $P(x)$ und Nachbedingung $res = true \Rightarrow Q(x, y)$. Dann gilt hat das folgende Programm bei Vorbedingung $P(x)$ die Nachbedingung $res \neq \text{fail} \Rightarrow Q(x, f(x))$:

```
T f'(U x) {
  T z = f(x);
  if (check_f(x,z)) return z;
  else return fail;
}
```

**Bemerkungen**

- Beweis kann direkt im Hoare-Kalkül geführt werden
- Korrektheit hängt nicht von der Korrektheit von `f`, sondern nur von der Korrektheit von `check_f(x,z)` ab
- Ist `f` ein Übersetzer, `x` Quellcode und `z` Zielcode spricht man von **Übersetzungsvalidierung** (engl. *translation validation*, Pnueli et. al. 1997)

☞ `check_f` ist bei Übersetzern wesentlich kleiner und einfacher als `f`.

## Korrektheit des Übersetzers in Binärcode

**Vorgehen: Bootstrapmethode (Goerigk, Langmaack et. al., 1998)**

- Entwickeln eines einfachen, gemäß den Transformationsregeln verifizierten Übersetzers $C$ für eine Sprache $\mathfrak{L}$ in Programmiersprache $\mathfrak{L}$
- Übersetzen von $C$ mit einem unverifizierten $\mathfrak{L}$-Übersetzer
- Manuelle Überprüfung, ob $C$ dadurch korrekt übersetzt wurde

☞ Geschickte Organisation erlaubt teilweise Automatisierung dieses Bootstraps.



## Korrektheit der semantischen Analyse

**Beobachtung**

Attribute enthalten semantische Informationen wie Typen, Zuordnung von Deklarationen

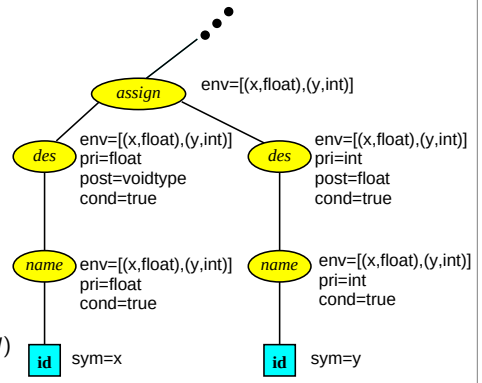
- Spezifikation z.B. durch attributierte Grammatik (nicht die, die im Übersetzer verwendet wird!)
- Überprüfung, ob Attribute gemäß Attributierungsregeln berechnet wurden

**Beispiel 1.8: Überprüfung der semantischen Analyse**

```
float x;
int y;
x=y;
```

**rule** $expr = \cdots | des | \cdots$
**rule** $assign ::= des\ expr$
**attr** $des.env \leftarrow assign.env$
$expr.env \leftarrow assign.env$
$des.post \leftarrow$ voidtype
$expr.post \leftarrow des.pri$
**rule** $des ::= name$
**attr** $name.env \leftarrow des.env$
$des.pri \leftarrow name.pri$
**cond** $des.pri \sqsubseteq des.post$
**rule** $name ::=$ **id**
**attr** $name.pri \leftarrow getType(\mathbf{id}.sym, name.env)$
**cond** $isDefined(\mathbf{id}.sym, name.env)$

$$
\begin{array}{lll}
 & getType(x, []) & = \text{errtype} \\
 & getType(x, (x, t) : l) & = t \\
x \neq y \Rightarrow & getType(x, (y, t) : l) & = getType(y, l) \\
 & isDefined(x, []) & = \text{false} \\
 & isDefinded(x, (x, t) : l) & = \text{true} \\
x \neq y \Rightarrow & isDefined(x, (y, t) : l) & = isDefined(y, l)
\end{array}
$$






## Korrektheit der Syntaxanalyse

**Beobachtungen**

- Definition der Semantik einer Programmiersprache setzt immer auf dem abstrakten Syntaxbaum oder dem attributierten Strukturbaum auf

☞ Insbesondere nie auf dem Quelltext einer Zeichenkette

⇒ Syntaxanalyse und semantische Analyse ist bereits durchgeführt

- Übersetzerbauer haben Freiheitsgrade beim Entwurf der abstrakten Syntax
- Syntaxanalyse transformiert einen Text in eine Baumstruktur

**Problem**

**Was ist dann Korrektheit der Syntaxanalyse?**

**Spezifikation der Syntaxanalyse**

**Relation** zwischen Quelltexten und abstrakten Syntaxbäumen

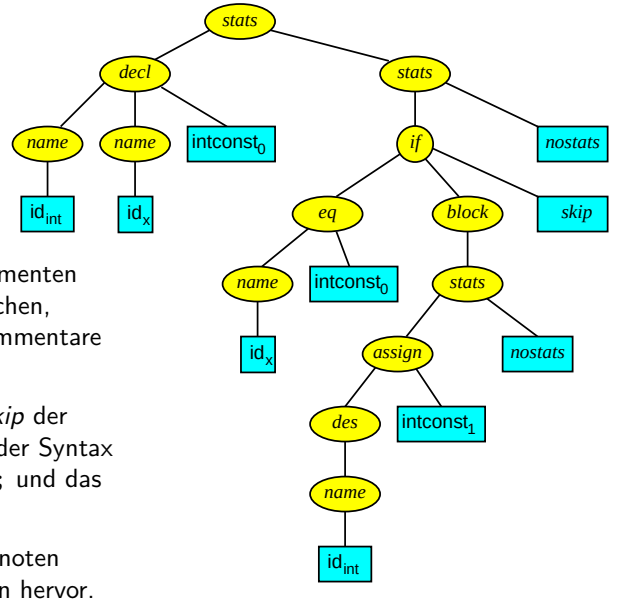
☞ Aus abstraktem Syntaxbaum ist der Quelltext bis auf Leerzeichen, Kommentare, überflüssiger Klammern etc. rekonstruierbar

⇒ Überprüfung der Korrektheit mit Programmprüfung



## Beispiel 1.9: Prüfung Syntaxanalyse

```
int x=0;// x-Koordinate
if (x==0){
  x=1;
}
```




- Zwischen zwei aufeinanderfolgenden Elementen dürfen höchstens Leerzeichen, Zeilenumbrüche oder Kommentare folgen
- Mit dem dritten Zweig *skip* der `if`-Anweisung gibt es in der Syntax die Möglichkeiten `else ;` und das Fehlen des `else`-Zweigs

☞ geht aus den dem *skip*-Knoten zugeordneten Koordinaten hervor.

## 1.4 Ausblick auf den Rest der Vorlesung

- Definition der Semantiken der Quellsprache, der Zwischen– und der Zielsprache auf Basis attributierte Strukturbäume

☞ Wir verwenden eine Teilmenge von C als Quellsprache und den Binärcode der DEC-Alpha als Zielsprache

- Ideal wäre, wenn der Zustandsraum bei einer Übersetzung erhalten bliebe

⇒ Erfüllt bei optimierenden Transformationen

⇒ Transformation des Zustandsraums ohne Änderung der Programmiersprache

☞ Speicherabbildung, Registerzuteilung

- Verifikation der Implementierung durch Überprüfung der korrekten Anwendung von Transformationsregeln
- Verallgemeinerung auf Übersetzungsvalidierung (engl. *Translation Validation*): Verfahren, die direkt die hinreichende Bedingungen für die eingeschränkte 1-1-Simulation überprüfen.



# Kapitel 2
# Grundlagen und Semantik von Programmiersprachen

**Wolf Zimmermann**

**Verifikation von Übersetzern**



## Inhalt

**Ziele**

- Kennenlernen der theoretischen Grundlagen
- Erwerben der Fähigkeit zur Formalisierung der Semantik von Programmiersprachen
- Umsetzen des Korrektheitsbegriffs auf formale Semantiken

1. Motivation
2. Signaturen, Terme, Algebren
3. Abstrakte Zustandsmaschinen
4. Modulare Konstruktion von Semantiken mit ASMs



## 2.1 Motivation

**Aufbau einer Programmiersprache**

Trennt Syntax und Semantik

**Syntax einer Programmiersprache**

Was ist wohlgeformtes Programm?

**Semantik einer Programmiersprache**

- Welche abstrakte Bedeutung hat ein wohlgeformtes Programm?
- Selbstbezug

**Pragmatik einer Programmiersprache**

- Welche konkrete Bedeutung hat ein wohlgeformtes Programm?
- Bezug auf Umwelt und Rechner

## Statische und dynamische Semantik

**Statische Semantik einer Programmiersprache**

- Definiert Eigenschaften von Programmen, die ohne Ausführung bestimmt werden können
  - ☞ Typisierung
  - ☞ Gültigkeitsbereich von Variablen

**Dynamische Semantik einer Programmiersprache**

- Ordnet Sprachelementen Bedeutung zu
- Definiert Ausführung des Programms



## Arten formaler Semantiken

**Denotationale Semantik**

Jedem Sprachelement wird eine (evtl. rekursiv) Zustandsübergangsfunktion zugeorndet

- Mathematische Basis: vollständige Halbordnungen, stetige Funktionen

**Strukturell operationale Semantik**

Jedem Sprachelement wird eine Zustandstransformation zugeordnet

- Mathematische Basis: Inferenzregeln

**Abstrakte Maschine**

- Jedem Sprachelement werden Befehlsfolgen (evtl. rekursiv) zugeordnet
- Jedem Befehl wird eine Zustandstransformation zugeordnet
- Basis: Algebren und Abstrakte Zustandsmaschinen



## Kriterien für die Übersetzungsverifikation

- Einheitliche Art der formalen Semantik für alle beteiligten Sprachen
- Modularer Aufbau der Sprachsemantiken ermöglicht Modularisierung der Korrektheitsbeweise
  - Formale Semantik für einen einfachen Sprachkern
  - Sukzessive Anreicherung um neue Sprachkonstrukte bis komplette Sprache erreicht wird **ohne** Änderung der bisher definierten Semantik der Teilsprachen
- Prozessorhandbücher ordnen jedem Maschinenbefehl Registertransfers zu

⇒ Operationale Semantik

- Zwischensprachsemantiken und Programmiersprachsemantiken lassen sich mit operationalen Semantiken mindestens so leicht formalisieren wie mit denotationalen Semantiken
- Abstrakte Maschinen sind für Maschinensprachen besser geeignet als strukturell operationale Semantiken
- Zwischensprachen und Programmiersprachsemantiken lassen sich modular über abstrakte Maschinen oft leichter aufbauen als strukturell operationale Semantiken



## Fazit und Ausblick auf den Rest des Kapitels

**Fazit**

Formalisierung der Semantiken der beteiligten Programmiersprachen über abstrakte Maschinen

**Durchführung**

Definition der abstrakten Maschinen mit Spezifikationsmethode **Abstrakte Zustandsmaschinen** (engl. *Abstract State Machines* oder kurz: ASM):

- Zustände sind $\Sigma$-Algebren über einer Signatur $\Sigma$
- Zustandsübergänge ändern die Interpretation von $\Sigma$-Termen

⇒ Abstrakte Zustandsmaschinen spezifizieren ein Zustandsübergangssystem

⇒ Begriffe der Korrektheit aus Kapitel 1 können direkt eingesetzt werden

⇒ Es genügt die Definition beobachtbaren Verhaltens und der Nachweis von $n$-$m$-Simulationen

## 2.2 Signaturen, Terme, Algebren

**Definition 2.1 (Signatur)**

Paar $\Sigma = (S, F)$ wobei $S$ Menge von **Sorten** ist und $F = (F_{w,s})_{w \in S^*, s \in S}$ Menge von **Operationssymbolen** ist.
**Notation:** $f : s_1 \times \cdots \times s_n \to s$ ist Operationssymbol der Stelligkeit $s_1 \cdots s_n, s$

**Definition 2.2 (Terme)**

Sei $\Sigma \triangleq (S, F)$ eine Signatur. Ein $\Sigma$-**Term der Sorte** $s$ **mit Variablen** $(X_s)_{s \in S}$ ist induktiv definiert ($X_s$ sind **Variablen der Sorte** $s$, $X_s \cap F_{\varepsilon,s} = \emptyset$):
**i.** Jede Variable $x \in X_s$ ist Term der Sorte $s$.
**ii.** Wenn $f : s_1 \times \cdots \times s_n \to s \in F$ und $t_1, \ldots, t_n$ Terme der Sorte $s_1, \ldots, s_n$ sind, dann ist $f(t_1, \ldots, t_n)$ Term der Sorte $s$ (Insbesondere auch für $n = 0$)
$\Sigma$-**Grundterm:** $\Sigma$-Term ohne Variablen

**Notationen**

- $T_s(\Sigma, X)$ ist die Menge aller $\Sigma$-Terme der Sorte $s$ mit Variablen $X$. $T(\Sigma, X) \triangleq \bigcup_{s \in S} T_s(\Sigma, X)$
- $T_s(\Sigma)$ ist die Menge aller $\Sigma$-Grundterme der Sorte $s$ $T(\Sigma) \triangleq \bigcup_{s \in S} T_s(\Sigma)$



## Beispiel 2.1: Signatur für die Abstrakte Syntax einer einfachen While-Sprache

**sig** WHILE

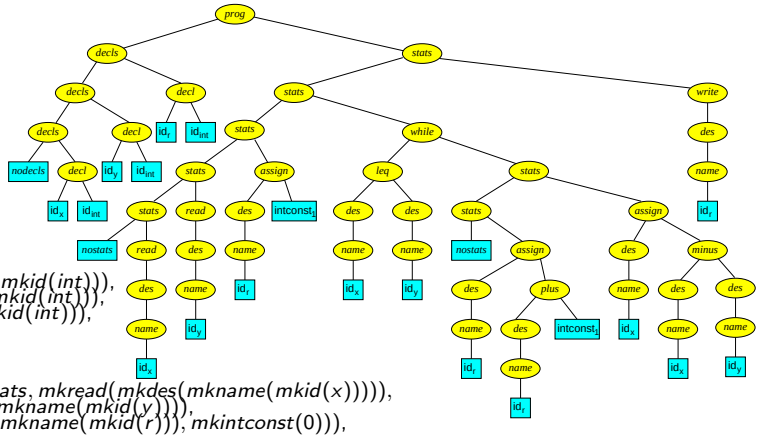
| | | | |
|---|---|---|---|
| **sorts** | ID, INTCONST, BOOLCONST, OP, PROG, STATS, DECLS, DECL, TYPE, STAT, EXPR | | |
| **operations** | *id* : | | → ID |
| | *z* : | | → INTCONST |
| | *true* : | | → BOOLCONST |
| | *false* : | | → BOOLCONST |
| | *plus*, *minus*, ... : | | → OP |
| | *type* : ID | | → TYPE |
| | *mkprog* : | DECLS × STATS | → PROG |
| | *mknodecls* : | | → DECLS |
| | *mkdecls* : | DECLS × DECL | → DECLS |
| | *mkdecl* : | ID × TYPE | → DECL |
| | *mknostats* : | | → STATS |
| | *mkstats* : | STATS × STAT | → STATS |
| | *mkassign* : | DES × EXPR | → STAT |
| | *mkif* : | EXPR × STATS × STATS | → STAT |
| | *mkwhile* : | EXPR × STATS | → STAT |
| | *mkread* : | DES | → STAT |
| | *mkwrite* : | EXPR | → STAT |
| | *mkdes* : | NAME | → DES |
| | *mkname* : | *ID* | → NAME |
| | *mkintconst* : | INTCONST | → EXPR |
| | *mkboolconst* : | INTCONST | → EXPR |
| | *mkgeq* : | EXPR × OP × EXPR | → EXPR |
| | *mkplus* : | EXPR × OP × EXPR | → EXPR |
| | *mkminus* : | EXPR × OP × EXPR | → EXPR |

wobei *id* beliebige Folge aus Buchstaben und Ziffern (beginnend mit Buchstabe) und *z* beliebige ganze Zahl in Dezimaldarstellung ist.



## Beispiel 2.1: Programme sind Terme der abstrakten Syntax

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```



```
mkprog(
 mkdecls(
  mkdecls(
   mkdecls(
    mknodecl,
    mkdecl(mkid(x), mkid(int))),
   mkdecl(mkid(y), mkid(int))),
  mkdecl(mkid(r), mkid(int))),
 mkstats(
  mkstats(
   mkstats(
    mkstats(
     mkstats(mknostats, mkread(mkdes(mkname(mkid(x))))),
     mkread(mkdes(mkname(mkid(y))))),
    mkassign(mkdes(mkname(mkid(r))), mkintconst(0))),
   mkwhile(
    mkgeq(mkdes(mkdes(mkname(mkid(x)))), mkdes(mkdes(mkname(mkid(y))))),
    mkstats(
     mkstats(
      mknostats,
      mkassign(
       mkdes(mkname(mkid(r))), mkplus(mkdes(mkdes(mkname(mkid(r)))),
       mkintconst(1)))),
     mkassign(
      mkdes(mkname(mkid(x))),
      mkminus(mkdes(mkname(mkid(x))), mkdes(mkdes(mkname(mkid(y)))))))))),
 mkwrite(mkdes(r)))))
```



## Diskussion

**Beobachtungen**

- Terme sind geeignet zur Darstellung abstrakter Syntaxbäume
- Nichtterminale *des*, *plus* etc. werden nur implizit über Konstruktoren als Ausdruck behandelt.
- Keine Behandlung von Alternativen
- Keine Navigation in abstrakten Syntaxbäumen

⇒ Statische Semantik ist nicht darstellbar

- Keine Interpretation von Termen

⇒ Auswertung von Ausdrücken, Zustandsraum nicht darstellbar

⇒ Zustandstransformation nicht darstellbar

**Lösungen**

- Interpretation von Termen durch $\Sigma$-Algebren
- Navigation durch Adressierung der AST-Knoten (*occurrences*)
- Einführung von Untersorten (erlaubt z.B. einheitliche Behandlung der unterschiedlichen Arten von Ausdrücken)
- Statische Semantik durch Bedingungen (bedingte Gleichungen, Formeln)
- Zustandstransformationen verändern Interpretation von Operationssymbolen (abstrakte Zustandsmaschinen)

## Algebren

**Definition 2.3 (Σ-Algebra)**

Sei $\Sigma = (S, F)$ eine mehrsortige Signatur mit Variablen $(X_s)_{s \in S}$. Eine Σ-Algebra ist ein Paar $\mathcal{A} = (A, \Phi)$ wobei

- $A = (A_s)_{s \in S}$ wobei $A_s$ eine Menge mit $X_s \subseteq A_s$ ist (**Trägermenge der Sorte** $s$)
- $\Phi = \{f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s | f : s_1 \times \cdots \times s_n \to s \in \Sigma\}$, d.h., jedem Operationssymbol $f$ der Stelligkeit $s_1 \cdots s_n, s$ wird eine Funktion $f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ zugeordnet



## Interpretation von Termen

**Definition 2.4 (Variablenbelegung, Interpretation)**

Sei $\Sigma \triangleq (S, F)$ eine mehrsortige Signatur mit Variablen $(X_s)_{s \in S}$ und $\mathcal{A} = (A, \Phi)$ eine Σ-Algebra. Eine **Variablenbelegung** ist eine Familie von Abbildungen $(\beta_s : X_s \to A_s)_{s \in S}$.
Die **Interpretation von Termen** $\beta$ **und** $\mathcal{A}$ ist eine Familie von Abbildungen $([\![\cdot]\!]^\beta_s : \mathsf{T}_s(\Sigma, X) \to A_s)_{s \in S}$, die wie folgt definiert ist:

i. $[\![x]\!]^\beta_s \triangleq \beta_s(x)$ für $x \in X_s$
ii. $[\![f]\!]^\beta_s \triangleq f_A$ für $f :\to s \in F$
iii. $[\![f(t_1, \ldots, t_n)]\!]^\beta_s \triangleq f_A([\![t_1]\!]^\beta_s, \ldots, [\![t_n]\!]^\beta_s)$ für $f : s_1 \times \cdots \times s_n \to s \in F$ und Terme $t_1 \in \mathsf{T}_{s_1}(\Sigma, X), \ldots, t_n \in \mathsf{T}_{s_n}(\Sigma, X)$

**Beobachtung**

Abstrakte Syntaxbäume können als Term interpretiert werden

- Algebra $\mathcal{T} = (\mathsf{T}_s(\Sigma), \iota)$ (**Termalgebra**)



## Beispiel 2.2: Algebra der ganzen Zahlen

**Signatur**

**sig** INT
**sorts** INT
**operations** $\ldots, -2, -1, 0, 1, 2, \ldots : \to \text{INT}$
$plus, mal, minus : \text{INT} \times \text{INT} \to \text{INT}$

INT-**Algebra**

$\mathfrak{Z} \triangleq \langle Z, \Phi_Z \rangle$

- Trägermenge $\text{INT}_Z \triangleq \mathbb{Z}$
- Intepretationen $\Phi$:
  $\ldots, -2_Z \triangleq -2, -1_Z \triangleq -1, 0_Z \triangleq 0, 1_Z \triangleq 1, 2_Z \triangleq 2, \ldots$
  $plus_Z \triangleq +$
  $mal_Z \triangleq *$
  $minus_Z \triangleq -$

**Interpretation eines Terms**

$[\![plus(mkintconst(3), mkintconst(5))]\!] = plus_Z([\![mkintconst(3)]\!], [\![mkintconst(5)]\!]) = [\![mkintconst(3)]\!] + [\![mkintconst(5)]\!] = 3 + 5 = 8$

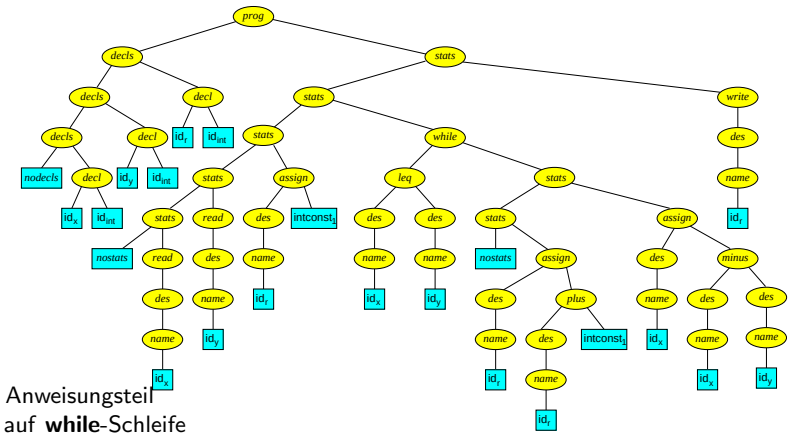


## Beispiel 2.3: Navigation in abstrakten Syntaxbäumen

**Idee**

Verwende Listen ganzer Zahlen zum Navigieren

- Leere Liste entspricht Wurzel
- Falls Liste $l$ auf Knoten $k$ verweist, verweist Liste $l.i$ auf $i$-tes Kind von $K$




1 verweist auf Anweisungsteil
1.0.1 verweist auf **while**-Schleife
1.0.1.0 verweist auf Bedingung der **while**-Schleife
2.0.5 verweist auf keinen AST-Knoten

## Diskussion

### Beobachtungen

- Signatur WHILE muss erweitert werden
- Aus Sicht der Navigation müssen alle Strukturbaumknoten einheitlich behandelt werden
- Es gibt Navigationslisten (engl *occurrences*), die auf keinen AST-Knoten verweisen.

### Lösungen

- Signaturerweiterungen mit entsprechender Erweiterung der Algebren
- Einführung von Untersorten
- Einführung von partiellen Operationen



## Partielle Signaturen mit Untersorten

**Definition 2.5 (Partielle Signaturen mit Untersorten)**

Tupel $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ wobei

i. $S$ endliche Menge von **Sortensymbolen**

ii. $\sqsubseteq \subseteq S \times S$ ist eine Halbordnung auf $S$ (**Untersortenbeziehung**)

iii. $F = \langle F_{w,s} \rangle_{w \in S^*, s \in S}$ ist eine Familie von Operationssymbolen (**totale Operationssymbole**) mit folgender Eigenschaft
Wenn $f : s_1 \times \cdots \times s_n \to s \in F$, $s_1' \sqsubseteq s_1, \ldots, s_n' \sqsubseteq s_n$ und $s \sqsubseteq s'$, dann ist $f : s_1' \times \cdots \times s_n' \to s' \in F$

iv. $F' = \langle F'_{w,s} \rangle_{w \in S^*, s \in S}$ ist eine von $F$ disjunkte Familie von Operationssymbolen, d.h. $F_{w,s} \cap F'_{w,s} = \emptyset$, (**partielle Operationssymbole**) mit folgender Eigenschaft:
Wenn $f : s_1 \times \cdots \times s_n \to ?s \in F'$ $s_1' \sqsubseteq s_1, \ldots, s_n' \sqsubseteq s_n$ unnd $s \sqsubseteq s'$, dann ist $f : s_1' \times \cdots \times s_n' \to ?s' \in F'$

**Notationen**

- $f : s_1 \times \cdots \times s_n \to s \in F$ statt $f \in F_{s_1 \cdots s_n, s}$
- $f : s_1 \times \cdots \times s_n \to ?s \in F'$ statt $f \in F'_{s_1 \cdots s_n, s}$

**Definition 2.6 ($\Sigma$-Terme)**

Definition analog den klassischen Signaturen
$\Rightarrow$ Sei $s \sqsubseteq s'$. Dann ist $\mathsf{T}_s(\Sigma, X) \subseteq \mathsf{T}_{s'}(\Sigma, X)$



## $\Sigma$-Algebren für partielle Signaturen mit Untersorten

**Definition 2.7 (Partielle ordnungssortierte Algebra)**

Sei $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ eine partielle Signatur mit Untersorten. Eine **partielle ordnungssortierte $\Sigma$-Algebra** ist ein Paar $\mathfrak{A} = \langle A, \Phi \rangle$ mit

i. $A = \langle A_s \rangle_{s \in S}$ ist eine Familie von **Trägermengen** ist, so dass $A_s \subseteq A_{s'}$ falls $s \sqsubseteq s'$ ist.

ii. Für alle Funktionssymbole $f : s_1^{(1)} \times \cdots \times s_n^{(1)} \to s^{(1)}, \ldots,$
$f : s_1^{(m)} \times \cdots \times s_n^{(m)} \to s^{(m)} \in F$ ist $f_A : \bigcup_{j=1}^{m} A_{s_1}^{(j)} \times \cdots \times \bigcup_{j=1}^{m} A_{s_n}^{(j)} \to \bigcup_{j=1}^{m} A_s^{(j)}$
eine totale Funktion (**Interpretation des Operationssymbols** $f$), die folgende Bedingung erfüllt: Falls $f : s_1 \times \cdots \times s_n \to s \in F$ und $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ dann ist auch $f_A(a_1, \ldots, a_n) \in A_s$.

iii. Für alle Funktionssymbole $f : s_1^{(1)} \times \cdots \times s_n^{(1)} \to ?s^{(1)}, \ldots,$
$f : s_1^{(m)} \times \cdots \times s_n^{(m)} \to ?s^{(m)} \in F'$ ist $f_A : \bigcup_{j=1}^{m} A_{s_1}^{(j)} \times \cdots \times \bigcup_{j=1}^{m} A_{s_n}^{(j)} \to \bigcup_{j=1}^{m} A_s^{(j)}$
eine partielle Funktion (**Interpretation des Operationssymbols** $f$), die folgende Bedingung erfüllt: Falls $f : s_1 \times \cdots \times s_n \to ?s \in F'$ und $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ dann ist auch $f_A(a_1, \ldots, a_n) \in A_s$.



## Erweiterungen von Signaturen

**Definition 2.8 (Signaturerweiterung)**

Sei $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ eine Signatur. Eine Signatur $\overline{\Sigma} = \langle \overline{S}, \overline{\sqsubseteq}, \overline{F}, \overline{F'} \rangle$ mit $S \subseteq \overline{S}$, $\sqsubseteq \subseteq \overline{\sqsubseteq}$, $F \subseteq \overline{F}$ und $F' \subseteq \overline{F'}$ heißt **Erweiterung der Signatur** $\Sigma$.

**Notationen**

- $\Sigma \subseteq \Sigma'$ notiert die Erweiterung $\Sigma'$ von $\Sigma$
- **sig** $\Sigma$ **extends** $\Sigma_1, \ldots, \Sigma_n$ **by sorts** $S$ **subsorts** $\sqsubseteq$ **operations** $F \uplus F'$ $\triangleq \Sigma = \langle S \cup S_1 \cup \cdots \cup S_n, \sqsubseteq \cup \sqsubseteq_1 \cup \cdots \cup \sqsubseteq_n, F \cup F_1 \cup \cdots \cup F_n, F' \cup F_1' \cup \cdots \cup F_n' \rangle$ wobei $\Sigma_i = \langle S_i, \sqsubseteq_i, F_i, F_i' \rangle$.

**Definition 2.9 (Restriktion einer $\Sigma$-Algebra)**

Sei $\mathfrak{A} = \langle A, \Phi \rangle$ eine $\Sigma_1$-Algebra zu einer partiellen Signatur $\Sigma_1$ und $\Sigma_0 \triangleq \langle (S_0, \sqsubseteq_0, F_0, F_0' \rangle \subseteq \Sigma_1$. Die **Restriktion von $\mathfrak{A}$ bzgl. $\Sigma$** ist definiert durch: $\mathfrak{A}|_{\Sigma_0} \triangleq \langle A', \Phi' \rangle$ wobei $A_s' \triangleq A_s$ für alle $s \in S_0$ und $f_{A'} \triangleq f_A$ für alle $f \in F_0 \cup F_0'$.

**Beobachtung**

$\mathfrak{A}|_{\Sigma_0}$ ist eine $\Sigma_0$-Algebra, die für die Sorten und für die Operationssymbole aus $\Sigma_0 \subseteq \Sigma_1$ dieselbe Interpretation hat wie die $\Sigma_1$-Algebra $\mathfrak{A}$.

## Beispiel 2.4: Navigation in abstrakten Syntaxbäumen

**Idee**

- Führe Signatur und Algebra für Listen ganzer Zahlen ein
- Erweitere Signatur für abstrakte Syntaxbäume um Navigationsleisten
- Definiere entsprechende Algebra

**Listen ganzer Zahlen**

**sig** OCC **extends** INT

| | | | | |
|---|---|---|---|---|
| **sorts** | OCC | | | |
| **operations** | $()$ : | | $\rightarrow$ OCC | leere Liste |
| | $(.)$ : | OCC $\times$ INT | $\rightarrow$ OCC | Infixnotation |

OCC-Algebra: $\mathfrak{O} \triangleq \langle O, \Phi \rangle$ wobei

- $\mathfrak{O}|_{\text{INT}} = \mathfrak{Z}$ und
- $\mathfrak{O}|_{\text{INT}'} = \mathfrak{T}(\text{INT}')$, $\text{INT}' \triangleq \langle \{\text{INT}\}, \{\ldots, -2, -1, 0, 1, 2, \ldots :\rightarrow \text{INT}\}\rangle$
- $O_{\text{OCC}}$ sind alle endlichen Listen ganzer Zahlen, d.h. $O_{\text{OCC}} \triangleq \{[x_1, \ldots, x_n] : n \geq 0, x_i \in \mathbb{Z}\}$
- $()_O \triangleq []$
  $(.)_O([x_1, \ldots, x_n], x) \triangleq [x_1, \ldots, x_n, x]$



## Beispiel 2.4: Navigation in abstrakten Syntaxbäumen (Forts.)

**Navigation in abstrakten Syntaxbäumen**

**sig** PROG **extends** WHILE, OCC

| | | | | |
|---|---|---|---|---|
| **sorts** | ASSIGN, READ, WRITE, WHILE, IF, PLUS, GEQ, MINUS, NODE | | | |
| **subsorts** | ASSIGN $\sqsubset$ STAT, READ $\sqsubset$ STAT, WRITE $\sqsubset$ STAT, WHILE $\sqsubset$ STAT, IF $\sqsubset$ STAT, DES $\sqsubset$ EXPR, PLUS $\sqsubset$ EXPR, GEQ $\sqsubset$ EXPR, MINUS $\sqsubset$ EXPR, ID $\sqsubset$ NODE, INTCONST $\sqsubset$ NODE, BOOLCONST $\sqsubset$ NODE, OP $\sqsubset$ NODE, PROG $\sqsubset$ NODE, STATS $\sqsubset$ NODE, DECLS $\sqsubset$ NODE, DECL $\sqsubset$ NODE, TYPE $\sqsubset$ NODE, STAT $\sqsubset$ NODE, EXPR $\sqsubset$ NODE | | | |
| **operations** | *mkassign* : | DES $\times$ EXPR | $\rightarrow$ ASSIGN | |
| | *mkif* : | EXPR $\times$ STATS $\times$ STATS | $\rightarrow$ IF | |
| | *mkwhile* : | EXPR $\times$ STATS | $\rightarrow$ WHILE | |
| | *mkread* : | DES | $\rightarrow$ READ | |
| | *mkwrite* : | EXPR | $\rightarrow$ WRITE | |
| | *mkgeq* : | EXPR $\times$ OP $\times$ EXPR | $\rightarrow$ GEQ | |
| | *mkplus* : | EXPR $\times$ OP $\times$ EXPR | $\rightarrow$ PLUS | |
| | *mkminus* : | EXPR $\times$ OP $\times$ EXPR | $\rightarrow$ MINUS | |
| | *mkdes* : | NAME | $\rightarrow$ EXPR | |
| | *occ* : | OCC $\times$ NODE | $\rightarrow$?NODE | |
| | *parent* : | OCC | $\rightarrow$?OCC | |

PROG-Algebra $\mathfrak{P}$ ist definiert durch: $\mathfrak{P}|_{\text{WHILE}} \triangleq \mathcal{T}(\text{WHILE})$, $\mathfrak{P}|_{\text{OCC}} \triangleq \mathfrak{O}$

*occ* ist in Beispiel 2.3 informell definiert

- $P_{\text{NODE}} \triangleq P_{\text{ID}} \cup \cdots \cup P_{\text{EXPR}}$
- Falls $occ_P([], t) = t$ für alle $t \in P_{\text{NODE}}$
- Falls $occ_P(o, t) = mkprog(d, s)$ ist, dann ist $occ_P(o{+}{+}[0], t) = d$, $occ(o{+}{+}[1], t) = s$ und $occ_P(o{+}{+}[i], t) = \bot_{\text{NODE}}$ für $i \geq 2$
- Falls $occ_P(o, t) = mkif(e, s_1, s_2)$ ist, dann ist $occ_P(o{+}{+}[0], t) = e$, $occ(o{+}{+}[1], t) = s_1$, $occ(o{+}{+}[2], t) = s_2$ und $occ_P(o{+}{+}[i], t) = \bot_{\text{NODE}}$ für $i \geq 3$ usw.
- $parent_P(o{+}{+}[i]) = o$



## Diskussion

**Problem**

Semantik wird an der konkreten Definition der Algebra festgelegt

- Eigenschaften nicht transparent auf Spezifikatiationsebene
- Umständliche Definition

**Idee**

Spezifikation der Eigenschaften durch Gleichungen, Formeln etc.

- Nachweis von Eigenschaften über Beweiskalküle
- Hier genügen zunächst bedingte und unbedingte Gleichungen



## Gleichungen und bedingte Gleichungen

**Definition 2.10 ((Bedingte) $\Sigma$-Gleichung, $\Sigma$-Definitionsbereichsprädikat, $\Sigma$-Sortenprüfung )**

Sei $\Sigma \triangleq (S, \sqsubseteq, F, F')$ eine Signatur. Eine **$\Sigma$-Gleichung** ist ein Paar von $\Sigma$-Termen $t_1 \doteq t_2$, $t_1, t_2 \in \mathsf{T}_s(\Sigma, X)$ für eine Sorte $s$ und Variablen $X$. Für $s \in S$, $t \in \mathsf{T}_s(\Sigma, X)$ heißen $D_s(t)$ und $\neg D_s(t)$ **$\Sigma$-Definitionsbereichsprädikat** sowie $t$ **is** $s$ und $\neg(t$ **is** $s)$ **$\Sigma$-Sortenprüfung**. Eine **bedingte $\Sigma$-Gleichung** ist ein $n+1$-Tupel $eq_1 \wedge \cdots \wedge eq_n \Rightarrow eq$, $n \geq 0$, wobei $eq_1, \ldots, eq_n, eq \in \mathsf{E}(\Sigma, X) \cup \mathsf{D}(\Sigma, X) \cup \mathsf{S}(\Sigma, X)$.

**Notationen:**

- $\mathsf{E}(\Sigma, X)$ notiert die Menge der $\Sigma$-Gleichungen mit Variablen $X$.
- $\mathsf{D}(\Sigma, X)$ notiert die Menge der $\Sigma$-Definitionsbereichsprädikate mit Variablen $X$.
- $\mathsf{S}(\Sigma, X)$ notiert die Menge der $\Sigma$-Sortenprüfungen mit Variablen $X$.
- $\mathsf{C}(\Sigma, X)$ notiert die Menge der bedingten $\Sigma$-Gleichungen mit Variablen $X$

## Erfüllung von Gleichungen und bedingten Gleichungen

**Definition 2.11 (Erfüllung)**

Sei $\mathfrak{A} = \langle A, \Phi \rangle$ eine $\Sigma$-Algebra und $X$ eine Menge von Variablen.

**i.** $\mathfrak{A}$ **erfüllt** $\Sigma$**-Gleichung** $t_1 \doteq t_2$ gdw. $[\![t_1]\!]_A^\beta = [\![t_2]\!]_a^\beta$ für alle Belegungen $\beta : X \to A$ (Notation: $\mathfrak{A} \models t_1 \doteq t_2$)

**ii.** $\mathfrak{A}$ **erfüllt Definitionsprädikat** $D_s(t)$ gdw. $[\![t]\!]_A^\beta \neq \bot$ für alle Belegungen $\beta$ mit $\beta_s(x) \neq \bot_s$ (Notation: $\mathfrak{A} \models D_s(t)$) und $\mathfrak{A} \models \neg D_s(t)$ gdw. $\mathfrak{A} \not\models D_s(t)$

**iii.** $\mathfrak{A}$ **erfüllt Typprüfung** $t$ **is** $s$ gdw. $[\![t]\!]_A^\beta \in A_s$ für alle Belegungen $\beta$. (Notation: $\mathfrak{A} \models t$ **is** $s$) und $\mathfrak{A} \models \neg(t$ **is** $s)$ gdw. $\mathfrak{A} \not\models t$ **is** $s$

**iv.** $\mathfrak{A}$ **erfüllt bedingte** $\Sigma$**-Gleichung** $eq_1 \wedge \cdots \wedge eq_n \Rightarrow eq$ gdw. $\mathfrak{A} \models eq_1, \ldots, \mathfrak{A} \models eq_n$ impliziert $\mathfrak{A} \models eq$. (Notation: $\mathfrak{A} \models eq_1 \wedge \cdots \wedge eq_n \Rightarrow eq$)

**Beobachtungen**

- Kann erweitert werden auf Mengen von Algebren und Mengen von Gleichungen
- Umkehrung: Menge bedingter Gleichungen definiert eine Menge von Algebren, die diese Gleichungen erfüllen
- Falls $\Sigma \subseteq \Sigma'$, $c \in \mathrm{C}(\Sigma, X)$ und $\mathfrak{A} \models c$ für eine $\Sigma'$-Algebra $\mathfrak{A}$, dann gilt auch $\mathfrak{A}|_\Sigma \models c$

$\Rightarrow$ Verwende bedingte Gleichungen zur Spezifikation gewünschter Eigenschaften



## Abstrakte Datentypen

**Definition 2.12 (Abstrakter Datentyp und Interpretation)**

Ein **abstrakter Datentyp** ist ein Tripel $\mathcal{A} \triangleq \langle \Sigma, X, \mathrm{C} \rangle$ wobei $\Sigma$ Signatur, $X$ Menge von Variablen und C Menge bedingter $\Sigma$-Gleichungen mit Variablen $X$ (**Axiome**). Die **Interpretation** von $\mathcal{A}$ ist die Menge $I_\mathcal{A} \triangleq \{\mathfrak{A} \in \mathrm{Alg}(\Sigma) : \mathfrak{A} \models \mathrm{C}\}$

**Beispiel 2.5: Navigationslisten**

**spec** PROG **extends** WHILE, OCC
**sorts** ASSIGN, READ, WRITE, WHILE, IF, PLUS, GEQ, MINUS, NODE
**subsorts** ASSIGN $\sqsubset$ STAT, READ $\sqsubset$ STAT, WRITE $\sqsubset$ STAT, WHILE $\sqsubset$ STAT, IF $\sqsubset$ STAT, DES $\sqsubset$ EXPR, PLUS $\sqsubset$ EXPR, GEQ $\sqsubset$ EXPR, MINUS $\sqsubset$ EXPR, ID $\sqsubset$ NODE, INTCONST $\sqsubset$ NODE, BOOLCONST $\sqsubset$ NODE, OP $\sqsubset$ NODE, PROG $\sqsubset$ NODE, STATS $\sqsubset$ NODE, DECLS $\sqsubset$ NODE, DECL $\sqsubset$ NODE, TYPE $\sqsubset$ NODE, STAT $\sqsubset$ NODE, EXPR $\sqsubset$ NODE

**operations**

| | | |
|---|---|---|
| $mkassign$ : | DES $\times$ EXPR | $\to$ ASSIGN |
| $mkif$ : | EXPR $\times$ STATS $\times$ STATS | $\to$ IF |
| $mkwhile$ : | EXPR $\times$ STATS | $\to$ WHILE |
| $mkread$ : | DES | $\to$ READ |
| $mkwrite$ : | EXPR | $\to$ WRITE |
| $mkgeq$ : | EXPR $\times$ OP $\times$ EXPR | $\to$ GEQ |
| $mkplus$ : | EXPR $\times$ OP $\times$ EXPR | $\to$ PLUS |
| $mkminus$ : | EXPR $\times$ OP $\times$ EXPR | $\to$ MINUS |
| $mkdes$ : | NAME | $\to$ EXPR |
| $occ$ : | OCC $\times$ NODE | $\to$?NODE |
| $parent$ : | OCC | $\to$?OCC |

**vars** $x$ : NODE; $o$ : OCC; $d$ : DECLS; $s, s_1, s_2$ : STAT, $i$ : INT

**axioms**

$$
\begin{array}{lcl}
 & & occ((), x) \doteq x \\
occ(o, x) \doteq mkprog(d, s) & \Rightarrow & occ(o.0, x) \doteq d \\
occ(o, x) \doteq mkprog(d, s) & \Rightarrow & occ(o.1, x) \doteq s \\
occ(o, x) \doteq mkprog(d, s) \wedge D_{\mathrm{OCC}}(occ(0.i), x) & \Rightarrow & i \geqslant 0 \doteq true \\
occ(o, x) \doteq mkprog(d, s) \wedge D_{\mathrm{OCC}}(occ(0.i), x) & \Rightarrow & i \leqslant 1 \doteq true \\
 & & \ldots \\
 & & \neg D_{\mathrm{OCC}}(parent(())) \\
 & & parent(o.i) \doteq o
\end{array}
$$



## Fazit

### Zusammenfassung

- Abstrakte Datentypen sind geeignet statische Strukturen einer Programmiersprache zu modellieren
- Die abstrakte Syntax kann durch eine Termalgebra dargestellt werden
- Eigenschaften können durch Algebren oder bedingte Gleichungen dargestellt werden
- Um in abstrakten Syntaxbäumen zu navigieren benötigt man partielle Operationssymbole und Untersorten



## 2.3 Abstrakte Zustandsmaschinen

### Vorgehen zur Definition einer Programmiersprachsemantik

- Definition der abstrakten Syntax samt Navigation
- Definition des nächsten auszuführenden Befehls (Bestandteil der statischen Semantik)
- Definition des Zustandsraums
- Definition des Anfangszustands
- Definition der Zustandsübergänge

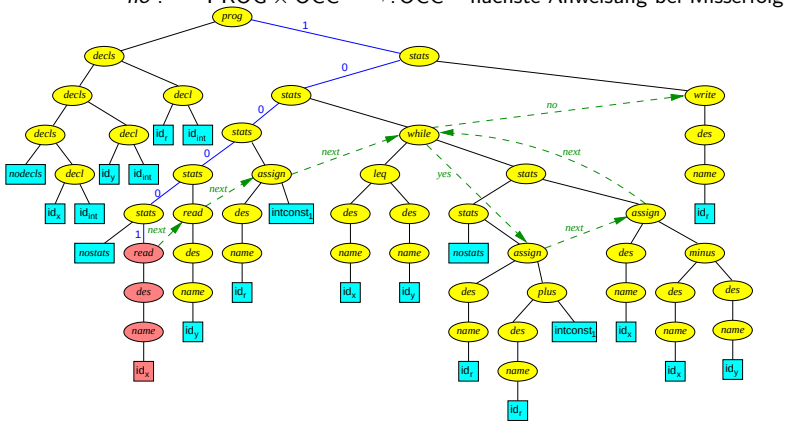
### Beobachtungen

- Abstrakte Syntax und Navigation wurde bereits durch die Spezifikation PROG definiert.
- Zustandsraum und Zustandsübergangssystem muss noch definiert werden

## Beispiel 2.6: Nächste auszuführende Befehle

**spec** PROGSTAT **extends** PROG
**operations**

| | | | |
|---|---|---|---|
| *first* : | PROG | $\rightarrow$ OCC | erste Anweisung |
| *next* : | PROG $\times$ OCC | $\rightarrow$?OCC | nächste Anweisung |
| *yes* : | PROG $\times$ OCC | $\rightarrow$?OCC | nächste Anweisung bei Erfolg |
| *no* : | PROG $\times$ OCC | $\rightarrow$?OCC | nächste Anweisung bei Misserfolg |




**Übung**

Spezifizieren Sie die Eigenschaften von *next*, *yes* und *no* als abstrakter Datentyp und als attributierte Grammatik. Welcher Zusammenhang besteht zwischen den Attributierungsregeln und den bedingten Gleichungen?



## Beispiel 2.7: Zustandsraum der While-Sprache

**Komponenten des Zustandsraums**

- Programm, das ausgeführt wird
- Speicher, d.h. die Menge der Variablen, die Wert enthalten können (Werte sind ganze Zahlen oder Wahrheitswerte)
- Befehlzeiger, d.h. die Stelle im Programm, die als nächstes ausgeführt wird
- Ein- und Ausgabeströme (Listen ganzer Zahlen)

$\Rightarrow$ Das definiert eine Signatur

**Spezifikation des Zustandsraums**

**spec** STATE **extends** PROG
**sorts** VALUE
**subsorts** INT $\sqsubseteq$ VALUE, BOOL $\sqsubseteq$ VALUE
**operations**

| | | | |
|---|---|---|---|
| *prog* : | | $\rightarrow$ PROG | |
| *mem* : | ID | $\rightarrow$?VALUE | |
| *pc* : | | $\rightarrow$ OCC | |
| *inp*, *out* : | | $\rightarrow$ OCC | |
| *eval* : | EXPR | $\rightarrow$?VALUE | |

**vars** $x$ : ID, $n$ : INT, $b$ : BOOL, $e_1$ : EXPR, $e_2$ : EXPR
**axioms**

$$
\begin{aligned}
eval(mkdes(mkname(x))) &\doteq mem(x)\\
eval(mkintconst(n)) &\doteq n\\
eval(mkboolconst(b)) &\doteq b\\
eval(mkgeq(e_1,e_2)) &\doteq intgeq(eval(e_1),eval(e_2))\\
eval(mkplus(e_1,e_2)) &\doteq intplus(eval(e_1),eval(e_2))\\
eval(mkminus(e_1,e_2)) &\doteq intminus(eval(e_1),eval(e_2))
\end{aligned}
$$



## Beispiel 2.8: Anfangszustände der While-Sprache

**Anfangszustand**

- Der Speicher ist leer
- Der Ausgabestrom ist leer
- Ausführung beginnt mit Ausführung der Anweisungsfolge

**Spezifikation des Anfangszustands**

**spec** INITIALSTATE **extends** STATE
**vars** $x$ : ID
**axioms**

$$
\begin{aligned}
&\neg D_{\mathrm{OCC}}(mem(x))\\
out &\doteq ()\\
pc &\doteq first(prog)
\end{aligned}
$$



## Diskussion

**Beobachtung**

- Der Zustandsraum ist eine mehrsortige Signatur $\Sigma$

$\Rightarrow$ Die Menge der Zustände ist eine Menge von $\Sigma$-Algebren

**Ziel**

Definition der Zustandsübergänge durch Änderung der Interpretation

**Definition 2.13 ($\Sigma$-Aktualisierung, Konsistenz, Nachfolgezustand)**

Sei $\Sigma \triangleq \langle S, \sqsubseteq, F, F'\rangle$ eine Signatur und $X \triangleq (X_s)_{s\in S}$ eine Menge von Variablen.

**i.** Eine **$\Sigma$-Aktualisierung** ist ein Paar $t_1 := t_2$ von $\Sigma$-Termen $t_1, t_2 \in \mathcal{T}_s(\Sigma, X)$ derselben Sorte $s \in S$.

**ii.** Eine Menge $\mathbb{U}$ von $\Sigma$-Aktualisierungen heißt **konsistent bzgl. einer $\Sigma$-Algebra** $\mathfrak{A}$ gdw. für alle $f(t_1,\ldots,t_n) := t_0, f(u_1,\ldots,u_n) := u_0 \in \mathbb{U}$ gilt: $[\![t_1]\!]_{\mathfrak{A}} = [\![u_1]\!]_{\mathfrak{A}}, \ldots [\![t_n]\!]_{\mathfrak{A}} = [\![u_n]\!]_{\mathfrak{A}}$ impliziert $[\![t_0]\!]_{\mathfrak{A}} = [\![u_0]\!]_{\mathfrak{A}}$..

**iii.** Sei $\mathfrak{A}$ eine $\Sigma$-Algebra und $\mathbb{U} \neq \emptyset$ eine konsistente Menge von $\Sigma$-Aktualisierungen. Der **Nachfolgezustand von $\mathfrak{A}$ bzgl.** $\mathbb{U}$ ist die $\Sigma$-Algebra $next_{\mathbb{U}}(\mathfrak{A}) \triangleq (N, [\![\cdot]\!]_{next(\mathfrak{A})})$ wobei $N = (A_s)_{s\in S}$ die Trägermengen von $\mathfrak{A}$ und $[\![\cdot]\!]_{next(\mathfrak{A})}$ durch

$$
[\![f]\!]_{next(\mathfrak{A})}(a_1,\ldots,a_n) \triangleq \begin{cases} a & \text{falls } f(t_1,\ldots,t_n) := t \in \mathbb{U} \text{ und} \\ & [\![t_1]\!]_{\mathfrak{A}} = a_1, \ldots, [\![t_n]\!]_{\mathfrak{A}} = a_n, [\![t]\!]_{\mathfrak{A}} = a \\ [\![f]\!]_A(a_1,\ldots,a_n) & \text{sonst} \end{cases}
$$

definiert ist.

# Beispiel 2.9: Aktualisierungen

### Spezifikation

**spec** INIT **extends** INITIALSTATE
**axioms** $prog \doteq$ Term aus Beispiel 1.1
$inp \doteq ().12.5$

### Init-Algebra

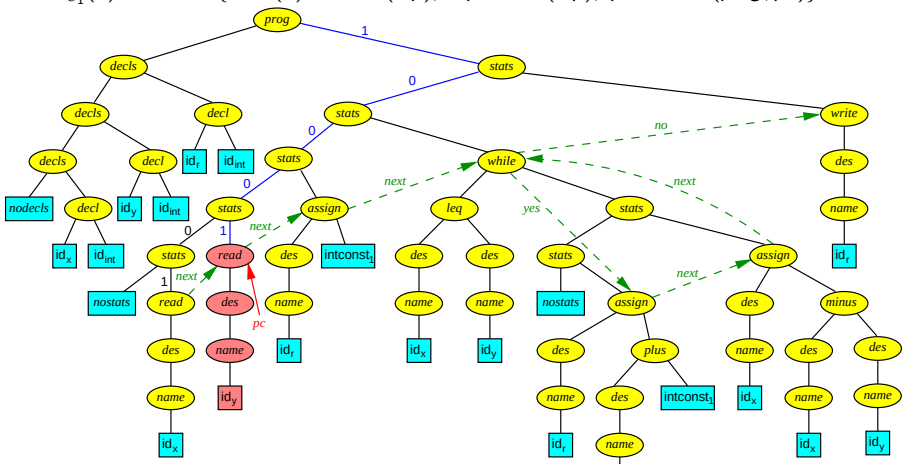
$\mathfrak{I} \triangleq \langle I, [\![\cdot]\!]_I \rangle$ mit $I_{\text{VALUE}} \triangleq \mathbb{Z} \cup \mathbb{B}$

$$
\begin{aligned}
[\![mem]\!]_I(x) &\triangleq \bot_{\text{VALUE}} \\
[\![inp]\!]_I &\triangleq [5, 12] \\
[\![out]\!]_I &\triangleq [] \\
[\![pc]\!]_I &\triangleq [\![().1.0.0.0.0.1]\!]_I
\end{aligned}
$$



# Beispiel 2.9: Erster Zustandsübergang

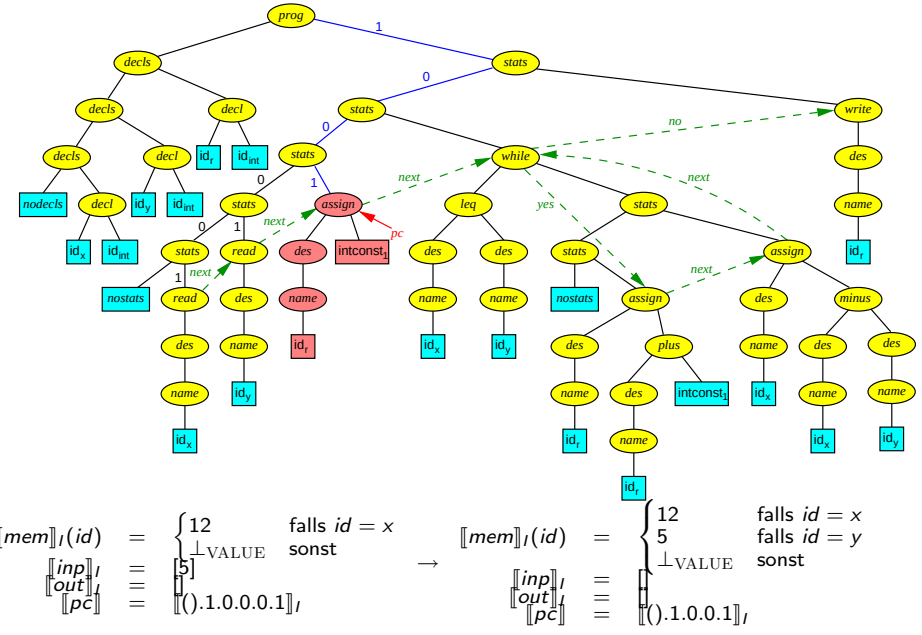
$\mathfrak{S}_1 \triangleq next_{\mathbb{U}_1}(\mathfrak{I})$ mit $\mathbb{U}_1 \triangleq \{mem(x) := head(inp),\ inp := tail(inp),\ pc := next(prog, pc)\}$




$$
\begin{aligned}
[\![mem]\!]_I(id) &\triangleq \bot_{\text{VALUE}} \\
[\![inp]\!]_I &\triangleq [12, 5] \\
[\![out]\!]_I &\triangleq [] \\
[\![pc]\!] &\triangleq [\![().1.0.0.0.0.1]\!]_I
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
[\![mem]\!]_I(id) &= \begin{cases} 12 & \text{falls } id = x \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\
[\![inp]\!]_I &= [5] \\
[\![out]\!]_I &= [] \\
[\![pc]\!] &= [\![().1.0.0.0.0.1]\!]_I
\end{aligned}
$$



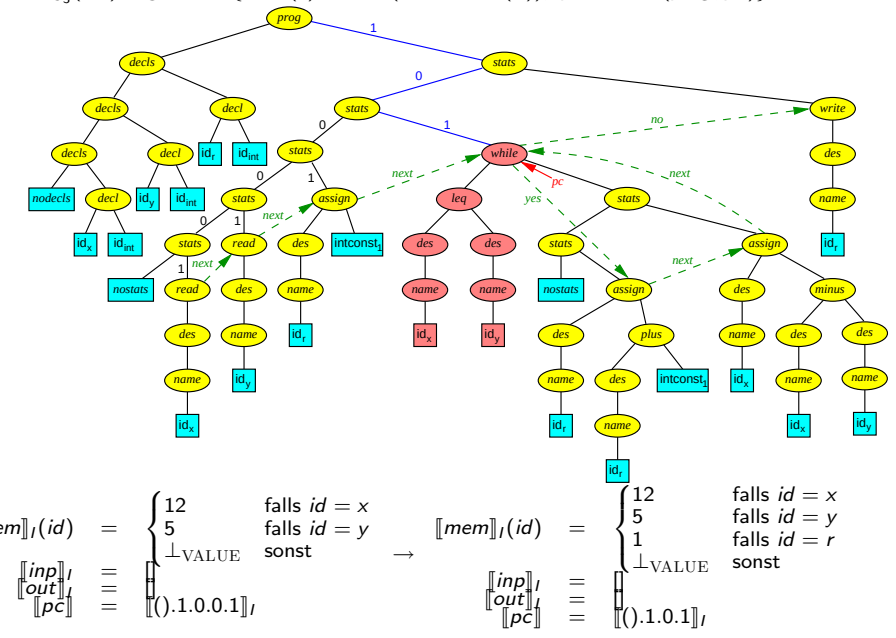
$\mathfrak{S}_2 \triangleq next_{\mathbb{U}_2}(\mathfrak{S}_1)$ bzgl. $\mathbb{U}_2 \triangleq \{mem(y) := head(inp),\ inp := tail(inp),\ pc := next(pc, prog)\}$




$$
\begin{aligned}
[\![mem]\!]_I(id) &= \begin{cases} 12 & \text{falls } id = x \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\
[\![inp]\!]_I &= [5] \\
[\![out]\!]_I &= [] \\
[\![pc]\!] &= [\![().1.0.0.0.0.1]\!]_I
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
[\![mem]\!]_I(id) &= \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\
[\![inp]\!]_I &= [] \\
[\![out]\!]_I &= [] \\
[\![pc]\!] &= [\![().1.0.0.0.1]\!]_I
\end{aligned}
$$

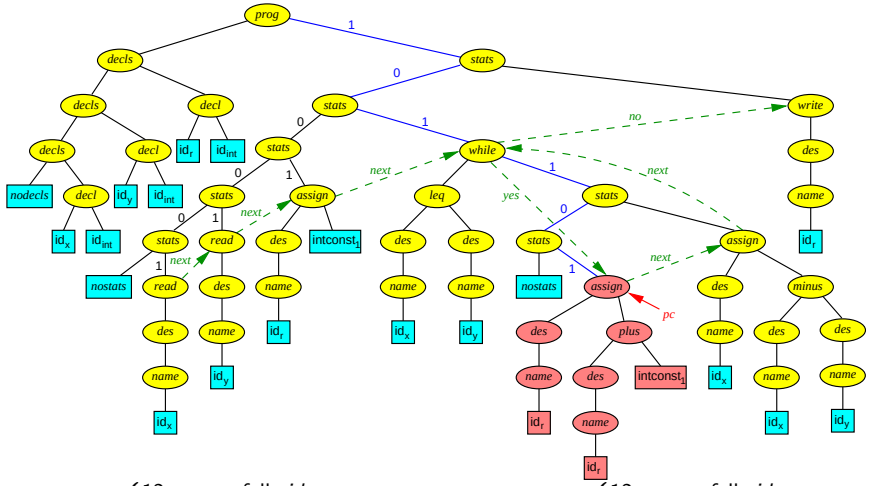


$\mathfrak{S}_3 \triangleq next_{\mathbb{U}_3}(\mathfrak{S}_2)$ bzgl. $\mathbb{U}_3 \triangleq \{mem(r) := eval(mkintconst(1)),\ pc := next(prog, pc)\}$




$$
\begin{aligned}
[\![mem]\!]_I(id) &= \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\
[\![inp]\!]_I &= [] \\
[\![out]\!]_I &= [] \\
[\![pc]\!] &= [\![().1.0.0.0.1]\!]_I
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
[\![mem]\!]_I(id) &= \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\
[\![inp]\!]_I &= [] \\
[\![out]\!]_I &= [] \\
[\![pc]\!] &= [\![().1.0.1]\!]_I
\end{aligned}
$$

$\mathfrak{S}_4 \triangleq next_{\mathbb{U}_4}(\mathfrak{S}_3)$ bzgl. $\mathbb{U}_4 \triangleq \{pc := yes(prog, pc)\}$

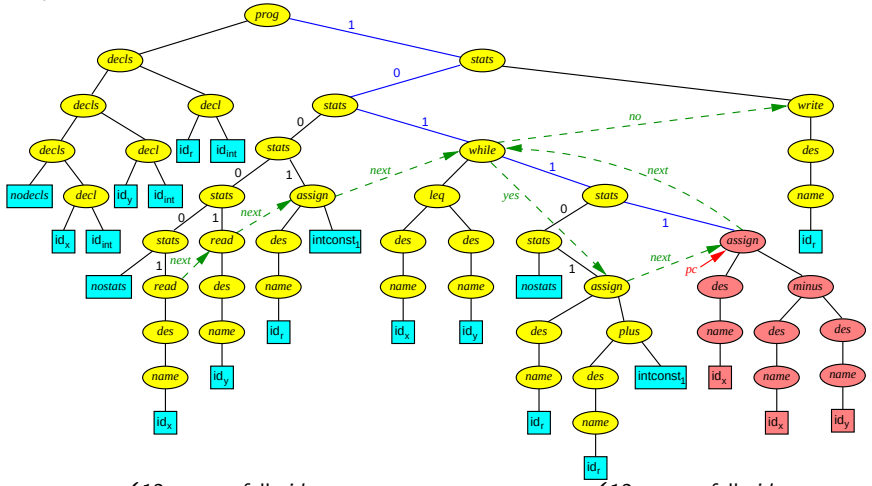



$$[\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \rightarrow [\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases}$$

$$[\![inp]\!]_I = [] \qquad [\![inp]\!]_I = []$$
$$[\![out]\!]_I = [] \qquad [\![out]\!]_I = []$$
$$[\![pc]\!] = [\![().1.0.1]\!]_I \qquad [\![pc]\!] = [\![().1.0.1.1.0.1]\!]_I$$



$\mathfrak{S}_5 \triangleq next_{\mathbb{U}_5}(\mathfrak{S}_4)$, $\mathbb{U}_5 \triangleq \{mem(r) := eval(mkplus(des(r), mkintconst(1))), pc := next(prog, pc)\}$




$$[\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \rightarrow [\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 2 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases}$$

$$[\![inp]\!]_I = [] \qquad [\![inp]\!]_I = []$$
$$[\![out]\!]_I = [] \qquad [\![out]\!]_I = []$$
$$[\![pc]\!] = [\![().1.0.1.1.0.1]\!]_I \qquad [\![pc]\!] = [\![().1.0.1.1.1.1]\!]_I$$



## Übung

Bestimmen Sie die folgenden STATE-Algebren:

- $\mathfrak{S}_6 \triangleq next_{\mathbb{U}_6}(\mathfrak{S}_5)$ bzgl.
  $\mathbb{U}_6 \triangleq \{\ mem(x) := eval(mkminus(des(x), des(y))), pc := next(prog, pc)\}$
- $\mathfrak{S}_7 \triangleq next_{\mathbb{U}_7}(\mathfrak{S}_6)$ bzgl. $\mathbb{U}_7 \triangleq \{\ pc := yes(prog, pc)\}$
- $\mathfrak{S}_8 \triangleq next_{\mathbb{U}_8}(\mathfrak{S}_7)$ bzgl. $\mathbb{U}_8 \triangleq$
  $\{\ mem(r) := eval(binexpr(des(r), plus, mkintconst(1))), pc := next(prog, pc)\}$
- $\mathfrak{S}_9 \triangleq next_{\mathbb{U}_9}(\mathfrak{S}_8)$ bzgl.
  $\mathbb{U}_9 \triangleq \{\ mem(x) := eval(mkminus(des(x), des(y))), pc := next(prog, pc)\}$
- $\mathfrak{S}_{10} \triangleq next_{\mathbb{U}_{10}}(\mathfrak{S}_9)$ bzgl. $\mathbb{U}_9 \triangleq \{\ pc := yes(prog, pc)\}$
- $\mathfrak{S}_{11} \triangleq next_{\mathbb{U}_{11}}(\mathfrak{S}_{10})$ bzgl.
  $\mathbb{U}_{11} \triangleq \{\ out := out.eval(des(r)), pc := next(prog, pc)\}$



## Diskussion

**Beobachtungen**

- Die Aktualisierungsmengen $\mathbb{U}_1$–$\mathbb{U}_5$ führen die Aktualisierungen der ersten 5 Anweisungen durch
- Die Trägermengen bleiben erhalten
- Die Interpretation vieler Operationssymbole bleiben erhalten

**Weiteres Vorgehen**

- Spezifikation der Zustandsübergangsregeln
- Semantik eines Programms ist Zustandsübergangsfolge

## Abstrakte Zustandsmaschinen

**Definition 2.14 (Abstrakte Zustandsmaschine (engl. Abstract State Machine, kurz ASM))**

Sei $\Sigma$ Signatur. Eine $\Sigma$-ASM ist ein Tupel $\mathcal{A} \triangleq \langle \Sigma, \mathbb{I}, \mathbb{A}, R \rangle$ wobei

**i.** $\mathbb{A}$ eine Menge von $\Sigma$-Algebren ist (**Zustände**)

**ii.** $\mathbb{I} \subseteq \mathbb{A}$ (**Anfangszustände**)

**iii.** und $R$ endliche Menge von Regeln der Form **if** $\varphi$ **then** $\mathbb{U}$, $\varphi$ eine prädikatenlogische Formel über $\Sigma$ und $\mathbb{U}$ eine endliche Menge von $\Sigma$-Aktualisierungen ist

Die Menge $Update_{\mathcal{A}}(\mathfrak{A}) \triangleq \{u \in \mathbb{U} : \exists \textbf{if } \varphi \textbf{ then } \mathbb{U} \in R \bullet \mathfrak{A} \models \varphi\}$ heißt **Aktualisierungsmenge im Zustand** $\mathfrak{A}$.
**Notation:** ASM($\Sigma$) ist die Menge aller $\Sigma$-ASMs



## Beispiel 2.10: ASM $\mathcal{W}$ für einfache While-Sprache

**if** $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog))$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog)) \doteq true$ **then**
  $pc := yes(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

$$
\begin{array}{llll}
lhs: & OCC \times PROG & \to ?NODE & D(lhs(o,p)) \Leftrightarrow occ(o,p) \textbf{ is ASSIGN} \\
rhs: & OCC \times PROG & \to ?NODE & lhs(o,p) \doteq occ(o.0,p) \\
dest: & OCC \times PROG & \to ?NODE & D(rhs(o,p)) \Leftrightarrow occ(o,p) \textbf{ is ASSIGN} \\
arg: & OCC \times PROG & \to ?NODE & rhs(o,p) \doteq occ(o.1,p) \\
cond: & OCC \times PROG & \to ?NODE & D(dest(o,p)) \Leftrightarrow occ(o,p) \textbf{ is READ} \\
 & & & dest(o,p) \doteq occ(o.0,p)
\end{array}
$$

$$
\begin{array}{rcl}
D(arg(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is WRITE} \\
arg(o,p) & \doteq & occ(o.0,p) \\
D(cond(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is WHILE} \vee occ(o,p) \textbf{ is IF} \\
cond(o,p) & \doteq & occ(o.0,p)
\end{array}
$$



## Beispiel 2.11: Aktualisierungsmengen

**if** $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq true$ **then**
  $pc := yes(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

$$
\begin{array}{llll}
lhs: & OCC \times PROG & \to ?NODE & \\
rhs: & OCC \times PROG & \to ?NODE & \\
dest: & OCC \times PROG & \to ?NODE & \\
arg: & OCC \times PROG & \to ?NODE & \\
cond: & OCC \times PROG & \to ?NODE &
\end{array}
$$

$$
\begin{array}{rcl}
D(lhs(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is ASSIGN} \\
lhs(o,p) & \doteq & occ(o.0,p) \\
D(rhs(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is ASSIGN} \\
rhs(o,p) & \doteq & occ(o.1,p) \\
D(dest(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is READ} \\
dest(o,p) & \doteq & occ(o.0,p) \\
D(arg(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is WRITE} \\
arg(o,p) & \doteq & occ(o.0,p) \\
D(cond(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is WHILE} \vee occ(o,p) \textbf{ is IF} \\
cond(o,p) & \doteq & occ(o.0,p)
\end{array}
$$

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

**Zustand** $\mathfrak{I}$:

$$
\begin{array}{rcl}
[\![mem]\!]_I(x) & \triangleq & \bot_{\text{VALUE}} \\
[\![inp]\!]_I & \triangleq & [5, 12] \\
[\![out]\!]_I & \triangleq & [] \\
[\![pc]\!]_I & \triangleq & [\![().1.0.0.0.0.1]\!]_I
\end{array}
$$

- Es gilt nur $\mathfrak{I} \models occ(prog, pc) \in$ READ.

$\Rightarrow$ $Update_{\mathcal{W}}(\mathfrak{I}) = \{$ $mem(x) := head(inp)$,
$inp := tail(inp)$,
$pc := next(prog, pc)\}$



## Beispiel 2.11: Aktualisierungsmengen

**if** $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq true$ **then**
  $pc := yes(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

$$
\begin{array}{llll}
lhs: & OCC \times PROG & \to ?NODE & \\
rhs: & OCC \times PROG & \to ?NODE & \\
dest: & OCC \times PROG & \to ?NODE & \\
arg: & OCC \times PROG & \to ?NODE & \\
cond: & OCC \times PROG & \to ?NODE &
\end{array}
$$

$$
\begin{array}{rcl}
D(lhs(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is ASSIGN} \\
lhs(o,p) & \doteq & occ(o.0,p) \\
D(rhs(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is ASSIGN} \\
rhs(o,p) & \doteq & occ(o.1,p) \\
D(dest(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is READ} \\
dest(o,p) & \doteq & occ(o.0,p) \\
D(arg(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is WRITE} \\
arg(o,p) & \doteq & occ(o.0,p) \\
D(cond(o,p)) & \Leftrightarrow & occ(o,p) \textbf{ is WHILE} \vee occ(o,p) \textbf{ is IF} \\
cond(o,p) & \doteq & occ(o.0,p)
\end{array}
$$

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

**Zustand** $\mathfrak{S}_1 = next_{Update_{\mathcal{W}}(\mathfrak{I})}(\mathfrak{I})$:

$$
\begin{array}{rcl}
[\![mem]\!]_I(id) & = & \begin{cases} 12 & \text{falls } id = x \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\
[\![inp]\!]_I & = & [5] \\
[\![out]\!]_I & = & [] \\
[\![pc]\!] & = & [\![().1.0.0.0.1]\!]_I
\end{array}
$$

- Es gilt nur $\mathfrak{S}_1 \models occ(prog, pc) \in$ READ.

$\Rightarrow$ $Update_{\mathcal{W}}(\mathfrak{S}_1) = \{$ $mem(y) := head(inp)$,
$inp := tail(inp)$,
$pc := next(prog, pc)\}$

## Beispiel 2.11: Aktualisierungsmengen

**if** $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog))$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq true$ **then**
  $pc := yes(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

$lhs : OCC \times PROG \to ?NODE$
$rhs : OCC \times PROG \to ?NODE$
$dest : OCC \times PROG \to ?NODE$
$arg : OCC \times PROG \to ?NODE$
$cond : OCC \times PROG \to ?NODE$

$D(lhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$lhs(o,p) \doteq occ(o.0,p)$
$D(rhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$rhs(o,p) \doteq occ(o.1,p)$
$D(dest(o,p)) \Leftrightarrow occ(o,p)$ **is** READ
$dest(o,p) \doteq occ(o.0,p)$
$D(arg(o,p)) \Leftrightarrow occ(o,p)$ **is** WRITE
$arg(o,p) \doteq occ(o.0,p)$
$D(cond(o,p)) \Leftrightarrow occ(o,p)$ **is** WHILE $\vee$ $occ(o,p)$ **is** IF
$cond(o,p) \doteq occ(o.0,p)$

**Zustand** $\mathfrak{S}_2 = next_{Update_{\mathcal{W}}(\mathfrak{S}_1)}(\mathfrak{S}_1)$:

$$[\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ \bot_{\mathrm{VALUE}} & \text{sonst} \end{cases}$$

$[\![inp]\!]_I = []$
$[\![out]\!]_I = []$
$[\![pc]\!] = [\![().1.0.0.1]\!]_I$

- Es gilt nur $\mathfrak{S}_2 \models occ(prog, pc) \in$ ASSIGN.

$\Rightarrow$ $Update_{\mathcal{W}}(\mathfrak{S}_2) = \{ \; mem(r) := 1,$
$pc := next(prog, pc)\}$



## Beispiel 2.11: Aktualisierungsmengen

**if** $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog)) \doteq true$ **then**
  $pc := yes(prog, pc)$
**if** $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog)) \doteq false$ **then**
  $pc := no(prog, pc)$

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

$lhs : OCC \times PROG \to ?NODE$
$rhs : OCC \times PROG \to ?NODE$
$dest : OCC \times PROG \to ?NODE$
$arg : OCC \times PROG \to ?NODE$
$cond : OCC \times PROG \to ?NODE$

$D(lhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$lhs(o,p) \doteq occ(o.0,p)$
$D(rhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$rhs(o,p) \doteq occ(o.1,p)$
$D(dest(o,p)) \Leftrightarrow occ(o,p)$ **is** READ
$dest(o,p) \doteq occ(o.0,p)$
$D(arg(o,p)) \Leftrightarrow occ(o,p)$ **is** WRITE
$arg(o,p) \doteq occ(o.0,p)$
$D(cond(o,p)) \Leftrightarrow occ(o,p)$ **is** WHILE $\vee$ $occ(o,p)$ **is** IF
$cond(o,p) \doteq occ(o.0,p)$

**Zustand** $\mathfrak{S}_3 = next_{Update_{\mathcal{W}}(\mathfrak{S}_2)}(\mathfrak{S}_2)$:

$$[\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\mathrm{VALUE}} & \text{sonst} \end{cases}$$

$[\![inp]\!]_I = []$
$[\![out]\!]_I = []$
$[\![pc]\!] = [\![().1.0.1]\!]_I$

- Es gilt $\mathfrak{S}_3 \models occ(prog, pc) \in$ WHILE und $\mathfrak{S}_3 \models eval(\mathtt{x>=y}) \doteq true$

$\Rightarrow$ $Update_{\mathcal{W}}(\mathfrak{S}_3) = \{ \; pc := yes(prog, pc)\}$

**Übung** Bestimmen Sie die Aktualisierungsmengen der Zustände $\mathfrak{S}_{i+1} = next_{Update_{\mathcal{W}}(\mathfrak{S}_i)}(\mathfrak{S}_i)$, $i = 4, \ldots$



## Makros und Notationen zur besseren Lesbarkeit

**Beobachtungen**

- Einige Aktualisierungen und Terme werden häufig benutzt
- Einige Terme sind unübersichtlich

$\Rightarrow$ Einführung von Abkürzungen

- Bei der Semantik einer while-Schleife würde eine "lokale" Regel einsichtiger sein

**Beispiel 2.12: Abkürzungen**

**if** $CT$ **is** ASSIGN **then**
  $mem(Dest) := eval(Src)$
  $Proceed$
**if** $CT$ **is** READ **then**
  $mem(Des) := head(inp)$
  $inp := tail(inp)$
  $Proceed$
**if** $CT$ **is** WRITE **then**
  $out := out.eval(Expr)$
  $Proceed$
**if** $CT$ **is** WHILE **then**
  **if** $eval(Cond) \doteq true$ **then**
    $pc := yes(prog, pc)$
  **else** $pc := no(prog, pc)$

$Proceed \triangleq pc := next(prog, pc)$
$CT \triangleq occ(pc, prog)$
$Dest \triangleq lhs(pc, prog)$
$Src \triangleq rhs(pc, prog)$
$Des \triangleq dest(pc, prog)$
$Expr \triangleq arg(pc, prog)$
$Cond \triangleq cond(pc, prog)$

**if** $\varphi$ **then**
  **if** $\psi$ **then** $U_1$ $\triangleq$ **if** $\varphi \wedge \psi$ **then** $U_1$
  **else** $U_2$ $\triangleq$ **if** $\varphi \wedge \neg\psi$ **then** $U_2$



## Abstrakte Zustandsmaschinen: Interpretation als Zustandsübergangssystem

**Satz 2.1 (ASM und Zustandsübergangssystem)**

Sei $\Sigma$ eine Signatur, $\mathcal{A} \triangleq \langle \Sigma, \mathbb{I}, \mathbb{A}, R \rangle$ eine $\Sigma$-ASM. Dann ist $\mathcal{A}$ ein Zustandsübergangssystem $\mathcal{S} \triangleq (Q, I, \to_R)$ mit $Q \triangleq \mathbb{A}$, $I \triangleq \mathbb{I}$ und $\to_R \triangleq \{(\mathfrak{A}_1, \mathfrak{A}_2) : \mathfrak{A}_1 \in \mathbb{A} \wedge \mathfrak{A}_2 = next_{Update_{\mathcal{A}}(\mathfrak{A}_1)}(\mathfrak{A})\}$. Eine $\Sigma$-Algebra $\mathfrak{F} \in \mathbb{A}$ ist genau dann final, wenn $Update_{\mathcal{A}}(\mathfrak{F})$ inkonsistent oder $Update_{\mathcal{A}}(\mathfrak{F}) = \emptyset$ ist.

**Beweis**

Folgt direkt aus Definition 1.1, Definition 2.13, $\to_R \sqsubseteq \mathbb{A} \times \mathbb{A}$ und Definition 2.14.

## Diskussion

**Beobachtung**

Alle Begriffe aus Kapitel 1 sind für ASMs definiert:

- (Vollständiger) Lauf einer ASM
- Abstraktion einer ASM
- Beobachtbares Verhalten
- ASM $\mathcal{A}_1$ 1-1-simuliert (eingeschränkt) ASM $\mathcal{A}_2$
- Wenn $[\![\cdot]\!] : \mathcal{T}_{\text{PROG}} \to \langle \Sigma, \text{ASM}(\Sigma)\rangle$ den Programmen einer Programmiersprache eine Semantik als ASM zuordnet, ist auch der Begriff der korrekten Übersetung bzw. des korrekten Übersetzers definiert.
- (Eingeschränkte) $n$-$m$-Simulation

**Folgerung**

Auch die Sätze für Zustandsübergangssystem gelten für ASMs

- Horizontale Dekomposition
- Vertikale Dekomposition
- Satz 1.3 (Voraussetzung, dass $n$-$m$-Simulation eine eingeschränkte 1-1-Simulation auf beobachtbarem Verhalten impliziert)



## Statische und dynamische Funktionen

**Zusammenfassung des Beispiels (außer abstrakter Syntax, Navigation und Basistypen)**

**if** $CT$ **is** ASSIGN **then**
  $mem(Dest) := eval(Src)$
  $Proceed$
**if** $CT$ **is** READ **then**
  $mem(Des) := head(inp)$
  $inp := tl(inp)$
  $Proceed$

**if** $CT$ **is** WRITE **then**
  $out := out.eval(Expr)$
  $Proceed$
**if** $CT$ **is** WHILE **then**
  **if** $eval(Cond) \doteq true$ **then**
    $pc := yes(pc)$
  **else** $pc := no(pc)$

$Proceed \triangleq pc := next(prog, pc)$
$CT \triangleq occ(pc, prog)$
$Dest \triangleq lhs(pc, prog)$
$Src \triangleq rhs(pc, prog)$
$Des \triangleq dest(pc, prog)$
$Expr \triangleq arg(pc, prog)$
$Cond \triangleq cond(pc, prog)$

$lhs(o, p) \doteq occ(o.0, p)$
$rhs(o, p) \doteq occ(o.1, p)$
$dest(o, p) \doteq occ(o.0, p)$
$arg(o, p) \doteq occ(o.0, p)$
$cond(o, p) \doteq occ(o.0, p)$

$eval(mkdes(mkname(x))) \doteq mem(x)$
$eval(mkintconst(n)) \doteq n$
$eval(mkboolconst(b)) \doteq b$
$eval(mkgeq(e_1, e_2)) \doteq intgeq(eval(e_1), eval(e_2))$
$eval(mkplus(e_1, e_2)) \doteq intplus(eval(e_1), eval(e_2))$
$eval(mkminus(e_1, e_2)) \doteq intminus(eval(e_1), eval(e_2))$

- Die Funktionen $mem$, $pc$, $inp$, $out$, $eval$ und $prog$ sind dynamisch. alle anderen Funktionen sind statisch.
- $prog$ ist konstant; die übrigen dynamischen Funktionen sind nicht konstant.
- Die Funktionen $mem$, $pc$, $inp$, $out$ sind direkt; $eval$ ist abgeleitet.

**Definition 2.15 (Statische und dynamische Funktionen)**

Sei $\Sigma \triangleq \langle S, \sqsubseteq, F, F'\rangle$ eine Signatur und $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R\rangle$, so dass $\mathbb{A}$ die Menge der von $\mathbb{I}$ erreichbaren Zustände ist. Eine Funktion $f \in F \cup F'$ heißt **dynamisch** gdw. wenn es $\mathfrak{A}, \mathfrak{A}'$ gibt, so dass $[\![f]\!]_{\mathfrak{A}} \neq [\![f]\!]_{\mathfrak{A}'}$ ist. Ansonsten heißt $f \in F \cup F'$ heißt **statisch**. Eine dynamische Funktion $f$ heißt **konstant** gdw. für jeden Initialzustand $\mathfrak{I} \in \mathbb{I}$ die Eigenschaft $[\![f]\!]_{\mathfrak{A}} = [\![f]\!]_{\mathfrak{I}}$ für alle von $\mathfrak{I}$ aus erreichbaren Zustände $\mathfrak{A}$ gilt. Eine nicht-konstante dynamische Funktion $f$ heißt **direkt**, wenn es eine Regel **if** $\varphi$ **then** $U$ und Terme $t_1, \ldots, t_n, t$ gibt, so dass $f(t_1, \ldots, t_n) := t \in U$ ist. Andernfalls heißt $f$ **abgeleitet**.



## Diskussion

**Konvention**

Ab sofort betrachten wir noch ASMs $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R\rangle$, bei denen jedes $\mathfrak{A} \in \mathbb{A}$ von einem Initialzustand $\mathfrak{I} \in \mathbb{I}$ erreichbar ist.

**Korollar 2.2 (Eigenschaften von Zuständen einer ASM)**

Sei $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R\rangle$ eine ASM, $\Delta \subseteq \Sigma$ die Signatur der statischen Funktionen und $\Gamma$ die Signatur der konstanten Funktionen von $\mathcal{A}$. Dann gilt:

**i.** $\mathfrak{A}|_\Delta = \mathfrak{A}'|_\Delta$ für alle $\mathfrak{A}, \mathfrak{A}' \in \mathbb{A}$ (**statische Algebra**)

**ii.** Für jedes $\mathfrak{I} \in \mathbb{I}$ gilt: $\mathfrak{A}|_{\Delta\cup\Gamma} = \mathfrak{I}|_{\Delta\cup\Gamma}$ für alle von $\mathfrak{I}$ aus erreichbaren Zustände $\mathfrak{A}$.

**Beobachtungen**

- Bei Erstellung einer ASM sollten direkte und abgeleitete Funktionen strikt voneinander getrennt werden.

⇒ Keine Axiome für direkte dynamische Funktionen

⇒ Aus den Axiomen für abgeleitete dynamische Funktionen dürfen keine Aussagen über direkte dynamische Funktionen gefolgert werden können

☞ Gefahr von inkonsistenten Spezifikationen



## Zusammenfassung

- Abstrakte Zustandsmaschinen sind Zustandsübergangssysteme
- Zustände sind $\Sigma$-Algebren über einer Signatur $\Sigma$
- Zustandsübergangsregeln definieren die Änderung der Interpretation von Funktionssymbolen
- Initialzustände und die statischen Algebren werden über abstrakte Datentypen spezifiziert.
- Abgeleitete dynamische Funktionen werden ebenfalls durch Gleichungen definiert.

## 2.4 Modulare Konstruktion von Semantiken mit ASMs

### Ziele

- Formale Definition einer Semantik von ASMs ausgehend von einer Kernsprache, die dann sukzessive erweitert wird
- Erweiterungen sollen keine Änderungen der bisherigen Zustandsübergangsregeln und Axiome erfordern

⇒ Auch Korrektheitsbeweise können entsprechend modular aufgebaut werden.

1. Allgemeines Vorgehen
2. Die Sprache C−−
3. Steuerstrukturen (C0)
4. Verbunde und Verbundzeiger (C1)
5. Prozeduren und Funktionen



## 2.4.1 Allgemeines Vorgehen

### Schritte

- Definition der abstrakten Syntax
- Definition der Navigation
- Definition des Zustandsraums
- Definition der Zustandsübergangsregeln

### Modularisierung

- Schrittweise Erweiterung der abstrakten Syntax
- Schrittweise Erweiterung der Navigation
- Schrittweise Erweiterung der Definition des Zustandsraums
- Schrittweise Erweiterung der Zustandsübergangsregeln



## Diskussion

**Probleme**

- Bei der Definition des Zustandsraums müssen manchmal auch zukünftige Erweiterungen berücksichtigt werden.
- Wenn neue Sprachkonzepte eingeführt werden, müssen manchmal auch für alte Sprachkonstrukte neue Zustandsübergänge eingeführt werden.

**Beispiel 2.13: Probleme beim Erweitern der einfachen While-Sprache**

- Ausdrucksauswertung mit *eval* kann so nicht mehr formuliert werden, wenn Funktionsaufrufe eingeführt werden

⇒ Jeder Ausdruck hat einen Wert und diese Zuordnung ist zu definieren.

- Bei Einführung von Prozeduren und zusammengesetzten Datentypen können mehrere Objekte demselben Namen zugeordnet sein

⇒ Speicher $mem : \text{ID} \to \text{VALUE}$ funktioniert so nicht

⇒ Bindung von Variablen an Adressen und Adressen an Werte: $bind : \text{ID} \to \text{LOC}$ und $mem : \text{LOC} \to \text{VALUE}$

- Bei Einführung von Ausnahmen muss bei Zuweisung etc. die Fortsetzung zur Ausnahmebehandlung definiert werden.

☞ Ohne Ausnahmebehandlung endet man in finalem Zustand bei Auftreten einer Ausnahme (z.B. Division durch 0)

**Lösungen**

- Allgemeinere Konzepte zur Definition des Zustandsraums
- Neue Zustandsübergänge für alte Befehle durch neue Regeln

☞ Bei alten Befehlen werden die alten und neuen Aktualisierungen vereinigt



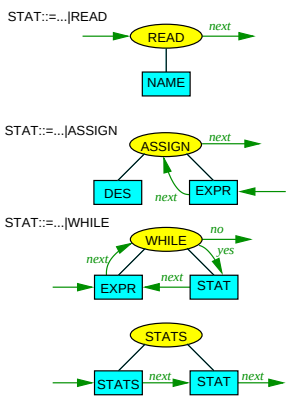
## Ausblick

**Weiteres Vorgehen in diesem Abschnitt**

- Informelle Vorstellung eines reduzierten C-Dialekts
- Definition des Zustandsraums
- Bei der Definition der Semantik wird die informelle Definition, die abstrakte Syntax, die Navigation und die Zustandsübergangsregel(n) eingeführt

**Notationen**

- Die abstrakte Syntax wird als Baum notiert. Die Knoten werden mit den entsprechenden Sorten gekennzeichnet. Rechtecke stehen für komplette Unterbäume. Alternativen werden als Produktion angegeben.
- Navigationen sind von der Sorte OCC und werden als Pfeile notiert.
- Jeder Unterbaum hat einen Anfang und mögliche Ausgänge, die durch ein– bzw. ausgehende Pfeile notiert werden.
- Eine Navigation auf einen kompletten Unterbaum hat als Ziel dessen Eingang
- Eine Navigation aus einem kompletten Unterbaum hat als Quellen alle Ausgänge des Unterbaums



## 2.4.2 Die Sprache C--

**Programme**

*prog* ::= *decl*$^*$ *block*

- Die in den Deklarationen vereinbarten Namen sind im Hauptprogram *block* sichtbar (**Gültigkeitsbereich**)
- Die in den Deklarationen vereinbarten Namen müssen eindeutig sein.
- Die in den Deklarationen vereinbarten Variablen, Prozeduren und Typen heißen **global**.
- Die Ausführung eines Programms führt den Block aus.

**Basistypen und Anpassung**

*type* ::= **id**

- Basisdatentypen sind ganze Zahlen (Typ `int`), Gleitkommazahlen (Typ `float`) und Wahrheitswerte (Typ `bool`) mit den üblichen Operationen und Konstanten
- Die Darstellung ganzer Zahlen entspricht der Darstellung ganzer Zahlen auf der Zielmaschine. Die Arithmetik ist Ringarithmetik (z.B ist `maxint` + 1 = `minint`)
- Die Gleitkommazahlen werden gemäß IEEE 754-1985 dargestellt. Die Genauigkeit entspricht der Wortlänge der Zielmaschine.
- Ganze Zahlen können automatisch in Gleitkommazahlen konvertiert werden.
- Konstanten werden in der selben Syntax wie C notiert



## Deklarationen

**Zusammengesetzte Typen**

*decl* ::= *vardecl*|*typedecl*|*procdecl*
*typedecl* ::= *struct*|*class*
*struct* ::= `struct` **id** {*vardecl*$^*$}
*class* ::= `class` **id** {*vardecl*$^*$}

- Ein Verbundtyp (`struct`) definiert ein kartesisches Produkt. Verbundobjekte (`struct`) bestehen aus Teilobjekten, die durch die Variablendeklarationen des Verbundtyps gegeben sind (**Verbundfelder**)
- Ein Klassentyp (`class`) definiert einen Zeigertyp auf Verbundtypen. Ein Klassenobjekt ist ein Zeiger auf ein Verbundobjekt, dessen Teilobjekte durch die Verbundfelder des Klassentyps definiert sind. Zeiger können auch leer sein (`NULL`)
- **id** ist jeweils der Name des Verbund– bzw. Klassentyps.
- Die Verbundfeldnamen müssen paarweise verschieden sein.

**Variablendeklarationen**

*vardecl* ::= *type* **id;**

- Eine Variablendeklaration vereinbart ein Objekt, in dem Werte des Typs *type* gespeichert werden können
- Der Bezeichner **id** ist der Name der Variablen. Mit diesem Namen kann die Variable in ihrem Gültigkeitsbereich angesprochen werden.



## Zugriffspfade

*des* ::= **id**|*fieldaccess*
*fieldaccess* ::= *des* **.id**

- Zugriffspfade verweisen auf Objekte
- Falls ein Zugriffspfad aus einem Bezeichner **id** besteht, muss dieses Objekt im Kontext durch eine Variablendeklaration vereinbart sein. In diesem Fall verweist der Zugriffspfad auf dieses Objekt.
- Bei einem Feldzugriff muss der Typ des Zugriffspfads *des* ein Verbund– oder Klassentyp sein, der ein Feld mit dem Namen **id** enthält.
- Falls *des* Verbundtyp ist, bezeichnet der Zugriffspfad das durch **id** definierte Teilobjekt
- Falls *des* Klassentyp ist, muss der Wert von *des* $\neq$ `NULL` sein, ansonsten bricht das Programm mit einer `NullPointerDereferenceException` ab.
- Wenn der Wert von *des* $\neq$ `NULL` ist, wird zunächst das durch den Zeiger verwiesene Objekt ermittelt (**Dereferenzieren**). Der Klassenfeldzugriff verweist dann auf das durch **id** definierte Teilobjekt.



## Ausdrücke (Syntax und statische Semantik)

*expr* ::= **null**|**intconst**|**fltconst**|**boolconst**|*des*|*unexpr*|*binexpr*|*call*|*new*|(*expr*)
*unexpr* ::= *unop expr*
*unop* ::= `!`|`-`|`+`
*binexpr* ::= *expr binop expr*
*unop* ::= *eqop*|*relop*|*addop*|*mulop*
*eqop* ::= `==`|`!=`
*relop* ::= `<`|`>`|`<=`|`>=`
*addop* ::= `+`|`-`
*mulop* ::= `*`|`/`|`%`
*boolop* ::= `||`|`&&`
*new* ::= `new` *type*

- Die Prioritäten der Operatoren sind in der folgenden Reihenfolge definiert: unäres + und –, *mulop*, *addop*, *relop*, *eqop*, `&&`, `||`, `!`.
- Die binären Operatoren (außer `relop`) sind linksassoziativ geklammert.
- Der Typ eines Zugriffspfads ist der Typ des durch ihn bezeichneten Objekts.
- Boolesche Operatoren erwarten von ihren Operanden den Typ `bool`. Boolesche Ausdrücke haben den Typ `bool`.
- Gleichheitsausdrücke erwarten entweder von beiden Operanden denselben Typ oder einer der Operanden muss vom Typ `int` und der andere vom Typ `bool` sein. Der Typ eines Gleichheitsausdrucks ist `bool`.
- Der Operator `%` erwarten von beiden Operanden den Typ `int`. Der Ausdruck ist vom Typ `int`.
- Relationale Ausdrücke (*relop*) und arithmetische Ausdrücke (außer `%`) erwarten von ihren Operanden den Typ `int` oder `float`. Der Typ eines relationalen Ausdrucks ist `bool`. Der Typ eines arithmetischen Ausdrucks ist `int`, wenn alle seine Operanden vom Typ `int` sind, ansonsten ist der Typ `float`.
- Der Typ einer `new`-Operation muss eine Klasse sein. Dieser Typ ist der Typ des Ausdrucks

## Ausdrücke (dynamische Semantik)

- Ausdrücke werden ausgewertet. Die Auswertereihenfolge arithmetischer und relationaler Ausdrücke ist implementierungsabhängig.
- Boolesche Ausdrücke werden durch Kurzauswertung ausgewertet.
- Der Wert einer Konstanten entspricht dem im Programm bezeichneten Wert
- Der Wert eines Zugriffspfad *des* ist der Inhalt des Objekts, auf das *des* verweist.
- Der Wert eines unärer Ausdrucks wendet den Operator auf den Wert des Operanden an (`!` ist die logische Negation)
- Die Werte binärer Ausdrücke verknüpfen die beiden Werte mit dem Operator des Ausdrucks (`||` ist die logische Disjunktion, `&&` die logische Konjunktion und `%` der Rest bzgl. der Division)
- Hat der rechte Operand von `/` oder `%` den Wert 0, dann bricht das Programm mit der Ausnahme `DivByZeroException` ab.
- Eine `new`-Operation erzeugt ein Objekt der Klasse und liefert einen Verweis auf dieses Objekt



## Anweisungen

**Zuweisung**

*stat* ::= *assign*|*read*|*write*|*if*|*while*|*break*|*continue*|*call* `;`|*return*|*block*
*assign* ::= *des* `=` *expr* `;`

- Der Typ von *expr* muss an den Typ *des* anpassbar sein.
- Der Zugriffspfad wird zu einem Verweis auf ein Objekt ausgewertet und der Ausdruck wird ausgewertet. Der Wert des Ausdrucks wird in das durch den Zugriffspfad verwiesene Objekt geschrieben. Die Auswertereihenfolge ist implementierungsabhängig.
- Danach wird mit der nächsten Anweisung fortgefahren.
- Wird an ein Verbundobjekt zugewiesen, dann werden die Werte der Verbundfelder in das Teilobjekt, auf das entsprechende Verbundfeld verweist, geschrieben. Wird an ein Verbundzeigerobjekt zugewiesen wird der Zeiger in das Objekt, auf das die linke Seite verweist, geschrieben.



## Anweisung(en)

**Lese-Anweisung**

*read* ::= `read ( ` *des* ` ) ;`

- Der Zugriffspfad darf kein Verbund– oder Klassentyp haben.
- Die Lese-Anwweisung bestimmt das Objekt, auf das der Zugriffspfad verweist und schreibt dort den aktuellen Wert des Eingabestroms. Die aktuelle Position des Eingabestroms auf den nächsten Wert fortgeschalten.
- Danach wird mit der nächsten Anweisung fortgefahren.

**Schreibe-Anweisung**

*write* ::= `write ( ` *expr* ` ) ;`

- Der Ausdruck darf kein Verbund– oder Klassentyp haben.
- Die Schreibe-Anwweisung wertet den Ausdruck aus und schreibt dessen Wert an das Ende des Ausgabestroms.
- Danach wird mit der nächsten Anweisung fortgefahren.



## Anweisung(en)

**Verzweigung**

*if* ::= `if ( ` *expr* ` )` *stat* [`else` *stat*]

- Der Ausdruck muss vom Typ `bool` sein.
- Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die erste Anweisung ausgeführt. Danach wird die Anweisung nach der Verzweigung ausgeführt. Ist das Ergebnis `false` und keine `else`-Anweisung vorhanden, dann wird die Anweisung nach der Verzweigung ausgeführt. Ansonsten wird zuerst die `else`-Anweisung und dann die Anweisung nach der Verzweigung ausgeführt.

**Schleifen**

*if* ::= `while ( ` *expr* ` )` *stat*

- Der Ausdruck muss vom Typ `bool` sein.
- Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die Anweisung ausgeführt. Danach wird wieder die Schleife ausgeführt.Ist das Ergebnis `false`, dann wird die Anweisung nach der Schleife ausgeführt.

## Anweisung(en)

**Schleifenabbruch und –fortsetzung**

*break* ::= `break ;`
*continue* ::= `continue ;`

- Eine `break`– oder `continue`-Anweisung darf nur innerhalb des Blocks einer Schleifenanweisung stehen.
- Bei Ausführen einer `break`-Anweisung wird die Schleife abgebrochen und danach die Anweisung nach der Schleife ausgeführt.
- Bei Ausführen einer `continue`-Anweisung wird die aktuelle Ausführung des Schleifenrumpfs abgebrochen und danach die Schleifenanweisung ausgeführt.

**Blockanweisung**

*block* ::= { *vardecl*$^*$ *stat*$^*$ }

- Die durch die Variablendeklarationen eingeführten Namen eines Blocks müssen paarweise verschieden sein.
- Die Namen der deklarierten Variablen sind nur im Block sichtbar. Sie verdecken Bezeichner desselben Namens, die außerhalb des Blocks sichtbar sind.
- Die Ausführung eines Blocks führt alle seine Anweisungen nacheinander aus. Danach wird die Anweisung nach dem Block ausgeführt.



## Prozeduren und Funktionen (Syntax und statische Semantik)

*procdecl* ::= *type* **id** ( *pars* ) *block*
*pars* ::= [*type* **id** (, *type* **id**)$^*$]
*call* ::= **id** ( *args*)
*args* ::= [*expr* (, `expr`)$^*$]
*return* ::= `return` [*expr*]

- Die Parameter sind im Prozedurrumpf (*block*) sichtbar.
- Die Namen der Parameter müssen paarweise verschieden sein.
- Der Typ in einer Prozedurdeklaration heißt **Rückgabetyp**. Ist der Rückgabetyp `void` so redet man von **Prozeduren**, sonst von **Funktionen**.
- Prozeduraufrufe sind Anweisungen, Funktionsaufrufe sind Ausdrücke. Der Typ eines Funktionsaufrufs ist der Rückgabetyp der aufgerufenen Funktion.
- Bei einem Aufruf einer Prozedur/Funktion muss die Anzahl der Parameter der aufgerufenen Prozedur **id** mit der Anzahl der Argumente übereinstimmen. Vom *i*-ten Argument wird der Typ des *i*-ten Parameters erwartet.
- Eine `return`-Anweisung darf nur innerhalb eines Prozedur– oder Funktionsrumpfs vorkommen. Bei einer Prozedur darf nur die `return`-Anweisung ohne Ausdruck verwendet werden. Bei einer Prozedur muss die `return`-Anweisung mit Ausdruck verwendet werden. Von diesem Ausdruck wird erwartet, dass er Typ gleich dem Rückgabetyp der Funktion ist.



## Prozeduren und Funktionen (dynamische Semantik)

- Mit dem Prozedur–/Funktionsaufruf werden den Parametern Objekte zugeordnet.
- Bei einem Prozedur– oder Funktionsaufruf werden die Argumente ausgewertet und an die Parameter positional übergehen, d.h. der Wert eines Arguments wird in das Objekt geschrieben, auf das der Parameter verweist. Danch wird der Prozedurrumpf ausgeführt. Die Reihenfolge der Auswertung der Argumente ist implementierungsabhängig.
- Ein Prozedur– oder Funktionsaufruf **kehrt zurück**, wenn die letzte Anweisung des Prozedurrumpfs oder eine `return`-Anweisung ausgeführt wurde.
- Nach Rückkehr von einem Prozeduraufruf wird mit die Anweisung nach dem Aufruf ausgeführt.
- Nach Rückkehr von einem Funktionsaufruf wird mit der Ausführung der Anweisung, in der der Aufruf vorkam, fortgefahren. Der Wert eines Funktionsaufrufs ist der Wert des Ausdrucks der ausgeführten `return`-Anweisung.



## 2.4.3 Steuerstrukturen (C0)

**Ausgangspunkt**

- Spezifikationen INT für ganze Zahlen (64-Bit, 2-Komplement, Ringarithmetik
- FLOAT für Gleitkommazahlen (64-Bit 754-1985)
- Spezifikation BOOL für Wahrheitswerte
- **spec** VALUE **extends** INT, FLOAT, BOOL
  **sorts** VALUE
  **subsorts** INT $\sqsubseteq$ VALUE, FLOAT $\sqsubseteq$ VALUE, BOOL $\sqsubseteq$ VALUE
- Spezifikation OCC für Listen ganzer Zahlen
- Spezifikation LISTOFVAL für Listen von Werten
- Spezifikation PROG **extends** OCC, LISTOFVAL $\cdots$, die die abstrakte Syntax, Attributierung und Navigation definiert

**Vorgehen**

- Definition des Zustandsraums und des Anfangszustands
- Graphische Angabe der abstrakten Syntax und der Navigationen
- Pro Konzept eine Zustandsübergangsregel

## Zustandsraum

### Umgebung und Speicher

- **Umgebungen** binden Variablenbezeichner an Adressen:

**spec** ENV **extends** PROG

| | | | | |
|---|---|---|---|---|
| **sorts** | ENV, LOC, IDLIST | | | |
| **operations** | $mkenv$ : | | $\rightarrow$ ENV | erzeugt leere Umgebung |
| | $newbind$ : | ENV $\times$ ID $\times$ LOC | $\rightarrow$?ENV | neue Zuordnung |
| | $isUsed$ | ENV $\times$ LOC | $\rightarrow$ BOOL | wahr gdw. Adresse gebunden |
| | $isBound$ | ENV $\times$ ID | $\rightarrow$ BOOL | prüft Bindung |
| | $bind$ : | ENV $\times$ ID | $\rightarrow$?LOC | ermittelt Bindung |
| | $delbind$ : | ENV $\times$ ID | $\rightarrow$ ENV | beseitigt Bindung |
| | $(++)$ : | ENV $\times$ ENV | $\rightarrow$ ENV | Anhängen zweier Umgebungen |
| | $(\setminus)$ : | ENV $\times$ IDLIST | $\rightarrow$ ENV | |

**axioms**

$$D(newbind(e,x,l)) \Leftrightarrow isUsed(e,l) \doteq false$$
$$D(bind(e,x)) \Leftrightarrow isBound(e,x) \doteq true$$
$$bind(newbind(e,x,l),x) \doteq l$$
$$\neg x \doteq y \Rightarrow bind(newbind(e,y,l),x) \doteq bind(e,x)$$
$$delbind(mkenv,x) \doteq mkenv$$
$$delbind(newbind(e,x,l),x) \doteq e$$
$$\neg x \doteq y \Rightarrow delbind(newbind(e,y,l),x) \doteq delbind(e,x)$$

Übrige Axiome sind zur Übung überlassen

- ENV $\models \forall x, y : \text{ID} \;\; \forall e : \text{ENV}\bullet$
$$D(bind(e,x)) \wedge D(bind(e,y)) \wedge bind(e,x) \doteq bind(e,y) \Rightarrow x \doteq y$$
- **Speicher** binden Werte an Adressen: $mem : LOC \rightarrow ?VALUE$
- $e \setminus i$ beseitigt die Bindungen der Bezeichner in Liste $i$ aus $e_1$



## Zustandsraum und Anfangszustand

### Zustand

Ein Zustand besteht aus einer Umgebung, einem Speicher, einem Eingabe– und Ausgabestrom, dem Programm sowie einem Befehlszeiger:

**spec** STATE **extends** ENV

| | | | |
|---|---|---|---|
| **operations** | $env$ : | | $\rightarrow$ ENV |
| | $mem$ : | LOC | $\rightarrow$?VALUE |
| | $inp$ : | | $\rightarrow$ LISTOFVAL |
| | $out$ : | | $\rightarrow$ LISTOFVAL |
| | $prog$ : | | $\rightarrow$ PROG |
| | $pc$ : | | $\rightarrow$ OCC |

### Anfangszustand

Am Anfang ist der Speicher leer, der Ausgabestrom leer, die Umgebung leer und der Befehlszeiger verweist auf die erste auszuführende Anweisung

**spec** INITSTATE **extends** STATE

**axioms**

$$\neg D(mem(x))$$
$$out \doteq []$$
$$env \doteq mkenv$$
$$pc \doteq first(prog)$$

- $\Rightarrow$ Bezug auf Attribute des attributierten Strukturbaums
- Die globalen Variablen des Hauptprogramms sind bereits zur Übersetzungszeit an Objekte gebunden
- $\Rightarrow$ Statische Funktion $globenv$ : PROG $\rightarrow$ ENV



## Ausdrucksauswertung

### Problem

Definition mit *eval* ist wegen Funktionsaufrufen nicht erweiterbar

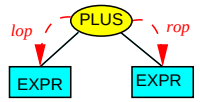
- $\Rightarrow$ Jeder Operator ist auszuwertender Befehl
- $\Rightarrow$ Ausdruckswert wird Knoten zugeordnet
- $\Rightarrow$ Direkte dynamische Funktion $val$ : OCC $\rightarrow$?VALUE
- Zugriffspfade bestimmen zusätzlich den Ort des Objekts, auf das sie verweisen: dynamische Funktion $loc$ : OCC $\rightarrow$?LOC

### Beobachtungen

- Ausdrücke benötigen die Werte ihrer Operanden
- Schleifen und Verzweigungen benötigen den Wert ihrer Bedingung
- Die Schreibanweisung benötigt den Wert ihres Argument
- $\Rightarrow$ Einführung von Datenflusskanten analog der Steuerflusskanten:

| | | | |
|---|---|---|---|
| $lop$ : | PROG $\times$ OCC | $\rightarrow$?OCC | linker Operand eines Ausdrucks |
| $rop$ : | PROG $\times$ OCC | $\rightarrow$?OCC | rechter Operand eines Ausdrucks |
| $opd$ : | PROG $\times$ OCC | $\rightarrow$?OCC | Operand eines unären Ausdrucks |
| $cond$ : | PROG $\times$ OCC | $\rightarrow$?OCC | Bedingung einer Schleife oder Verzweigung |
| $arg$ : | PROG $\times$ OCC | $\rightarrow$?OCC | Auszudruckender Ausdruck |

- ☞ Use-Def-Beziehungen
- Notation durch gestrichelte Pfeile:
- Eine Datenflusskante auf einen kompletten Unterbaum hat als Ziel dessen Wurzel
- Die Quelle einer Datenflusskante ist immen ein AST-Knoten






## Ausdrucksauswertereihenfolgen

### Problem

Spezifikation der implementierungsabhängigen Reihenfolge

- $\Rightarrow$ Aus Sicht der Sprachsemantik sind alle Reihenfolgen zulässig, nur die linken Operanden von AND bzw. OR müssen vor den rechten ausgewertet werden
- $\Rightarrow$ Nur der Datenfluss muss eingehalten werden

### Beobachtung

Eine Programmiersprache definiert eine Halbordnung, welche Operatoren vor anderen Operatoren berechnet werden müssen.

- $\Rightarrow$ Jede topologische Sortierung dieser Halbordnung ist eine gültige Reihenfolge
- ☞ Bei strenger Links-Rechts-Auswertereihenfolge (wie z.B. in Java oder C#) ist die Halbordnung sogar eine Totalordnung.

## Ausdrucksauswertereihenfolgen

**Formalisierung**

- Wenn ein Ausdruck ausgewertet werden soll, wird irgendein Knoten $k$ gewählt, der minimal bzgl. der Halbordnung ist.
- Nichtdeterministische Auswahl einer topologischen Sortierung, die $k$ als erstes Element enthält
- Abarbeitung dieser topologischen Sortierung

$\Rightarrow$ Weitere Form von Zustandsübergangsregeln erforderlich: **choose** $x : S \bullet \varphi$ **from** $U$Jede Menge $\mathbb{U} \triangleq U[t/x]$ von Aktualisierungen mit $\mathfrak{A} \models \varphi[t/x]$ ist eine Aktualisierungsmenge im Zustand $\mathfrak{A}$

**ASM-Zustandsübergangsregel**

- Dynamische Funktion *toBeEval* :⟶ LISTOFOCC
- Falls ein Ausdruck das erste Mal betreten wird, wird nichtdeterministisch eine topologische Sortierung gewählt:

**if** *next*(*pc*, *prog*) **is** EXPR **then**
**choose** *o* : OCC • *isMin*(*o*, *next*(*pc*, *prog*)) **from** *pc* := *o*
**choose** *l* : LISTOFOCC • *isTopOrder*(*l*, *next*(*pc*, *prog*)) **from** *toBeEval* := *l*

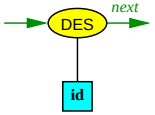
**Bemerkung**

Wir betrachten hier der Einfachheit halber eine Links-Rechts-Auswertereihenfolge



## Zugriffspfade

- Falls ein Zugriffspfad aus einem Bezeichner **id** besteht, muss dieses Objekt im Kontext durch eine Variablendeklaration vereinbart sein. In diesem Fall verweist der Zugriffspfad auf dieses Objekt.
- Der Wert eines Zugriffspfad *des* ist der Inhalt des Objekts, auf das *des* verweist.




**if** *CT* **is** DES **then**
*loc*(*pc*) := *Loc*
*Coerce*(*val*(*pc*), *mem*(*Loc*), *pc*)
*Proceed*

| | | |
|---|---|---|
| *CT* | $\triangleq$ | *occ*(*prog*, *pc*) |
| *Env* | $\triangleq$ | *env*++*globenv*(*prog*) |
| *Loc* | $\triangleq$ | *bind*(*Env*, *id*(*pc*, *prog*)) |
| *Pri*(*p*) | $\triangleq$ | *pri*(*prog*, *p*) |
| *Post*(*p*) | $\triangleq$ | *post*(*prog*, *p*) |
| *Coerce*(*src*, *dest*, *p*) | $\triangleq$ | **if** ¬*Pri*(*p*) = *Post*(*p*) **then** *dest* := *coerce*(*src*, *Pri*(*p*), *Post*(*p*)) **else** *dest* := *src* |
| *Proceed* | $\triangleq$ | *pc* := *next*(*prog*, *pc*) |

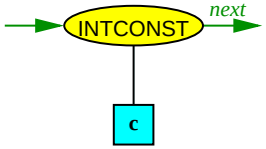
**Statische Funktionen:**

- *id* : OCC × PROG →?ID zur Bestimmung des Bezeichners eines Zugriffspfads
- *pri*, *post* : PROG × OCC →?TYPE ist der Typ des Ausdrucks bzw. der vom Kontext erwartete Typ
- *coerce* : VALUE × TYPE × TYPE →?VALUE konvertiert einen Wert des ersten Typs in einen äquivalenten Wert des zweiten Typs



## Konstanten

Der Wert einer Konstanten entspricht dem im Programm bezeichneten Wert




**if** *CT* **is** CONST **then**
*Coerce*(*val*(*pc*), *const*(*pc*, *prog*), *pc*)
*Proceed*

**Untersorten:** INTCONST ⊑ CONST, FLTCONST ⊑ CONST, BOOLCONST ⊑ CONST

**Statische Funktion:** *const* : *OCC* × *PROG* →?*VALUE* zur Bestimmung der Konstanten

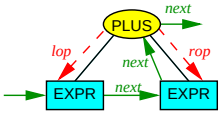


## Binärer Operationen

**Binäre Ausdrucke (ohne Divisionsoperatoren)**

- Die Werte binärer Ausdrücke verknüpfen die beiden Werte mit dem Operator des Ausdrucks
- Ganze Zahlen werden – falls erforderlich – automatisch in Gleitkommazahlen konvertiert werden.

**if** *CT* **is** PLUS **then**
*Coerce*(*val*(*pc*), *Sum*, *pc*)
*Proceed*

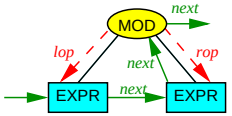



| | | |
|---|---|---|
| *Lop* | $\triangleq$ | *val*(*lop*(*pc*, *prog*)) |
| *Rop* | $\triangleq$ | *val*(*rop*(*pc*, *prog*)) |
| *Sum* | $\triangleq$ | *intplus*(*Lop*, *Rop*) |

Multiplikation und Subtraktion werden analog behandelt.

**Division**

- Die Werte binärer Ausdrücke verknüpfen die beiden Werte mit dem Operator des Ausdrucks
- Hat der rechte Operand von / oder % den Wert 0, dann bricht das Programm mit der Ausnahme `DivByZeroException` ab.




**if** *CT* **is** MOD **then**
**if** *Rop* = 0 **then**
*exc* := *divByZero*
*pc* := *handle*(*divByZero*, *pc*)
**else** *Coerce*(*val*(*pc*), *Rem*, *pc*)
*Proceed*

*Rem* $\triangleq$ *mod*(*Lop*, *Rop*)

- *handle* : EXCEPTION → OCC spezifiziert die Anweisung, mit der bei Ausnahme fortgefahren werden muss
- Division wird analog behandelt.

## Unäre Ausdrücke

Der Wert eines unärer Ausdrucks wendet den Operator auf den Wert des Operanden an.


**if** *CT* **is** NOT **then**
  $val(pc) := not(Opd)$
  *Proceed*

$Opd \triangleq val(opd(pc, prog))$

Unäres Minus und Plus wird analog behandelt unter Berücksichtigung von Anpassungen



## Anweisungsfolgen

### Leere Anweisungsfolge

Eine leere Anweisungsfolge hat keinen Effekt.

Stats::=NoStats


**if** *CT* **is** NOSTATS **then** *Proceed*

### Nichtleere Anweisungsfolge

Die Ausführung eines Blocks führt alle seine Anweisungen nacheinander aus.

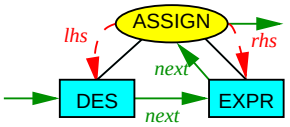

Keine Zustandsübergangsregel erforderlich, da das Fortschalten des Befehlszeigers bei den einzelnen Anweisungen erfolgt



## Anweisungen: Zuweisung

- Der Zugriffspfad wird zu einem Verweis auf ein Objekt ausgewertet und der Ausdruck wird ausgewertet. Der Wert des Ausdrucks wird in das durch den Zugriffspfad verwiesene Objekt geschrieben.
- Danach wird mit der nächsten Anweisung fortgefahren.

STAT ::= ...|ASSIGN|...




**if** *CT* **is** ASSIGN $\wedge$ $isAtomic(Pri(lhs))$ **then**
  $mem(loc(lhs)) := val(rhs)$
  *Proceed*

- Das geht nur für atomare Werte

$\Rightarrow$ Statische Funktion $isAtomic$ : TYPE $\rightarrow$ BOOL

- $isAtomic(x) \doteq true$ für die Typen $x$ = INT, BOOL und FLOAT



## Anweisungen: Lesen und Schreiben

**Leseanweisung**

- Die Lese-Anweisung bestimmt das Objekt, auf das der Zugriffspfad verweist und schreibt dort den aktuellen Wert des Eingabestroms. Die aktuelle Position des Eingabestroms auf den nächsten Wert fortgeschalten.
- Danach wird mit der nächsten Anweisung fortgefahren.

STAT ::= ...|READ|...


**if** *CT* **is** READ **then**
  $mem(loc(dest)) := head(inp)$
  $inp := tail(inp)$
  *Proceed*

Der Zugriffspfad darf kein Verbund– oder Klassentyp haben.

**Schreibeanweisung**

- Die Schreibe-Anwweisung wertet den Ausdruck aus und schreibt dessen Wert an das Ende des Ausgabestroms.
- Danach wird mit der nächsten Anweisung fortgefahren.

STAT ::= ...|WRITE|...


**if** *CT* **is** WRITE **then**
  $out := out.val(Arg)$
  *Proceed*

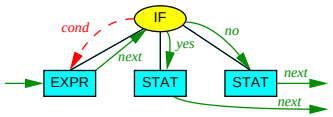
$Arg \triangleq arg(pc, prog)$

## Anweisungen: Verzweigungen und Schleifen

### Verzweigung

Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die erste Anweisung ausgeführt. Danach wird die Anweisung nach der Verzweigung ausgeführt. Ist das Ergebnis `false` und keine `else`-Anweisung vorhanden, dann wird die Anweisung nach der Verzweigung ausgeführt. Ansonsten wird zuerst die `else`-Anweisung und dann die Anweisung nach der Verzweigung ausgeführt.

STAT ::= ...|IF|...

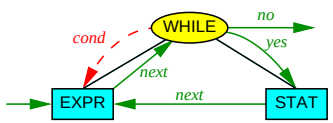



**if** $CT$ **is** IF **then**
  **if** $val(cond) \doteq true$ **then**
    $pc := yes(pc, prog)$
  **else** $pc := no(pc, prog)$

### Schleifen

Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die Anweisung ausgeführt. Danach wird wieder die Schleife ausgeführt. Ist das Ergebnis `false`, dann wird die Anweisung nach der Schleife ausgeführt.

STAT ::= ...|WHILE|...




**if** $CT$ **is** WHILE **then**
  **if** $val(cond) \doteq true$ **then**
    $pc := yes(pc, prog)$
  **else** $pc := no(pc, prog)$



## Anweisungen: Schleifenabbruch– und Fortsetzung

### Grundprinzip

- Jede Schleife kennt über *no* die Anweisung nach der Schleife und den ersten Befehl der Schleifenbedingung
- Diese werden über $break : \text{PROG} \times \text{OCC} \rightarrow ?$ bzw. $continue : \text{PROG} \times \text{OCC} \rightarrow ?$ auf die Anweisungen des Schleifenrumpfs fortgeschrieben

$\Rightarrow$ Teil der statischen Semantik

### Break-Anweisung

Bei Ausführen einer `break`-Anweisung wird die Schleife abgebrochen und danach die Anweisung nach der Schleife ausgeführt

STAT ::= ...|BREAK|...


**if** $CT$ **is** BREAK **then**
  $pc := break(prog, pc)$

### Continue-Anweisung

Bei Ausführen einer `continue`-Anweisung wird die Schleife abgebrochen und danach und danach die Schleifenanweisung ausgeführt.

STAT ::= ...|CONTINUE|...

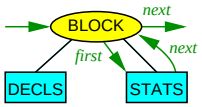

**if** $CT$ **is** CONTINUE **then**
  $pc := continue(prog, pc)$



## Blockanweisung

Die Ausführung eines Blocks führt alle seine Anweisungen nacheinander aus. Danach wird die Anweisung nach dem Block ausgeführt.

STAT ::= ...|BLOCK|...




### Beobachtungen

- Den lokalen Variablen des Blocks müssen bei Betreten **unterschiedliche** Objekte zugeordnet werden und diese Zuordnung bei Verlassen wieder aufgehoben werden.

$\Rightarrow$ Der Blockknoten wird zwei Mal betreten

$\Rightarrow$ Zwei Zustandsübergangsregeln

$\Rightarrow$ Dynamische Funktion $entered : OCC \rightarrow ?\text{BOOL}$

### Zustandsübergangsregeln

**if** $CT$ **is** BLOCK $\wedge$ $entered(pc) \doteq false$ **then**
  $AllocateVars(locVars(prog, pc))$
  $entered(pc) := true$
  $pc := first(prog, pc)$

**if** $CT$ **is** BLOCK $\wedge$ $entered(pc) \doteq true$ **then**
  $DeAllocateVars(locVars(prog, pc))$
  $entered(pc) := false$
  $Proceed$

wobei $locVars : \text{PROG} \times \text{OCC} \rightarrow ?\text{IDLIST}$ die Liste der lokalen Variablen eines Blocks ergibt.



## Allokation und Deallokation von Objekten

### Allokation

**Idee:** Bestimmung einer Umgebung der lokalen Variablen mit Bindungen an paarweise unterschiedliche Objekte, die alle verschieden von den bisher gebundenen Objekten sind.

**Eigenschaften:** $Good(e, ids) \triangleq$
$\forall x : \text{ID} \bullet isBound(e, x) \doteq true \Leftrightarrow isElem(x, ids) \doteq true \wedge$
$\forall x, y : \text{ID} \bullet D(bind(e, x)) \wedge D(bind(e, y)) \wedge bind(e, x) \doteq bind(e, y) \Rightarrow x \doteq y \wedge$
$\forall x : \text{ID} \bullet isBound(e, x) \doteq true \Rightarrow IsUsed(bind(e, x)) \doteq false$
wobei das Makro $IsUsed(l)$ angibt, ob ein Verweis $l$ schon benutzt wurde (wird später definiert).

**Makro:** $AllocateVars(ids) \triangleq$ **choose** $e : \text{ENV} \bullet Good(e, ids)$ **do**
  $env := e{+}{+}env$

### Deallokation

Bindungen werden wieder bei Verlassen des Blocks zerstört:
$DeAllocateVars(ids) \triangleq \quad env := env \setminus ids$

## Zusammenfassung

- Der Zustandsraum ist nun:

  **spec** STATE **extends** ENV
  **sorts** EXCEPTION
  **operations**

| | | | |
|---|---|---|---|
| *env* : | | → | ENV |
| *mem* : | LOC | → | ?VALUE |
| *inp* : | | → | LISTOFVAL |
| *out* : | | → | LISTOFVAL |
| *prog* : | | → | PROG |
| *pc* : | | → | OCC |
| *exc* : | | → | EXCEPTION |
| *val* : | OCC | → | ?VALUE |
| *loc* : | OCC | → | ?LOC |
| *entered* : | OCC | → | ?BOOL |

- Makro *IsUsed*, das angibt, ob ein Verweis *l* schon benutzt wurde, muss noch definiert werden.

☞ Es muss aber auf jeden Fall gelten:
$isUsed(l) \doteq true \Rightarrow IsUsed(env, l) \doteq true$



## 2.4.4 Verbunde und Klassen (C1)

**Verbundobjekte**

- Werte sind Tupel. Der Zugriff auf die Komponenten erfolgt über die Namen der Felder

⇒ Unter jedem der Namen ist ein Wert gespeichert
⇒ Für jeden der Namen ist ein Verweis auf ein Objekt zugeordnet
⇒ Zugriff auf Verbundfelder ergeben den dem Namen zugeordneten Verweis
⇒ Verbunde sind ebenfalls Umgebungen

- Aber bei Zuweisung müssen explizit die einzelnen Verbundfelder kopiert werden

**Klassen**

- Klassen sind Zeiger auf Verbunde

⇒ Es werden Verweise auf Verbunde gespeichert

- Bei Zugriff auf eine Komponente muss erst diesem Verweis gefolgt werden.

**Zustandsraum**

**spec** $\text{ENV}_1$ **extends** ENV
**subsorts** LOC ⊑ VALUE
**operations**

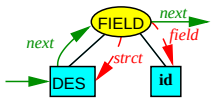
| | | | |
|---|---|---|---|
| *floc* : | LOC × ID | → | LOC |
| *null* : | | → | LOC |

**spec** $\text{STATE}_1$ **extends** STATE, $\text{ENV}_1$
**spec** $\text{INITSTATE}_1$ **extends** $\text{STATE}_1$, INITSTATE



## Verbundfeld– und Klassenfeldzugriffe

- Falls *des* Verbundtyp ist, bezeichnet der Zugriffspfad das durch **id** definierte Teilobjekt
- Falls *des* Klassentyp ist, muss der Wert von *des* ≠ `NULL` sein, ansonsten bricht das Programm mit einer `NullPointerDereferenceException` ab.
- Wenn der Wert von *des* ≠ `NULL` ist, wird zunächst das durch den Zeiger verwiesene Objekt ermittelt (**Dereferenzieren**). Der Klassenfeldzugriff verweist dann auf das durch **id** definierte Teilobjekt.




**if** *CT* **is** FIELD **then**
  **if** *isClass*(*Pri*(*pc*)) **then**
    **if** *loc*(*pc*) = *null* **then**
      *exc* := *nullPointerDeref*
      *pc* := *handle*(*nullPointerDeref*, *pc*)
    **else** *loc*(*pc*) := *FldLoc*(*ClassLoc*)
      *Coerce*(*val*(*pc*), *mem*(*FldLoc*(*ClassLoc*)), *pc*)
      *Proceed*
  **else** *loc*(*pc*) := *FldLoc*(*StrctLoc*)
    *Coerce*(*val*(*pc*), *mem*(*FldLoc*(*StrctLoc*)), *pc*)
    *Proceed*

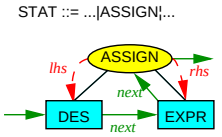
$StrctLoc \triangleq loc(strct(prog, pc))$
$ClassLoc \triangleq mem(StrctLoc)$
$FldLoc(l) \triangleq floc(l, field(prog, pc))$



## Zuweisung

Wird an ein Verbundobjekt zugewiesen, dann werden die Werte der Verbundfelder in das Teilobjekt, auf das entsprechende Verbundfeld verweist, geschrieben. Wird an ein Verbundzeigerobjekt zugewiesen wird der Zeiger in das Objekt, auf das die linke Seite verweist, geschrieben.

⇒ Nur für Verbundtypen A gilt $isAtomic(A) \doteq false$




**if** *CT* **is** ASSIGN ∧ $isAtomic(Pri(lhs)) \doteq false$ **then**
  *StructCopy*(*lhs*, *rhs*)
  *Proceed*

$StructCopy(dest, src) \triangleq$ **forall** $x$ : ID • $isField(Pri(dest), x)$ **do**
  **if** $isAtomic(gettype(fields(Pri(dest), x))$ **then**
    $mem(FldLoc(loc(dest))) := mem(FldLoc(loc(src)))$
  **else** $StructCopy(FldLoc(loc(dest)), FldLoc(loc(src)))$

**Bemerkung**

*StructCopy* ist definiert, da jedes Verbundobjekt endlich ist.

## forall und choose

**Definition 2.16 (forall-Aktualisierungen)**

Sei $\Sigma$ eine Signatur und $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R \rangle$ eine ASM, wobei in einer Regel **if** $\varphi$ **then** $U$ auch Aktualisierungen der Form $FORALL \triangleq$ **forall** $x : \mathrm{A} \bullet \psi$ **do** $U'$ und $CHOOSE \triangleq$ **choose** $x : \mathrm{A} \bullet \chi$ **do** $U''$ vorkommen dürfen.

i. Durch **forall** wird eine Aktualisierungsmenge $Update_{\mathfrak{A}}(FORALL) \triangleq \{u[t/x] : t \in \mathcal{T}_{\mathrm{A}}(\Sigma) \wedge u \in U' \wedge \mathfrak{A} \models \psi[t/x]\}$. Die **forall**-Aktualisierungsmenge ist **beschränkt** gdw. $|\{\langle [\![t_1]\!]_{\mathfrak{A}}, [\![t_1]\!]_{\mathfrak{A}} \rangle : t_1 := t_2 \in Update_{\mathfrak{A}}(FORALL)\}| < \infty$ ist.

ii. Jede Aktualisierungsmenge $\{u[x/t] : u \in U''\}$ für ein $t \in \mathcal{T}_{\mathrm{A}}(\Sigma)$ mit $\mathfrak{A} \models \chi[t/x]$ ist eine *CHOOSE*-**Aktualisierungsmenge**

iii. Sei $U = U_1 \uplus U_2 \uplus U_3$ wobei $U_1$ eine Menge von Aktualisierungen der Form $t_1 := t_2$, $U_2$ eine Menge von **forall**-Aktualisierungen und $U_3$ eine Menge von **choose**-Aktualisierungen ist. Weiter sei $\mathfrak{A} \in \mathbb{A}$ ein Zustand. Eine Menge $\bar{U} \triangleq U_1 \cup \bigcup_{f \in U_2} Update_{\mathfrak{A}}(f) \cup \bigcup_{c \in U_3} \{u : u \text{ ist } c\text{-Aktualisierung}\}$ heißt durch $\mathfrak{A}$ und $U$ definierte Aktualisierungsmenge.

iv. Sei $\mathbb{U} \triangleq \{\bar{U} : \exists \textbf{if } \varphi \textbf{ then } U \in R \bullet \mathfrak{A} \models \varphi \wedge \bar{U} \text{ ist durch } \mathfrak{A} \text{ und } U \text{ definierte Aktualisierungsmenge}\}$ eine Menge von Aktualisierungsmengen. Eine Menge $Update_{\mathcal{A}}(\mathfrak{A}) = \bigcup_{\bar{U} \in \mathbb{U}} \bar{U}$ heißt **Aktualisierungsmenge** im Zustand $\mathfrak{A}$. Ein Zustand $\mathfrak{A}'$ heißt **Nachfolgezustand** von $\mathfrak{A}$ gdw. $\mathfrak{A}' = next_U(\mathfrak{A})$ für eine Aktualisierungsmenge $U$ im Zustand $\mathfrak{A}$.

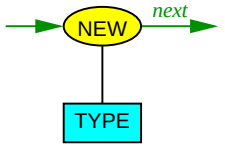


## Erzeugen von Objekten

Eine new-Operation erzeugt ein Objekt der Klasse und liefert einen Verweis auf dieses Objekt.

⇒ Dieser Verweis darf nicht benutzt worden sein

EXPR ::=...|NEW|...




**if** *CT* **is** NEW **then**
  **choose** *l* : LOC • *IsUsed*(*l*) ≐ *false* **do**
    *loc*(*pc*) := *l*
  *Proceed*

**Benutzte Verweise**

- Verweise auf Verbundfelder werden immer benutzt

⇒ $isUsed(e, floc(l, x)) \doteq false$



## 2.4.5 Prozeduren und Funktionen

**Beobachtung**

- Prozeduraufruf sichert Zustand (Bindungen, Ausdruckswerte) und legt neue Objekte an
- Dabei werden Parameter übergeben
- Nach Rückkehr aus der Prozedur wird der alte Zustand wieder hergestellt
- Dabei werden Funktionsresultate übergeben

⇒ Klassisches Verhalten eines Kellers (engl. *stack*)

**Laufzeitkeller**

**spec** $\mathrm{ENV}_2$ **extends** $\mathrm{ENV}_1$
  **sorts** FRAME, FRAMESTACK, EXPRVALS
  **operations**

| | | |
|---|---|---|
| *mkframe* : | ENV × OCC × EXPRVALS | → FRAME |
| *mkstack* : | | → FRAMESTACK |
| *push* : | FRAMESTACK × FRAME | → FRAMESTACK |
| *pop* : | FRAMESTACK | →?FRAMESTACK |
| *getenv* : | FRAMESTACK | →?ENV |
| *getpc* : | FRAMESTACK | →?OCC |
| *getval* : | FRAMESTACK | →?EXPRVALS |
| *novals* : | | → EXPRVALS |
| *addval* : | OCC × VALUE × EXPRVALS | → EXPRVALS |
| *findval* : | OCC × EXPRVALS | →?VALUE |

- *mkframe* verwaltet den Zustand
- Mit *getenv*, *getpc* und *getval* ist die Wiederherstellung des Zustands möglich



## Erweiterung des Zustandsraums und des Anfangszustands

**Erweiterung des Zustandsraums**

Für die Verwaltung der Prozeduraufrufe muss noch zusätzlich gegenbüber C0 ein Prozedurkeller eingeführt werden.

**spec** $\textsc{State}_2$ **extends** $\textsc{State}_1, \textsc{Env}_2$
**operations** *procstack* :→ FRAMESTACK

**Anfangszustand**

Zusätzlich zu den Bedingungen in C0 muss noch der Prozedurkeller leer sein:

**spec** $\textsc{InitState}_1$ **extends** $\textsc{State}_1, \textsc{InitState}$
**axioms** $procstack \doteq mkstack$

## Anlegen der Parameter und Parameterübergabe

**Vorgehen**

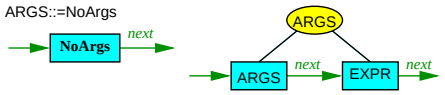
- Anlegen der Objekte für die Parameter

⇒ Parameter können übergeben werden

- Vorbereiten für andere Übergabemodi

⇒ Statische Funktion *mode* : OCC →?MODE

**Auswerten der Argumente**




**if** *CT* **is** NOPARS **then** *Proceed*

**Parameterübergabe**

$PASS(ids) \triangleq$ **choose** $e$ : ENV • $Good(e, ids)$ **do**
$env := e$
**forall** $i$ : INT • $0 \leq i < length(ids)$ **do**
**if** $isAtomic(Post(Arg(i))) \doteq true$ **then**
$mem(bind(e, x)) := val(Arg(i))$
**else** $StructCopy(bind(e, x), Arg(i))$

$Arg(i) \triangleq arg(prog, pc, i)$



## Zustandssicherung

**Was ist zu tun?**

- Sichern des Befehlszeigers
- Sichern der Bindungen lokaler Variable an Objekte
- Bei Funktionen müssen noch Zwischenergebnisse des Ausdrucks der gerade berechnet wird, gesichert werden.
  - Statische Funktion *exproccs* : PROG × *OCC* → *OCCLIST*; $exproccs(p, o)$ ist die eine Liste der Navilagitionslisten, die zum selben Ausdruck wie $o$ gehören. Die Liste ist leer, falls $\neg occ(p, o)$ **is** EXPR.
  - Sorte SAVELIST mit Funktionen

| | | |
|---|---|---|
| *mksavelist* : | | → SAVELIST |
| *save* : | OCC × VALUE × SAVELIST | → SAVELIST |
| *findval* : | OCC × SAVELIST | → VALUE |

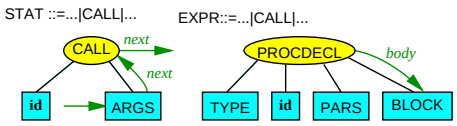
  - Abgeleitete dynamische Funktion *save* : OCCLIST →?SAVELIST:

| | |
|---|---|
| $save(nil)$ | $\doteq mksavelist$ |
| $save(cons(o, occs)$ | $\doteq save(o, val(o), save(occs)$ |



## Prozeduraufruf

- Mit dem Prozedur–/Funktionsaufruf werden den Parametern Objekte zugeordnet.
- Bei einem Prozedur– oder Funktionsaufruf werden die Argumente ausgewertet und an die Parameter positional übergehen, d.h. der Wert eines Arguments wird in das Objekt geschrieben, auf das der Parameter verweist. Danach wird der Prozedurrumpf ausgeführt.




**if** *CT* **is** CALL **then**
$Pass(pars(CurProc))$
$procstack := push(procsstack, Frame)$
$pc := body(CurProc)$

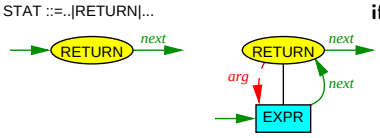
$Frame \triangleq mkframe(env, pc, save(exproccs(pc)))$
$CurProc \triangleq identifyDef(deftab(prog, pc), occ(prog, pc.0))$

- *deftab* ist eine Definitionstabelle, *identifyDef* identifiziert eine Deklaration
- Eine Prozedurdefinition speichert u.A. die Parameter(namen) und den Rumpf



## Prozedurrückkehr

- Ein Prozedur– oder Funktionsaufruf **kehrt zurück**, wenn die letzte Anweisung des Prozedurrumpfs oder eine `return`-Anweisung ausgeführt wurde.
- Nach Rückkehr von einem Prozeduraufruf wird die Anweisung nach dem Aufruf ausgeführt.
- Nach Rückkehr von einem Funktionsaufruf wird mit der Ausführung der Anweisung, in der der Aufruf vorkam, fortgefahren. Der Wert eines Funktionsaufrufs ist der Wert des Ausdrucks der ausgeführten `return`-Anweisung.




**if** *CT* **is** RETURN ∨ $\neg D(pc)$ **then**
$pc := next(prog, OldPc)$
$env := OldEnv$
$procstack := pop(procstack)$
**forall** $o$ : OCC • $isElem(o, exproccs(prog, OldPc))$ **do**
$val(o) := findval(o, OldVals)$
**if** $isFunction(CurProc)$ **then**
**if** $isAtomic(rettype(CurProc)) \doteq true$ **then**
$Coerce(val(pc), val(Arg), OldPc)$
**else choose** $l$ : LOC • $lsUsed(l) \doteq false$ **do**
$StructCopy(l, Arg)$
$loc(OldPc) := l$

$OldPc \triangleq getpc(procstack)$
$OldEnv \triangleq getenv(procstack)$
$OldVals \triangleq getval(procstack)$
$Arg \triangleq arg(prog, pc)$

Statische Funktion *rettype* : PROCDEF → TYPE ergibt Rückgabetyp der Funktion

## Diskussion

- Der Zustandsraum ist nun:
  **spec** STATE **extends** ENV$_1$
  **sorts** EXCEPTION
  **operations**

| | | |
|---|---|---|
| *env* : | | → ENV |
| *mem* : | LOC | →?VALUE |
| *inp* : | | → LISTOFVAL |
| *out* : | | → LISTOFVAL |
| *prog* : | | → PROG |
| *pc* : | | → OCC |
| *exc* : | | → EXCEPTION |
| *val* : | OCC | →?VALUE |
| *loc* : | OCC | →?LOC |
| *entered* : | OCC | →?BOOL |
| *procstack* : | | →?FRAMESTACK |

- Makro *IsUsed*, das angibt, ob ein Verweis *l* schon benutzt wurde, muss noch definiert werden.
- Auch Verweise im Prozedurkeller werden benutzt: Funktion
  *isUsed* : FRAMESTACK × LOC → BOOL

⇒ $IsUsed(l) \triangleq isUsed(env, l) \doteq true \vee isUsed(procstack, l) \doteq true$



## Übungen

- Erweitern Sie die Semantik um Ausnahmebehandlung:
  *exception* ::=`try` *block*
  (`when` *excp stat*)$^+$
  [`otherwise` *stat*]
  Wird in *block* eine Ausnahme ausgelöst, die in einem der `when`-Fälle aufgeführt ist, dann wird die entsprechend Anweisung ausgeführt. Danach wird die Anweisung nach dem Block ausgeführt. Mit `otherwise` werden alle nicht-aufgeführt Ausnahmen behandelt. Ein Prozedur– oder Funktionsaufruf endet mit einer Ausnahme, wenn sie im Rumpf nicht behandelt wird.
- Einführung von Benutzerdefinierten Ausnahmen:
  *exceptdecl* ::= `exception` **id**
  *raise* ::= `raise` **id**
  - Die Ausnahmen müssen global deklariert werden. Die Bezeichner in allen Deklarationen müssen paarweise verschieden sein.
  - Mit der `raise`-Anweisung wird explizit eine Ausnahme ausgelöst. Die ausgelöste Ausnahme muss deklariert sein.
- Einführung von Referenzaufruf (Modus `ref`) und Wert-/Ergebnisaufruf (Modus `inout`)



## Übung: Einführen von Reihungen

*arrtype* ::= *type* `[ ]`
*arracc* ::= *des* `[` *expr* `[]`
*arrcreate* ::= `new` *type* `[` *expr* `[]`

- Reihungen ordnen Indices Werten zu
- Reihungszugriffe sind Zugriffspfade
- Der Typ eines Reihungszugriffs ist der Elementtyp der Reihung
- Der Typ des Indexausdrucks muss `int` sein.
- Der Typ einer Reihungserzeugung ist der Elementtyp.
- Der Typ des Ausdrucks einer Reihungserzeugung muss `int` sein.
- Ein Reihungszugriff werden den Indexausdruck zu einer Zahl $n$ aus. Ist $n < 0$ oder $n$ größer gleich des Reihungsumfangs, so wird die Ausnahme `IndexOutOfBoundException` ausgelöst. Ansonsten bezeichnet der Reihungszugriff das $n + 1$-te Element der Reihung bzw. dessen Wert
- Eine Reihungserzeugung wertet den Ausdruck zu einer Zahl $n$ aus. Ist $n \leq 0$ so wird die Ausnahme `IndexOutOfBoundException` ausgelöst. Ansonsten wird eine Reihung mit $n$ Elementen erzeugt. Das Ergebnis ist ein Verweis auf die erzeugte Reihung



# Kapitel 3
# Korrektheit der Speicherabbildung

**Wolf Zimmermann**

**Verifikation von Übersetzern**

**Ziele**

- Kennenlernen der allgemeinen Vorgehensweise bei Transformation des Zustandsraums
- Korrektheitsnachweis der Speicherabbildung
- Bewusstsein für das Zusammenwirken mit der statischen Semantik

1. Transformation des Zustandsraums
2. Verifikation der Speicherabbildung



# Kapitel 4
# Korrektheit der Zwischencodeerzeugung

**Wolf Zimmermann**

**Verifikation von Übersetzern**





**Ziele**

- Funktionsweise von Simulationsbeweisen
- Verifikation von Transformationsregeln
- Konzept der Übersetzungsvalidierung

1. Simulationsbeweise
2. Verifikation der Zwischencodeerzeugung
3. Übersetzungsvalidierung
4. Ausblick: Optimierungen



# Kapitel 5
# Korrektheit der Codeerzeugung und der Assemblierung

**Wolf Zimmermann**

**Verifikation von Übersetzern**

**Ziele**

- Kennenlernen der termersetzungsbasierten Codeerzeugung
- Verifikation der Codeerzeugung
- Bewusstsein für die Problematik des Zusammenwirkens von Registerzuteilung und Codeerzeugung
- Verifikation der Assemblierung

1. Codeerzeugung durch Termersetzung
2. Verifikation von termersetzungs-basierter Codeerzeugung
3. Überprüfung der Korrektheit der Codeerzeugung
4. Verifikation der Assemblierung