Verifikation von Übersetzern

Wolf Zimmermann

Verifikation von Übersetzern

Lernziele

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

Kapitel 1
Einleitung

Wolf Zimmermann

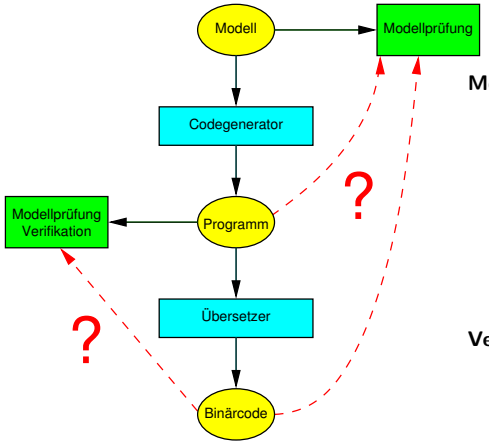
Verifikation von Übersetzern

Inhalt

Ziele

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

1.1 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=36;
(2) int j=24;
(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

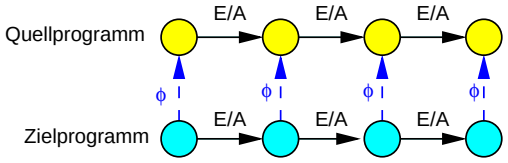
☞ Meist bei Ausdrucksauswertung

- 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

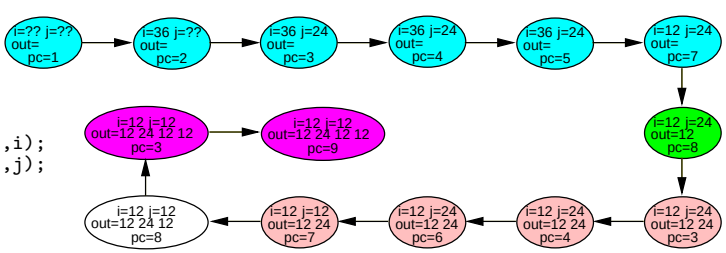
Formale Definition der Korrektheit: Zustandsübergangssysteme

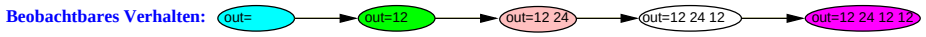
Definition 1.1 (Zustandsübergangssystem und Lauf)

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=36;
(2) int j=24;
(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: 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: Erhaltung beobachtbaren Verhaltens

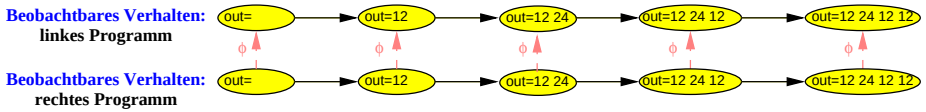
Definition 1.3 (1-1-Simulation)

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 $\phi : Q_2 \rightarrow Q_1$ 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=36;
(2) int j=24;
(3) while (i!=j) {
(4)    if (i>j) i=i-j;
(5)    else j=j-i;
(6)    printf("%d ",i);
(7)    printf("%d ",j);
(8) }
```

```
(1) printf("24 ");
(2) printf("12 ");
(3) printf("12 ");
(4) printf("12 ");
```



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: 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 $\phi : Q_2 \rightarrowtail Q_1$ 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 $\mathcal{S}_2$ ist endlich, q_n ist nicht im Definitionsbereich von ϕ, $q_0, \ldots, q_{n-1}$ ist im Definitionsbereich von ϕ und $\langle \phi(q_0), \ldots, \phi(q_{n-1}) \rangle$ ist echter Präfix eines vollständigen Laufs von $\mathcal{S}_1$.
Notation: $\mathcal{S}_2 \precapprox \mathcal{S}_1$

Formale Definition der Korrektheit: Zusammenfassung

Beobachtung

Erhaltung beobachtbaren Verhaltens bedeutet eingeschränkte 1-1-Simulation

- Semantik von Programmiersprachen ordnen jedem Programm π 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 π_Z der Programmiersprache Z heißt **korrekte Übersetzung** eines Programms π_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 π 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

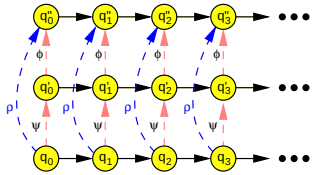
Satz 1.2 (Horizontale Dekomposition)

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.

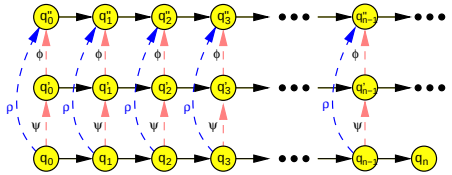
Beweisskizze

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 $q_n \notin \mathrm{DOM}(\rho)$



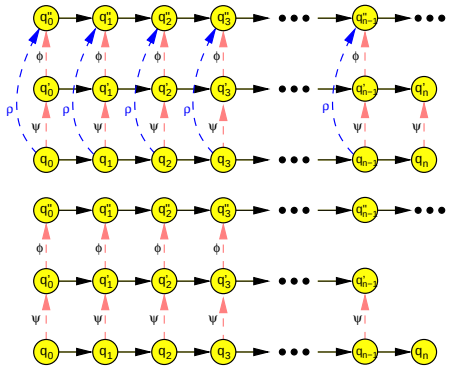
Horizontale Dekomposition (Forts.)

Beweisskizze

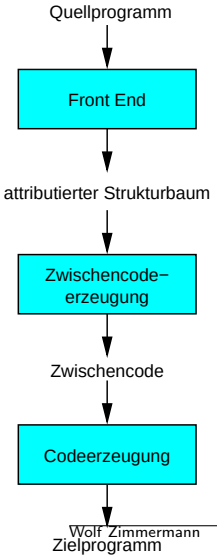
3. Fall: Lauf für $\mathcal{S}_3$ endlich, $q_0, \ldots, q_n \in \mathrm{DOM}(\psi)$ und $\psi(q_0), \ldots, \psi(q_{n-1}) \in \mathrm{DOM}(\phi)$ und $\psi(q_n) \notin \mathrm{DOM}(\phi)$. Also: $q_0, \ldots, q_{n-1} \in \mathrm{DOM}(\rho)$ und $q_n \notin \mathrm{DOM}(\rho)$

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 $\mathcal{S}_i = (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$.

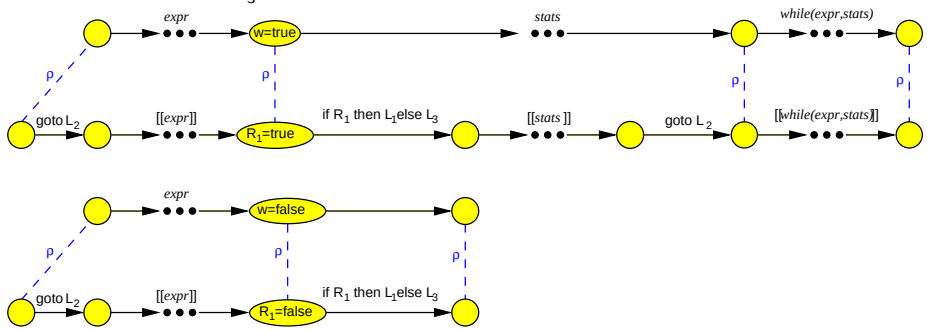
- Relation $\rho \subseteq Q_2 \times Q_1$ an den Grenzen der Transformationsregeln
- Beobachtbares Verhalten muss erhalten bleiben
- Einzelne Simulationen müssen zusammengesetzt werden

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

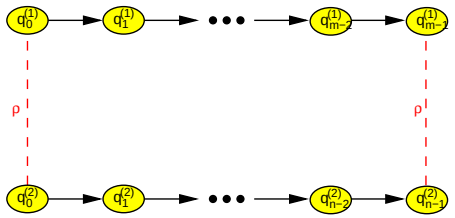
Definition 1.5 (Urbild– und Bildbereich von Relationen)

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** ρ. Die Menge $\mathrm{RAN}(\rho) \triangleq \{u \in U_2 : \exists u' \in U_1 \bullet (u', u) \in \rho\}$ heißt **Bildbereich von** ρ.

Was muss gelten?

Gegeben seien zwei Zustandsübergangssysteme $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$ mit jeweils Abstraktionen α_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 ϕ.
Vorgehen: Definition einer Relation $\rho \subseteq Q_2 \times Q_1$ und Zusammensetzen von "Schnappschüssen":

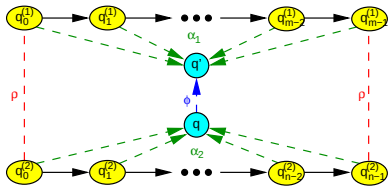
- Lauf $\langle q_0^{(2)}, \ldots, q_{n-1}^{(2)} \rangle$ von $\mathcal{S}_2$ mit höchstens einem beobachtbaren Zustandsübergang wird auf Lauf $\langle q_0^{(1)}, \ldots, q_{m-1}^{(1)} \rangle$ mit höchstens einem beobachtbaren Zustandsübergang abgebildet
- $q_0^{(2)}, q_{n-1}^{(2)} \in \mathrm{DOM}(\rho)$ 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 $j = 0, \ldots, m-1$

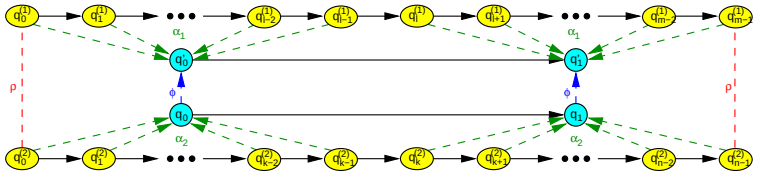


Vertikale Dekomposition (Forts.)

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

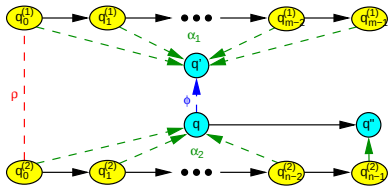
$\alpha_2(q_i^{(2)}) = \alpha_2(q_0^{(2)})$ für $i = 0, \ldots, k-1$, $\alpha_2(q_i^{(2)}) = \alpha_2(q_k^{(2)})$ für $i = k+1, \ldots, n-1$,
$\alpha_1(q_j^{(1)}) = \phi(\alpha_2(q_0^{(2)}))$ für $j = 0, \ldots, l-1$ und $\alpha_1(q_j^{(1)}) = \phi(\alpha_2(q_k^{(2)}))$ für $j = k, \ldots, m-1$.



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 α_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 α_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 $i > 0$, $q_j^{(1)} \notin \mathrm{RAN}(\rho)$ für $j > 0$ oder $(q_i^{(2)}, q_j^{(1)}) \in \rho$ 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)

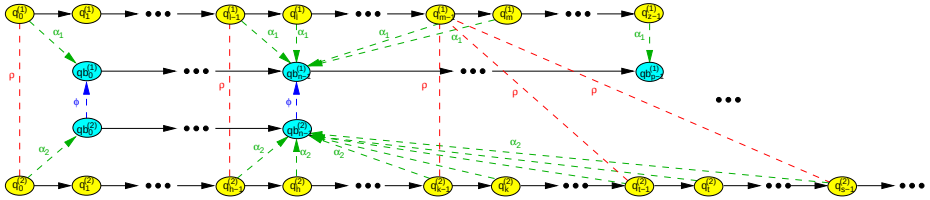
Vorgehen:
- Zeige durch Induktion über die Länge der Läufe von $\mathcal{S}_2$, dass mit ϕ die Bedingungen für $\mathcal{S}_2' \precsim \mathcal{S}_1'$ für jeden Präfix $\langle qb_0^{(2)}, \ldots, qb_{n-1}^{(2)} \rangle$ eines vollständigen Laufs von $\mathcal{S}_2'$ erfüllt sind
- Zeige durch noethersche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

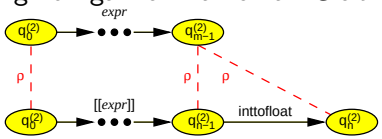
Die Bedingung (ii) schließt die folgende Situation aus:



0-m- und n-0-Simulationen

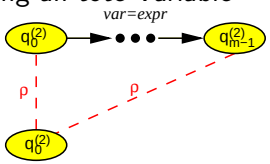
Beispiel 1.6: 0-m-**Simulation**

Implizite Anpassung von ganzen Zahlen an Gleitkommazahlen



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

Streichen von Zuweisung an tote Variable



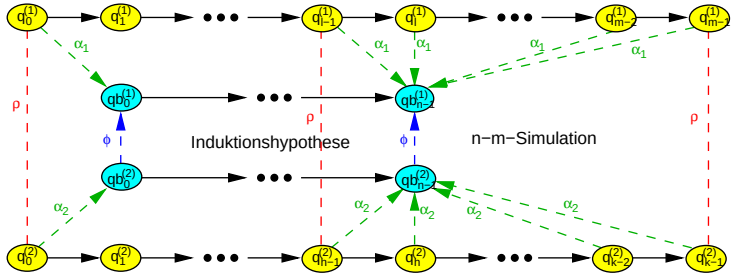
Lemma 1.4 (Endliche Präfixe)

Seien $\mathcal{S}_i \triangleq (Q_i, I_i, \rightarrow_i)$, $i = 1, 2$, zwei Zustandsübergangssysteme mit jeweils Abstraktionen α_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 $(q_0^{(2)}, q_0^{(1)}), (q_{k-1}^{(2)}, q_{m-1}^{(1)}) \in \rho$.

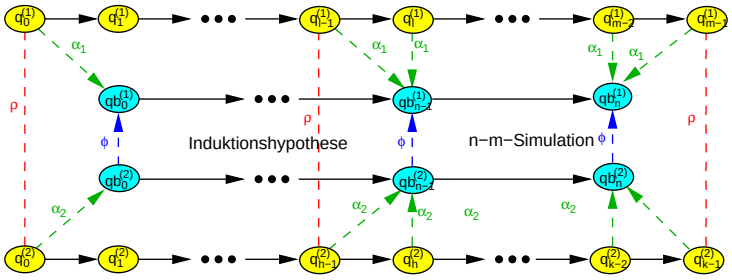
Beweis durch Induktion über die Länge k

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 α_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 $i > 0$, $q_j^{(1)} \notin \mathrm{RAN}(\rho)$ für $j > 0$ oder $(q_i^{(2)}, q_j^{(1)}) \in \rho$ 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 α_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

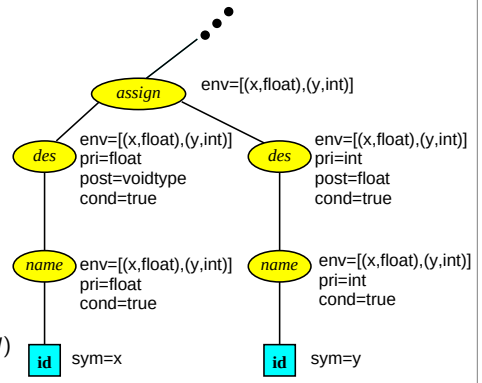
- Spezifikation z.B. durch attributierte Grammatik (nicht die, die im Übersetzer verwendet wird!)
- Überprüfung, ob Attribute gemäß Attributierungsregeln berechnet wurden

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

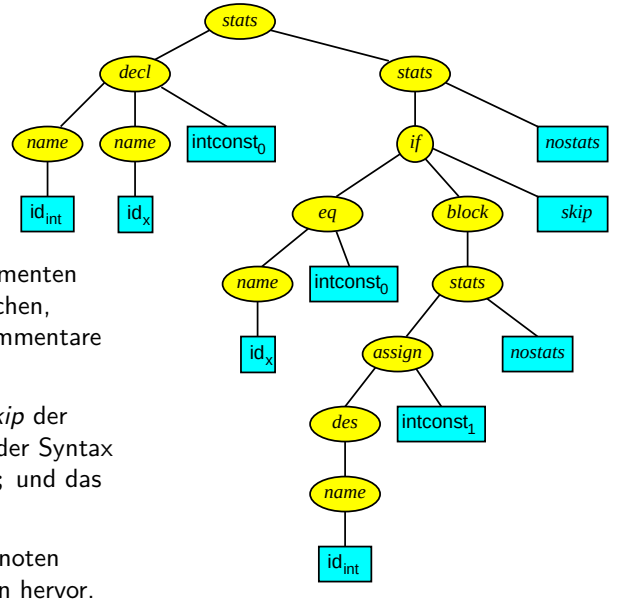
Relation zwischen Quelltexten und abstrakten Syntaxbäumen

☞ 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.