2. Programmiersprache

Ziel: Verifikation von Programmen mit fine-grained Concurrency, zeige also, dass das Programm keine Fehler enthält.

Benötigt:
- Argumentation über alle Ausführungen des Programms.
- Erfordert Formalismus zur Beschreibung des Programmverhaltens — der Semantik.

2.1 Syntax von While-Programmen

Ansatz:
- Um die Grundlagen der Semantik / Verifikation zu erlernen, verzichten wir vorübergehend auf Pointer.
- Studiere die Semantik von while-Programmen über integer-Variablen.

Definition (Syntax):

Programme unserer Programmiersprache (W--) (-- = keine Pointer) sind durch folgende Backus-Naur-Form gegeben:

$$a ::= k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$$

// Arithmetische Ausdrücke über ganzen Zahlen

$$b ::= 0 \mid 1 \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2$$

// Boolesche Ausdrücke

$$c ::= \underline{skip} \mid x := a \mid \underline{assume}\ b \mid c_1 ; c_2$$
$$\mid \underline{if}\ b\ \underline{then}\ c_1\ \underline{else}\ c_2\ \underline{fi} \mid \underline{while}\ b\ \underline{do}\ c\ \underline{od},$$

// Befehle (commands)

1- mit $k \in \mathbb{Z}$, $0, 1 \in \mathbb{B}$ und $x \in Var$.

2.2 Strukturierte operationelle Semantik

(SOS, Gordon Plotkin 1981)

Ziel: • Definiere operationelle Semantik von W-- Programmen.

Idee: • Konfigurationen (Laufzeitzustände) eines Programms haben eine syntaktische Struktur: Komposition atomarer Elemente mittels Operatoren.

• Damit lassen sich Beweissysteme (Kalküle) nutzen, um das Verhalten von Konfigurationen zu definieren:

Eine Transition existiert gdw. sie in Beweissystem herleitbar ist.

• Technisch nutzt das Beweissystem eine Induktion nach der Struktur von Konfigurationen.

↳ Axiome definieren die Transitionen von atomaren Elementen (die Basis-Commands ausführen).

↳ Beweisregeln definieren die Transitionen zusammengesetzter Konfigurationen über die Transitionen der Operanden.

• Vorteile:

↳ Einfachheit und Eleganz

↳ Möglichkeit, Eigenschaften von Transitionen über Induktion entlang der Ableitung herzuleiten.

Fakt: Zwei Stile struktureller Semantik:

Small-Step-Semantik: Definiert den Effekt jeder Operation

2. Big-Step-Semantik: Fasst den Effekt von Programmausführungen zusammen.

Formel ist eine Semantik eine Abbildung,
die jedem Programm in C--
eine Bedeutung zuordnet.
Dabei ist die Bedeutung
ein Element eines semantischen Bereichs.
Bevor wir Programmen eine Bedeutung zuordnen können,
müssen wir das für arithmetische und Boolesche Ausdrücke erledigen.
Das kennen wir aus der Logik.

Definition (Signatur, Struktur):

- Eine Signatur ist ein Paar $Sig = (Funk, Päd)$
  mit
  - $Funk$ einer Menge von Funktionssymbolen f_n (mit Stelligkeit)
  - $Päd$ einer Menge von Prädikatssymbolen p_n.
- Eine Struktur der Signatur Sig ist ein Paar $S = (D, I)$
  mit
  - $D \neq \emptyset$ einer Universmenge oder Datenbereich und
  - I einer Interpretation, die jedem
    Funktionssymbol $f_n \in Funk$ eine tatsächliche Funktion
    $$\tilde{I}(f) : D^n \to D$$
    und jedem Prädikatssymbol $p_n \in Päd$ ein tatsächliches Prädikat
    $$\tilde{I}(p) : D^n \to \mathbb{B}$$
    zuordnet.

Bemerkung (Signatur und Struktur von While):

- Für While ist $Sig = (Funk, Präd)$

  mit

  $$Funk := \{+/_2, -/_2, */_2\} \cup \{h/_0 \mid h \in \mathbb{Z}\}$$

  $$Präd := \{>/_2\}.$$

- Die Sig-Struktur von While-Programmen ist

  $$\mathcal{S} = (\mathbb{Z}, \mathcal{I})$$

  mit $\mathcal{I}(+), \mathcal{I}(-), \mathcal{I}(*), \mathcal{I}(>)$

  der erwarteten Interpretation über den ganzen Zahlen.

- Beachte:

  ↳ Wir haben $=$ nicht in Präd aufgenommen.

  Stattdessen nehmen wir Prädikatenlogik mit Gleichheit,

  in der $=$ immer als Identität auf der Wertemenge definiert ist.

  Das Symbol kann also gar nicht interpretiert werden.

  ↳ Diese Konvention (Gleichheit nicht zu interpretieren)

  findet man übrigens auch in Programmiersprachen.

  Java: $==$ ist immer die Identität auf Objekten.

  C++: Hier gibt es dies nicht, $==$ kann überladen werden.

  ↳ Auch die Junktoren $\neg, \wedge, \vee$ werden nicht interpretiert.

  Sie verknüpfen also auch Boolesche und nicht arithmetische Ausdrücke,

  und das auf die erwartete Weise.

Das Verhalten eines Programms hängt von der Belegung der Variablen

auch Zustand genannt, als: $\sigma \in Stat = \mathbb{Z}^{Var} = Var \to \mathbb{Z}$.

Damit erhält man die Semantik von arithmetischen Ausdrücken in der Sig-Struktur S genau, wie in der Logik definiert:

$$S[\![-]\!] : AExp \to (2^{Vars} \to \mathbb{Z}), \text{ wobei } S[\![a]\!] : 2^{Vars} \to \mathbb{Z}$$

mit

$$S[\![x]\!]\sigma := \sigma(x)$$

$$S[\![f(a_1,\dots,a_n)]\!]\sigma := \mathcal{I}(f)(S[\![a_1]\!]\sigma, \dots, S[\![a_n]\!]\sigma) \text{ mit } f \in Funk.$$

Die Semantik von Booleschen Ausdrücken in S ist

$$S[\![b]\!] : 2^{Vars} \to \mathbb{B}$$

mit

$$S[\![a_1 = a_2]\!]\sigma := 1, \text{ falls } S[\![a_1]\!]\sigma = S[\![a_2]\!]\sigma$$

$$S[\![p(a_1,\dots,a_n)]\!]\sigma := \mathcal{I}(p)(S[\![a_1]\!]\sigma, \dots, S[\![a_n]\!]\sigma) \text{ mit } p \in Präd$$

$$S[\![\neg b]\!]\sigma := 1 - S[\![b]\!]\sigma$$

$$S[\![b_1 \wedge b_2]\!]\sigma := \min\{S[\![b_1]\!]\sigma, S[\![b_2]\!]\sigma\}$$

$$S[\![b_1 \vee b_2]\!]\sigma := \max\{S[\![b_1]\!]\sigma, S[\![b_2]\!]\sigma\}.$$

2.2.1 Small-Step (operationelle) Semantik

Ziel: Definiere das Verhalten eines Programms als die Ausführung eines Befehls nach dem anderen.

Ansatz:
- Definiere Beweisregeln, die auf Konfigurationen angewendet werden, bis ein finaler Zustand abgeleitet ist.
- Wir schreiben Beweise als Transitionen zwischen Konfigurationen: $cf \to cf'$ bedeutet: cf' kann aus cf in einem Schritt mit einem Beweis abgeleitet werden.

- $cf \to^n cf'$ bedeutet, cf' kann aus cf in genau n-Schritten abgeleitet werden (für $n=0$, $cf' = cf$).
- $cf \to^* cf'$ bedeutet, es gibt ein $n \in \mathbb{N}$, so dass $cf \to^n cf'$.

Definition:

Eine Konfiguration ist ein Paar $(c, \sigma) \in \omega\text{--} \times State$, wobei

- c das noch auszuführende Programm und
- σ der Zustand des Programms ist.

Definition:

Die Small-Step-Transitionsrelation zwischen Konfigurationen

$$\to \in (\omega\text{--} \times State) \times (\underbrace{(\omega\text{--} \times State)}_{\text{nicht final}} \cup \underbrace{State}_{\text{final}})$$

ist die kleinste Relation, die folgenden Regeln genügt:

$$\text{(SKIP)}\ \frac{}{(skip, \sigma) \to \sigma} \qquad \text{(ASSIGN)}\ \frac{}{(x := a, \sigma) \to \sigma[x \mapsto \mathcal{S}[\![a]\!]\sigma]}$$

$$\text{(SEQ1)}\ \frac{(c_1, \sigma) \to \sigma'}{(c_1; c_2, \sigma) \to (c_2, \sigma')} \qquad \text{(SEQ2)}\ \frac{(c_1, \sigma) \to (c_1', \sigma')}{(c_1; c_2, \sigma) \to (c_1'; c_2, \sigma')}$$

$$\text{(IF\_TRUE)}\ \frac{}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi}, \sigma) \to (c_1, \sigma)},\ \text{falls } \mathcal{S}[\![b]\!]\sigma = 1$$

$$\text{(IF\_FALSE)}\ \frac{}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi}, \sigma) \to (c_2, \sigma)},\ \text{falls } \mathcal{S}[\![b]\!]\sigma = 0$$

$$\text{(WHILE\_TRUE)}\ \frac{}{(\textbf{while } b \textbf{ do } c \textbf{ od}, \sigma) \to (c; \textbf{while } b \textbf{ do } c \textbf{ od}, \sigma)},\ \text{falls } \mathcal{S}[\![b]\!]\sigma = 1$$

(WHILE-FALSE) $\dfrac{}{(\underline{\text{while}}\ b\ \underline{\text{do}}\ c\ \underline{\text{od}}, \sigma) \to \sigma}$, falls $S[\![b]\!]\sigma = 0$

(ASSUME) $\dfrac{}{(\text{assume}(b), \sigma) \to \sigma}$, falls $S[\![b]\!]\sigma = 1$

Bemerkung:

- Die Definition der Transitionsrelation benutzt Regelschemata der Form

$$\frac{\text{Prämisse}}{\text{Konklusion}}$$

mit folgender Bedeutung:

Falls die Prämisse instantiiert werden kann, kann man die entsprechende Instantiierung der Konklusion schließen.

- Regeln ohne Prämissen heißen Axiome.
- Wiederholte Anwendungen der Regeln erzeugen einen Ableitungsbaum:

↳ Axiome = Blätter

↳ Transition für die Konfiguration von Interesse = Wurzel.

Bemerkung:

- Ein assume-Statement hat nur eine Regel, für den Fall, dass die Bedingung gilt. Falls die Bedingung nicht gilt blockiert die Ausführung.
- Es gibt eine alternative Definition der Transitionsrelation als

$$\to \subseteq (W \times \text{Stat}) \times (W \times \text{Stat}),$$

z.B. die mit sieben Regeln auskommt. Wie? Hinweis: while usw. ij.

Beispiel:

Betrachte folgendes Programm

$$c \equiv x := x+1;\ loop$$

mit $loop \equiv$ while $x > 0 \wedge y < 1$ do $y := y+1$ od

Wir betrachten Zustände σ des Programms als

$$\sigma = (i, j) \quad \text{mit der Bedeutung} \quad \begin{matrix} \sigma(x) = i \\ \sigma(y) = j. \end{matrix}$$

Dann liefert das Beweissystem folgende Ableitungen:

(1)
$$\frac{\dfrac{}{(x := x+1, (0,0)) \to (1,0)}\ (ASSIGN)}{(x := x+1;\ loop, (0,0)) \to (loop, (1,0))}\ (SEQ1)$$

(2)
$$\frac{}{(loop, (1,0)) \to (y := y+1;\ loop, (1,0))}\ (WHILE\_TRUE)$$

denn

$$\begin{aligned} &\mathcal{S}[\![x > 0 \wedge y < 1]\!] (1,0) \\ &= \min \{ \mathcal{S}[\![x > 0]\!] (1,0),\ \mathcal{S}[\![y < 1]\!] (1,0) \} \\ &= \min \{ 1 >_{\mathbb{Z}} 0_{\mathbb{Z}},\ 0 <_{\mathbb{Z}} 1_{\mathbb{Z}} \} \\ &\qquad \text{II(>)} \\ &= \min \{ 1, 1 \} \\ &= 1. \end{aligned}$$

(3)
$$\frac{\dfrac{}{(y := y+1, (1,0)) \to (1,1)}\ (ASSIGN)}{(y := y+1;\ loop, (1,0)) \to (loop, (1,1))}\ (SEQ1)$$

(4)
$$\frac{}{(loop, (1,1)) \to (1,1)}\ (WHILE\_FALSE)$$

- Die Semantik von While

ist nun eine Abbildung

$$[\![-]\!] : While \longrightarrow (2^{Vars} \longrightarrow \text{Transitionssysteme}),$$

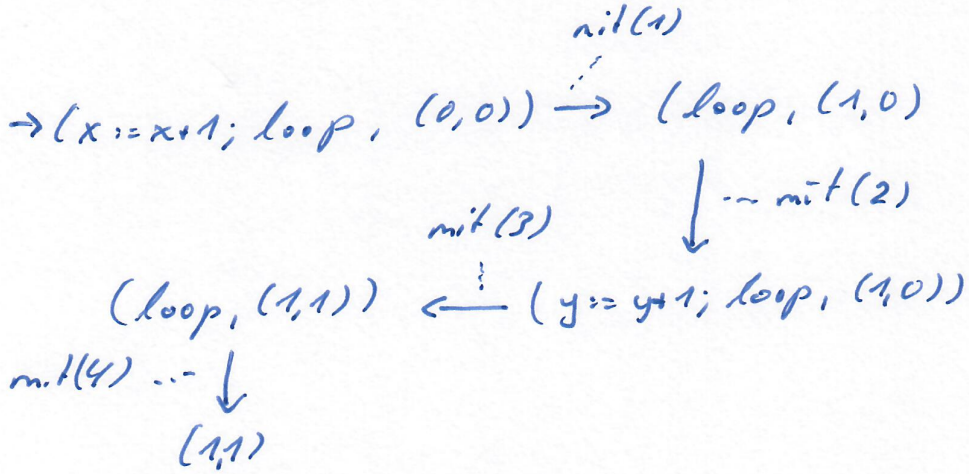
die jedem Programm c
und jedem Zustand σ
das Transitionssystem

$$[\![c]\!]\sigma := ((While \times State) \cup State, \rightarrow, (c, \sigma))$$

mit Initialkonfiguration (c, σ) zuordnet.

- Oft betrachtet man nur die
von der initialen Konfiguration erreichbaren Konfigurationen.

In obigem Beispiel:



Lemma:
Die Small-Step-Transitionsrelation ist deterministisch,
für alle $(c, \sigma) \rightarrow cf_1$ und $(c, \sigma) \rightarrow cf_2$ gilt $cf_1 = cf_2$.