Programmverifikation

Die Sprache IMP

Grundlegende Aspekte der Semantik von Programmiersprachen klären wir anhand von IMP, einer minimalistischen imperativen Programmiersprache, die uns als Studienobjekt dienen wird.

Beispiel für ein Programm in IMP:

```
y := 1;
while not n = 0 do
   y := y*x;
   n := n - 1
end
```

Beispiel für ein Programm in IMP:

```
y := 1;
while not n = 0 do
   y := y*x;
   n := n - 1
end
```

Es berechnet zu ganzen Zahlen x, n mit $n \geq 0$ die Potenz $y = x^n$. Klar ersichtlich.

Beispiel für ein Programm in IMP:

```
y := 1;
while not n = 0 do
   y := y*x;
   n := n - 1
end
```

Es berechnet zu ganzen Zahlen x, n mit $n \geq 0$ die Potenz $y = x^n$. Klar ersichtlich.

Das *schnelle Exponentiation* genannte Programm

```
y := 1;
while 2 <= n do
   q := n/2; r := n - 2*q;
   if r = 1 then y := y*x else skip end;
   n := q; x := x*x
end;
if n = 1 then y := y*x else skip end
```

berechnet die Potenz ebenfalls. Immer noch klar ersichtlich?

Wir würden dies gern *beweisen*.

Zunächst wird eine formale Definition der Sprache IMP unternommen.

Zunächst wird eine formale Definition der Sprache IMP unternommen.

Es sei Int die Menge der ganzen Zahlen im Dezimalsystem. Während also $\mathbb{Z}$ die Menge der ganzen Zahlen bezeichnet, besteht Int aus konkreten syntaktischen Darstellungen der Zahlen aus $\mathbb{Z}$.

Zunächst wird eine formale Definition der Sprache IMP unternommen.

Es sei Int die Menge der ganzen Zahlen im Dezimalsystem. Während also $\mathbb{Z}$ die Menge der ganzen Zahlen bezeichnet, besteht Int aus konkreten syntaktischen Darstellungen der Zahlen aus $\mathbb{Z}$.

Entsprechend sei Bool := {**false**, **true**}, wobei die Symbole **false** und **true** syntaktische Darstellungen der Wahrheitswerte seien.

Ein **arithmetischer Ausdruck** a, kurz **Term**, sei durch die folgenden Produktionsregeln festgelegt.

- Eine ganze Zahl aus Int ist ein Term.
- Eine Variable aus Loc ist ein Term.
- Sind a, a' Terme, so sind auch $a + a'$, $a - a'$, $a * a'$, a / a' Terme.
- Nichts anderes ist ein Term.

Wir fassen die Terme hierbei als abstrakte Syntaxbäume auf. Auf die genaue Grammatik und die Konstruktion eines Parsers will ich an dieser Stelle nicht näher eingehen, damit der Fokus auf die eigentliche Problemstellung nicht verloren geht. Die Operationen sollen auf die gewöhnliche Art geschrieben werden, die Operatoren dabei die gewöhnliche Rangfolge besitzen.

Ein **boolescher Ausdruck** b, kurz **Ausdruck**, sei durch die folgenden Produktionsregeln festgelegt.

- Die Symbole **`false`** und **`true`** sind Ausdrücke.
- Sind a, a' Terme, so sind $a = a'$ und $a <= a'$ Ausdrücke.
- Ist b ein Ausdruck, so ist auch **`not`** b ein Ausdruck.
- Sind b, b' Ausdrücke, so sind auch b **`and`** b und b **`or`** b Ausdrücke.
- Nichts anderes ist ein Ausdruck.

Ein **Kommando** c, auch **Programm** genannt, sei durch die folgenden Produktionsregeln festgelegt.

- Das Symbol **`skip`** ist ein Kommando.
- Ist X eine Variable und a ein Term, so ist $X := a$ ein Kommando.
- Sind c, c' Kommandos, so ist auch c; c' ein Kommando.
- Ist b ein Ausdruck und sind c, c' Kommandos, so ist auch **`if`** b **`then`** c_1 **`else`** c_2 **`end`** ein Kommando.
- Ist b ein Ausdruck und c ein Kommando, so ist auch **`while`** b **`do`** c **`end`** ein Kommando.
- Nichts anderes ist ein Kommando.

Denotationelle Semantik

Wir bezeichnen

- mit Aexp die Menge der arithmetischen Ausdrücke,
- mit Bexp die Menge der booleschen Ausdrücke
- mit Com die Menge der Kommandos.

Wir bezeichnen

- mit Aexp die Menge der arithmetischen Ausdrücke,
- mit Bexp die Menge der booleschen Ausdrücke
- mit Com die Menge der Kommandos.

Will man nun einen arithmetischen Ausdruck auswerten, denkt man sich dafür eine Funktion $A\colon \mathrm{Aexp} \to \mathrm{Int}$. Es ist

$$A[\![(1+2)]\!] = 3$$

usw. Nun kann ein arithmetischer Ausdruck aber auch Variablen enthalten, womit bspw. auch der Wert $A[\![(x+1)]\!]$ bezüglich $x \in \mathrm{Loc}$ bestimmt sein muss.

Die Auswertungsfunktion A wird daher parametrisiert durch den aktuellen Zustand s.
Wir haben also $A\colon \mathrm{Aexp} \to (S \to \mathrm{Int})$ bzw. $A\colon \mathrm{Aexp} \times S \to Z$.

Die Auswertungsfunktion A wird daher parametrisiert durch den aktuellen Zustand s. Wir haben also $A\colon \mathrm{Aexp} \to (S \to \mathrm{Int})$ bzw. $A\colon \mathrm{Aexp} \times S \to Z$.

Einen **Zustand** s modellieren wir schlicht als die Belegung der verfügbaren Variablen, also als eine Funktion $s \in S$ mit $S := \mathrm{Abb}(\mathrm{Loc}, \mathrm{Int})$.

Die Auswertungsfunktion A wird daher parametrisiert durch den aktuellen Zustand s. Wir haben also $A\colon \mathrm{Aexp} \to (S \to \mathrm{Int})$ bzw. $A\colon \mathrm{Aexp} \times S \to Z$.

Einen **Zustand** s modellieren wir schlicht als die Belegung der verfügbaren Variablen, also als eine Funktion $s \in S$ mit $S := \mathrm{Abb}(\mathrm{Loc}, \mathrm{Int})$.

Zum Beispiel

$$s(X) := \begin{cases} 1, & \text{wenn } X = \mathrm{x}, \\ 2, & \text{wenn } X = \mathrm{y}, \\ 0 & \text{sonst.} \end{cases}$$

Wir legen $A\colon \mathrm{Aexp} \to (S \to \mathrm{Int})$ rekursiv fest gemäß

$$
\begin{aligned}
&A[\![n]\!](s) := n,\\
&A[\![X]\!](s) := s(X),\\
&A[\![(a + a')]\!](s) := A[\![a]\!](s) + A[\![a']\!](s),\\
&A[\![(a - a')]\!](s) := A[\![a]\!](s) - A[\![a']\!](s),\\
&A[\![(a * a')]\!](s) := A[\![a]\!](s) \cdot A[\![a']\!](s),\\
&A[\![(a / a')]\!](s) := A[\![a]\!](s) / A[\![a']\!](s).
\end{aligned}
$$

bezüglich $n \in \mathrm{Int}$, $X \in \mathrm{Loc}$ und $a, a' \in \mathrm{Aexp}$. Die Operationen auf der rechten Seite sind hierbei auf die übliche Art und Weise berechnet. Wir verwenden euklidische Ganzzahldivision und setzen $n / 0 := 0$ für jedes $n \in \mathrm{Int}$.

Wir legen $B\colon \mathrm{Bexp} \to (S \to \mathrm{Bool})$ rekursiv fest gemäß

$$B[\![\,\mathbf{false}\,]\!](s) := \mathbf{false},$$
$$B[\![\,\mathbf{true}\,]\!](s) := \mathbf{true},$$
$$B[\![\,a = a'\,]\!](s) := (A[\![\,a\,]\!](s) = A[\![\,a'\,]\!](s)),$$
$$B[\![\,a <= a'\,]\!](s) := (A[\![\,a\,]\!](s) \leq A[\![\,a'\,]\!](s)),$$
$$B[\![\,(\mathbf{not}\ b)\,]\!] := \neg B[\![\,b\,]\!](s),$$
$$B[\![\,(b\ \mathbf{and}\ b')\,]\!] := B[\![\,b\,]\!](s) \wedge B[\![\,b'\,]\!](s),$$
$$B[\![\,(b\ \mathbf{or}\ b')\,]\!] := B[\![\,b\,]\!](s) \vee B[\![\,b'\,]\!](s),$$

bezüglich $a, a' \in \mathrm{Aexp}$ und $b, b' \in \mathrm{Bexp}$. Die Relationen bzw. Verknüpfungen auf der rechten Seite werden hierbei auf die übliche Art und Weise berechnet.

Wir legen $C\colon \mathrm{Com} \to (S \rightharpoonup S)$ rekursiv fest gemäß

$$C[\![\textbf{skip}]\!](s) := s,$$

$$C[\![X := a]\!](s) := s[X := A[\![a]\!](s)],$$

$$C[\![c; c']\!](s) := C[\![c']\!](C[\![c]\!](s)) = (C[\![c']\!] \circ C[\![c]\!])(s),$$

$$C[\![\textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end}]\!](s) := \begin{cases} C[\![c]\!](s), & \text{wenn } B[\![b]\!](s) = \textbf{true}, \\ C[\![c']\!](s) & \text{sonst.} \end{cases}$$

$$C[\![\textbf{while } b \textbf{ do } c \textbf{ end}]\!](s) := \begin{cases} \varphi_{b,c}(C[\![c]\!](s)), & \text{wenn } B[\![b]\!](s) = \textbf{true}, \\ s & \text{sonst} \end{cases}$$

mit $\varphi_{b,c}(s) := C[\![\textbf{while } b \textbf{ do } c \textbf{ end}]\!](s)$.

Wir legen $C\colon \mathrm{Com} \to (S \rightharpoonup S)$ rekursiv fest gemäß

$$C[\![\textbf{skip}]\!](s) := s,$$

$$C[\![X := a]\!](s) := s[X := A[\![a]\!](s)],$$

$$C[\![c; c']\!](s) := C[\![c']\!](C[\![c]\!](s)) = (C[\![c']\!] \circ C[\![c]\!])(s),$$

$$C[\![\textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end}]\!](s) := \begin{cases} C[\![c]\!](s), & \text{wenn } B[\![b]\!](s) = \textbf{true}, \\ C[\![c']\!](s) & \text{sonst.} \end{cases}$$

$$C[\![\textbf{while } b \textbf{ do } c \textbf{ end}]\!](s) := \begin{cases} \varphi_{b,c}(C[\![c]\!](s)), & \text{wenn } B[\![b]\!](s) = \textbf{true}, \\ s & \text{sonst} \end{cases}$$

mit $\varphi_{b,c}(s) := C[\![\textbf{while } b \textbf{ do } c \textbf{ end}]\!](s)$.

Bei $C[\![c]\!]$ handelt es sich um eine partielle Funktion, da die rekursive Auswertung der while-Schleife unter Umständen nicht terminiert – zum Beispiel bei

while true do skip end.

Die Zusicherungssprache

Zusicherungen sind Aussagen, die man in einem bestimmten Zustand als erfüllt sehen will. Zum Beispiel ist nach der Ausführung des Kommandos `y := x*x` die Zusicherung $y \geq 0$ erfüllt.

Zusicherungen sind Aussagen, die man in einem bestimmten Zustand als erfüllt sehen will. Zum Beispiel ist nach der Ausführung des Kommandos $y := x*x$ die Zusicherung $y \geq 0$ erfüllt.

Zusicherungen sind logische Formeln, die der Sprache der einsortigen Logik erster Stufe enstammen sollen. Die logische Sprache wird passend zur Programmiersprache IMP definiert, dergestalt dass das Diskursuniversum Int sei und die Variablen aus Loc in den Termen auftauchen dürfen. Benötigte Funktionssymbole der Signatur $\mathrm{Int}^n \to \mathrm{Int}$ und Relationssymbole der Signatur $\mathrm{Int}^n \to \mathrm{Bool}$ zu $n \in \mathbb{N}_{\geq 0}$ kann man je nach Bedarf in ihrer üblichen Bedeutung hinzufügen; die Operatoren von IMP sollen dabei aber mindestens verfügbar sein.

Zusicherungen sind Aussagen, die man in einem bestimmten Zustand als erfüllt sehen will. Zum Beispiel ist nach der Ausführung des Kommandos $y := x*x$ die Zusicherung $y \geq 0$ erfüllt.

Zusicherungen sind logische Formeln, die der Sprache der einsortigen Logik erster Stufe enstammen sollen. Die logische Sprache wird passend zur Programmiersprache IMP definiert, dergestalt dass das Diskursuniversum Int sei und die Variablen aus Loc in den Termen auftauchen dürfen. Benötigte Funktionssymbole der Signatur $\text{Int}^n \to \text{Int}$ und Relationssymbole der Signatur $\text{Int}^n \to \text{Bool}$ zu $n \in \mathbb{N}_{\geq 0}$ kann man je nach Bedarf in ihrer üblichen Bedeutung hinzufügen; die Operatoren von IMP sollen dabei aber mindestens verfügbar sein.

Wir müssen nun allerdings zwischen Variablen x, y, z und Variablen $x, y, z \in \text{Loc}$ unterscheiden. Die kursiven tauchen als freie und gebundene Variablen in den Formeln auf. Die aufrechten verhalten sich dagegen wie Konstantensymbole, über sie kann nicht quantifiziert werden.

Die Notation $I, s \models A$ stehe für die Aussage, dass die Interpretation $I = (\mathcal{M}, \beta)$ und der Zustand s die Formel A erfüllen. Hierbei ist $\mathcal{M}$ eine Struktur, die die Bedeutung der Funktions- und Relationssymbole festlegt, und β eine Belegung der kursiven Variablen. Der Zustand s belegt die aufrechten Variablen. Die Definition der Erfüllung geschieht analog zur gewöhnlichen Logik erster Stufe, weshalb ich sie hier nicht näher ausführen will.

Die Notation $I, s \models A$ stehe für die Aussage, dass die Interpretation $I = (\mathcal{M}, \beta)$ und der Zustand s die Formel A erfüllen. Hierbei ist $\mathcal{M}$ eine Struktur, die die Bedeutung der Funktions- und Relationssymbole festlegt, und β eine Belegung der kursiven Variablen. Der Zustand s belegt die aufrechten Variablen. Die Definition der Erfüllung geschieht analog zur gewöhnlichen Logik erster Stufe, weshalb ich sie hier nicht näher ausführen will.

Wir betrachten nur das Modell $\mathcal{M}_0$, das die Symbole mit ihrer üblichen Bedeutung versieht. Daher sei $\mathcal{I}_0 := \{(\mathcal{M}, \beta) \mid \mathcal{M} = \mathcal{M}_0\}$, das heißt, $\mathcal{I}_0$ sei die Menge der Interpretationen, deren Struktur $\mathcal{M}_0$ ist.

Die Notation $I, s \models A$ stehe für die Aussage, dass die Interpretation $I = (\mathcal{M}, \beta)$ und der Zustand s die Formel A erfüllen. Hierbei ist $\mathcal{M}$ eine Struktur, die die Bedeutung der Funktions- und Relationssymbole festlegt, und β eine Belegung der kursiven Variablen. Der Zustand s belegt die aufrechten Variablen. Die Definition der Erfüllung geschieht analog zur gewöhnlichen Logik erster Stufe, weshalb ich sie hier nicht näher ausführen will.

Wir betrachten nur das Modell $\mathcal{M}_0$, das die Symbole mit ihrer üblichen Bedeutung versieht. Daher sei $\mathcal{I}_0 := \{(\mathcal{M}, \beta) \mid \mathcal{M} = \mathcal{M}_0\}$, das heißt, $\mathcal{I}_0$ sei die Menge der Interpretationen, deren Struktur $\mathcal{M}_0$ ist.

Wir schreiben später $s \models A$ als Abkürzung für $I, s \models A$, sofern sich dadurch keine Zweideutigkeiten ergeben.

Der Hoare-Kalkül

An ein Kommando c können wir Zusicherungen machen. Wir notieren $\{A\}c\{B\}$ für die Aussage, dass die Aussage B unter allen Umständen nach der Ausführung von c erfüllt ist, sofern die Aussage A zuvor erfüllt war. Man nennt A diesbezüglich eine *Vorbedingung* und B eine *Nachbedingung* von c.

An ein Kommando c können wir Zusicherungen machen. Wir notieren $\{A\}c\{B\}$ für die Aussage, dass die Aussage B unter allen Umständen nach der Ausführung von c erfüllt ist, sofern die Aussage A zuvor erfüllt war. Man nennt A diesbezüglich eine *Vorbedingung* und B eine *Nachbedingung* von c.

Beispiel für ein allgemeingültiges Tripel:

$\{\textbf{true}\}$
`y := x*x`
$\{y \geq 0\}$

Die Allgemeingültigkeit des Tripels $\{A\}c\{B\}$ wird dahingehend definiert als

$$(\models \{A\}c\{B\}) :\Leftrightarrow \forall I \in \mathcal{I}_0 : \forall s \in S : (I, s \models A) \Rightarrow \forall s' \in S : C[\![c]\!](s) = s' \Rightarrow (I, s' \models B).$$

Die Allgemeingültigkeit des Tripels $\{A\}c\{B\}$ wird dahingehend definiert als

$$(\models \{A\}c\{B\}) :\Leftrightarrow \forall I \in \mathcal{I}_0 : \forall s \in S : (I, s \models A) \Rightarrow \forall s' \in S : C[\![c]\!](s) = s' \Rightarrow (I, s' \models B).$$

In Bezug auf IMP ist der Wert s', sofern $C[\![c]\!](s)$ existiert, eindeutig bestimmt. Denken wir uns nun den ungültigen Zustand $\perp$, der sich ergeben soll, wenn c eine nicht terminierende Schleife ist, bekommt man eine totale Funktion $C[\![c]\!] : S \cup \{\perp\} \to S \cup \{\perp\}$, wobei $C[\![c]\!](\perp) := \perp$ gesetzt wird. Diesbezüglich verkürzt sich die Allgemeingültigkeit zu

$$(\models \{A\}c\{B\}) \Leftrightarrow \forall I \in \mathcal{I}_0 : \forall s \in S : (I, s \models A) \Rightarrow (I, C[\![c]\!](s) \models B).$$

Hierbeit verlangt man, dass $I, \perp \models A$ für jede Formel A gilt.

Bemerkung. Wir können die Zusicherungssprache erweitern um eine modale Operation $\Box_c B$ je Kommando c, üblicherweise $[c]B$ geschrieben.

Bemerkung. Wir können die Zusicherungssprache erweitern um eine modale Operation $\Box_c B$ je Kommando c, üblicherweise $[c]B$ geschrieben. Deren Semantik sei

$$(I, s \models [c]B) \;:\Leftrightarrow\; \forall s' \in S \colon R_c(s, s') \Rightarrow (I, s' \models B)$$

mit der Zugänglichkeitsrelation $R_c(s, s') \;:\Leftrightarrow\; C[\![c]\!](s) = s'$.

Bemerkung. Wir können die Zusicherungssprache erweitern um eine modale Operation $\Box_c B$ je Kommando c, üblicherweise $[c]B$ geschrieben. Deren Semantik sei

$$(I, s \models [c]B) \;:\Leftrightarrow\; \forall s' \in S \colon R_c(s, s') \Rightarrow (I, s' \models B)$$

mit der Zugänglichkeitsrelation $R_c(s, s') \;:\Leftrightarrow\; C[\![c]\!](s) = s'$.

Diesbezüglich sind $\{A\}c\{B\}$ und $A \to [c]B$ semantisch äquivalent.

Bemerkung. Wir können die Zusicherungssprache erweitern um eine modale Operation $\Box_c B$ je Kommando c, üblicherweise $[c]B$ geschrieben. Deren Semantik sei

$$(I, s \models [c]B) \;:\Leftrightarrow\; \forall s' \in S \colon R_c(s, s') \Rightarrow (I, s' \models B)$$

mit der Zugänglichkeitsrelation $R_c(s, s') \;:\Leftrightarrow\; C[\![c]\!](s) = s'$.

Diesbezüglich sind $\{A\}c\{B\}$ und $A \to [c]B$ semantisch äquivalent.

Die so erweiterte Logik nennt man die *dynamische Logik* von IMP. Die denotationelle Semantik nimmt hierbei die Rolle einer Kripke-Semantik ein, wobei die Zustände die Kripke-Welten sind.

Wir diskutieren nun die **Schlussregeln** des Kalküls.

Wir diskutieren nun die **Schlussregeln** des Kalküls.

Für Zuweisungen gilt

$$\frac{}{\vdash \{A[X := a]\}X := a\{A\}},$$

eine Schlussregel ohne Prämissen. Mit $A[X := a]$ ist hierbei die Formel gemeint, die aus A hervorgeht, indem jedes Vorkommen von X in A durch den Term a ersetzt wird.

Wir diskutieren nun die **Schlussregeln** des Kalküls.

Für Zuweisungen gilt

$$\frac{}{\vdash \{A[X := a]\}X := a\{A\}},$$

eine Schlussregel ohne Prämissen. Mit $A[X := a]$ ist hierbei die Formel gemeint, die aus A hervorgeht, indem jedes Vorkommen von X in A durch den Term a ersetzt wird.

Zum Beispiel erkennt man das Tripel

$$\{x \geq 0\}$$
$$x := x + 1$$
$$\{x \geq 1\}$$

unschwer als allgemeingültig. Dieses fällt unter die Fittiche der Regel, indem $X := x$, $a := x + 1$ und $A := (x \geq 1)$ gesetzt wird. Damit ergibt sich $A[X := a]$ zu $x + 1 \geq 1$, was logisch äquivalent zu $x \geq 0$ ist.

Zur Gültigkeit der Regel. Wir wollen also

$$\models \{A[X := a]\}X := a\{A\}$$

zeigen.

Zur Gültigkeit der Regel. Wir wollen also

$$\models \{A[X := a]\}X := a\{A\}$$

zeigen. Sei dazu s fest, aber beliebig. Es gelte $s \models A[X := a]$. Des Weiteren gelte $C[\![X := a]\!](s) = s'$. Zu zeigen ist $s' \models A$.

Zur Gültigkeit der Regel. Wir wollen also

$$\models \{A[X := a]\}X := a\{A\}$$

zeigen. Sei dazu s fest, aber beliebig. Es gelte $s \models A[X := a]$. Des Weiteren gelte $C[\![X := a]\!](s) = s'$. Zu zeigen ist $s' \models A$.

Gemäß der denotationellen Semantik gilt $C[\![X := a]\!](s) = s[X := a]$, womit wir $s' = s[X := a]$ haben. Schließlich folgt die Behauptung vermittels der Äquivalenz

$$(s \models A[X := a]) \Leftrightarrow (s[X := a] \models A),$$

die man unschwer als richtig erkennt oder pedantisch per struktureller Induktion über den Aufbau von A beweisen kann.

Die Regel zu **skip** lautet

$$\frac{}{\vdash \{A\}\textbf{skip}\{A\}}.$$

Die Regel zu **skip** lautet

$$\frac{}{\vdash \{A\}\mathbf{skip}\{A\}}.$$

Zur Gültigkeit der Regel. Wir wollen $\models \{A\}\mathbf{skip}\{B\}$ zeigen. Dazu sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte $C[\![\mathbf{skip}]\!](s) = s'$. Zu zeigen ist $s' \models A$.

Die Regel zu **skip** lautet

$$\frac{}{\vdash \{A\}\mathbf{skip}\{A\}}.$$

Zur Gültigkeit der Regel. Wir wollen $\models \{A\}\mathbf{skip}\{B\}$ zeigen. Dazu sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte $C[\![\mathbf{skip}]\!](s) = s'$. Zu zeigen ist $s' \models A$.

Gemäß der denotationellen Semantik gilt $C[\![\mathbf{skip}]\!](s) = s$, womit wir $s' = s$ und somit bereits die Behauptung haben.

Die Regel für Sequenzen von Kommandos lautet

$$\frac{\vdash \{A\}c\{B\} \qquad \vdash \{B\}c'\{C\}}{\vdash \{A\}c;c'\{C\}}.$$

Die Regel für Sequenzen von Kommandos lautet

$$\frac{\vdash \{A\}c\{B\} \qquad \vdash \{B\}c'\{C\}}{\vdash \{A\}c;c'\{C\}}.$$

Zur Gültigkeit der Regel. Wir wollen $\models \{A\}c;c'\{C\}$ zeigen. Dazu sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte $C[\![c;c']\!](s) = s''$. Zu zeigen ist $s'' \models C$.

Die letzten beiden C's stehen für Unterschiedliches; aber eine Verwechslung ist ausgeschlossen, denke ich.

Die Regel für Sequenzen von Kommandos lautet

$$\frac{\vdash \{A\}c\{B\} \qquad \vdash \{B\}c'\{C\}}{\vdash \{A\}c;c'\{C\}}.$$

Zur Gültigkeit der Regel. Wir wollen $\models \{A\}c;c'\{C\}$ zeigen. Dazu sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte $C[\![c;c']\!](s) = s''$. Zu zeigen ist $s'' \models C$.
Die letzten beiden C's stehen für Unterschiedliches; aber eine Verwechslung ist ausgeschlossen, denke ich.

Gemäß der denotationellen Semantik existiert $s' = C[\![c]\!](s)$ mit

$$C[\![c;c']\!](s) = C[\![c']\!](s') = s''.$$

Aus $s \models A$ und $\models \{A\}c\{B\}$ folgt nun zunächst $s' \models B$. Mit $\models \{B\}c'\{C\}$ folgt daraufhin $s'' \models C$ aus $s' \models B$.

Literatur

- Glynn Winskel: *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.
- David Harel, Dexter Kozen, Jerzy Tiuryn: *Dynamic Logic*. The MIT Press, 2000.
- Benjamin C. Pierce u. a.: *Software Foundations*.

Anlage

IMP-Interpreter – Führt ein IMP-Programm gemäß der denotationellen Semantik aus. In Python verfasst, unter 300 Zeilen Quelltext.

Ende.

Dezember 2024