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

Beispiel für ein Programm in IMP:

```
y := 1; k := n;
while not k = 0 do
   y := y*x;
   k := k - 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; k := n;
while not k = 0 do
   y := y*x;
   k := k - 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; a := x; k := n;
while 2 <= k do
   q := k/2; r := k - 2*q;
   if r = 1 then y := y*a else skip end;
   k := q; a := a*a
end;
if k = 1 then y := y*a 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 aus Loc 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 **`else`** c' **`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 werden 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 $b, b' \in \mathrm{Bexp}$. Der Wahrheitswert der Relationen bzw. Verknüpfungen auf der rechten Seite wird 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 entstammen 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 $\mathtt{y := x*x}$ die Zusicherung $\mathtt{y} \geq 0$ erfüllt.

Zusicherungen sind logische Formeln, die der Sprache der einsortigen Logik erster Stufe entstammen 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.

Wir müssen nun allerdings zwischen Variablen x, y, z und Variablen $\mathtt{x}, \mathtt{y}, \mathtt{z} \in \mathrm{Loc}$ 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

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

Hierbei 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 $A \Rightarrow [c]B$ 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 $A \Rightarrow [c]B$ 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.

Ein Tripel $\{A\}c\{B\}$ fordert allerdings nicht, dass c terminiert; lediglich dass B gilt, *falls* c terminiert. Man nennt ein solches Programm *partiell korrekt* in Bezug auf A, B.

Ein Tripel $\{A\}c\{B\}$ fordert allerdings nicht, dass c terminiert; lediglich dass B gilt, *falls* c terminiert. Man nennt ein solches Programm *partiell korrekt* in Bezug auf A, B.

Terminiert das Programm zusätzlich, nennen wir es *total korrekt* in Bezug auf A, B und notieren dies $[A]c[B]$.

Ein Tripel $\{A\}c\{B\}$ fordert allerdings nicht, dass c terminiert; lediglich dass B gilt, *falls* c terminiert. Man nennt ein solches Programm *partiell korrekt* in Bezug auf A, B.

Terminiert das Programm zusätzlich, nennen wir es *total korrekt* in Bezug auf A, B und notieren dies $[A]c[B]$.

Da die Semantik von IMP eine deterministische ist, ist der Zustand $s' = C[\![c]\!](s)$, sofern dieser existiert, eindeutig bestimmt; es gibt also keine weiteren Zustände, an die wir B fordern müssten. Daher ergibt sich

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

Ein Tripel $\{A\}c\{B\}$ fordert allerdings nicht, dass c terminiert; lediglich dass B gilt, *falls* c terminiert. Man nennt ein solches Programm *partiell korrekt* in Bezug auf A, B.

Terminiert das Programm zusätzlich, nennen wir es *total korrekt* in Bezug auf A, B und notieren dies $[A]c[B]$.

Da die Semantik von IMP eine deterministische ist, ist der Zustand $s' = C[\![c]\!](s)$, sofern dieser existiert, eindeutig bestimmt; es gibt also keine weiteren Zustände, an die wir B fordern müssten. Daher ergibt sich

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

In Bezug auf die Modaloperation $\Diamond_c B$ bzw. $\langle c \rangle B$ mit der Semantik

$$(s \models \langle c \rangle B) \ :\Leftrightarrow \ \exists s' \in S \colon R_c(s, s') \wedge (s' \models B).$$

sind nun $[A]c[B]$ und $A \Rightarrow \langle c \rangle B$ semantisch äquivalent.

Außerdem bestehen die Äquivalenzen $\langle c \rangle B \Leftrightarrow \neg[c]\neg B$ und $[c]B \Leftrightarrow \neg\langle c \rangle \neg B$.

Die Formalismen stehen im engen Bezug zum dijkstraschen wp-Kalkül. Dijkstra notiert $\mathrm{wp}(c, B)$ für die schwächste Vorbedingung, engl. *weakest precondition*, unter der das Programm c in einem Zustand terminiert, in dem B erfüllt ist. Für jede Vorbedingung A gilt daher $A \Rightarrow \mathrm{wp}(c, B)$.

Die Formalismen stehen im engen Bezug zum dijkstraschen wp-Kalkül. Dijkstra notiert $\mathrm{wp}(c, B)$ für die schwächste Vorbedingung, engl. *weakest precondition*, unter der das Programm c in einem Zustand terminiert, in dem B erfüllt ist. Für jede Vorbedingung A gilt daher $A \Rightarrow \mathrm{wp}(c, B)$.

Analog steht $\mathrm{wlp}(c, B)$ für die schwächste liberale Vorbedingung, die lediglich B fordert, falls das Programm terminiert. Demnach sind $[c]B$ und $\mathrm{wlp}(c, B)$ bedeutungsgleich. Entsprechend drücken $\{A\}c\{B\}$ und $A \Rightarrow \mathrm{wlp}(c, B)$ dasselbe aus.

Die Formalismen stehen im engen Bezug zum dijkstraschen wp-Kalkül. Dijkstra notiert $\mathrm{wp}(c, B)$ für die schwächste Vorbedingung, engl. *weakest precondition*, unter der das Programm c in einem Zustand terminiert, in dem B erfüllt ist. Für jede Vorbedingung A gilt daher $A \Rightarrow \mathrm{wp}(c, B)$.

Analog steht $\mathrm{wlp}(c, B)$ für die schwächste liberale Vorbedingung, die lediglich B fordert, falls das Programm terminiert. Demnach sind $[c]B$ und $\mathrm{wlp}(c, B)$ bedeutungsgleich. Entsprechend drücken $\{A\}c\{B\}$ und $A \Rightarrow \mathrm{wlp}(c, B)$ dasselbe aus.

Aufgrund der deterministischen Natur von IMP sind des Weiteren $\langle c \rangle B$ und $\mathrm{wp}(c, B)$ bedeutungsgleich. Entsprechend drücken $[A]c[B]$ und $A \Rightarrow \mathrm{wp(c, B)}$ dasselbe aus.

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

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

Regel zur Zuweisung

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

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.

Regel zur Zuweisung

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

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.

Ob die Rechnung stimmt, prüft man so: Man betrachtet die Nachbedingung, wendet auf diese die mit der Zuweisung übereinstimmende Substitution an, und dies muss dann in der Vorbedingung resultieren.

Ob die Rechnung stimmt, prüft man so: Man betrachtet die Nachbedingung, wendet auf diese die mit der Zuweisung übereinstimmende Substitution an, und dies muss dann in der Vorbedingung resultieren.

Beweis der Gültigkeit der Regel. Wir wollen

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

zeigen.

Ob die Rechnung stimmt, prüft man so: Man betrachtet die Nachbedingung, wendet auf diese die mit der Zuweisung übereinstimmende Substitution an, und dies muss dann in der Vorbedingung resultieren.

Beweis der Gültigkeit der Regel. Wir wollen

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

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 $s' \models A$.

Ob die Rechnung stimmt, prüft man so: Man betrachtet die Nachbedingung, wendet auf diese die mit der Zuweisung übereinstimmende Substitution an, und dies muss dann in der Vorbedingung resultieren.

Beweis der Gültigkeit der Regel. Wir wollen

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

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

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

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

Regel zum leeren Kommando

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

Regel zum leeren Kommando

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

Beweis ihrer Gültigkeit. 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 $s' \models A$.

Regel zum leeren Kommando

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

Beweis ihrer Gültigkeit. 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. □

Regel zur Sequenz von Kommandos

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

Regel zur Sequenz von Kommandos

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

Beweis ihrer Gültigkeit. 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.

Regel zur Sequenz von Kommandos

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

Beweis ihrer Gültigkeit. 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 $s' \models B$. □

Regel zur Verzweigung

$$\frac{\vdash \{A \wedge b\} c \{C\} \qquad \vdash \{A \wedge \neg b\} c' \{C\}}{\vdash \{A\} \textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end} \{C\}}$$

Regel zur Verzweigung

$$\frac{\vdash \{A \wedge b\} c \{C\} \qquad \vdash \{A \wedge \neg b\} c' \{C\}}{\vdash \{A\} \textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end} \{C\}}$$

Beweis ihrer Gültigkeit. Sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte

$$C[\![\textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end}]\!](s) = s'.$$

Zu zeigen ist $s' \models C$.

Regel zur Verzweigung

$$\frac{\vdash \{A \wedge b\} c \{C\} \qquad \vdash \{A \wedge \neg b\} c' \{C\}}{\vdash \{A\} \textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end} \{C\}}$$

Beweis ihrer Gültigkeit. Sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte

$$C[\![\textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end}]\!](s) = s'.$$

Zu zeigen ist $s' \models C$. Fallunterscheidung. Im Fall $B[\![b]\!](s) = \textbf{true}$ gilt $s \models b$, also $s \models A \wedge b$. Außerdem ergibt sich in diesem Fall die Vereinfachung

$$C[\![\textbf{if } b \textbf{ then } c \textbf{ else } c' \textbf{ end}]\!](s) = C[\![c]\!](s).$$

Vermittels $\models \{A \wedge b\} c \{C\}$ erhalten wir somit $s' \models C$. Die Argumentation im Fall $B[\![b]\!](s) = \textbf{false}$ verläuft analog. □

Regel zur Schleife

$$\frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \textbf{while } b \textbf{ do } c \textbf{ end} \{A \wedge \neg b\}}$$

Regel zur Schleife

$$\frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \textbf{while } b \textbf{ do } c \textbf{ end} \{A \wedge \neg b\}}$$

Sofern A also sowohl vor der Schleife gilt, als auch vor und hinter dem Schleifenrumpf, muss A auch hinter der Schleife gelten. Man nennt A hierbei eine *Schleifeninvariante*. Die Auffindung einer zielführenden Invariante stellt eine wesentliche Schwierigkeit bei der Programmverifikation dar.

Beweis ihrer Gültigkeit. Sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte

$$C[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}]\!](s) = s''.$$

Zu zeigen ist $s'' \models A \wedge \neg b$.

Beweis ihrer Gültigkeit. Sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte

$$C[\![\textbf{while } b \textbf{ do } c \textbf{ end}]\!](s) = s''.$$

Zu zeigen ist $s'' \models A \wedge \neg b$. Induktion über die Anzahl der Durchläufe. Im Anfang findet kein Durchlauf statt. Das geht aber nur, wenn $B[\![b]\!](s) = \textbf{false}$, also $s \models \neg b$. Da dem Zustand s nach der Semantik der Schleife in diesem Fall keine Änderung widerfährt, gilt $s = s''$. Demnach gilt $s'' \models A$ und $s'' \models \neg b$, also $s'' \models A \wedge \neg b$.

Beweis ihrer Gültigkeit. Sei s fest, aber beliebig. Es gelte $s \models A$. Des Weiteren gelte

$$C[\![\textbf{while } b \textbf{ do } c \textbf{ end}]\!](s) = s''.$$

Zu zeigen ist $s'' \models A \wedge \neg b$. Induktion über die Anzahl der Durchläufe. Im Anfang findet kein Durchlauf statt. Das geht aber nur, wenn $B[\![b]\!](s) = \textbf{false}$, also $s \models \neg b$. Da dem Zustand s nach der Semantik der Schleife in diesem Fall keine Änderung widerfährt, gilt $s = s''$. Demnach gilt $s'' \models A$ und $s'' \models \neg b$, also $s'' \models A \wedge \neg b$.

Zum Induktionsschritt. Da die Schleife durchlaufen wird, existiert s' mit $C[\![c]\!](s) = s'$. Die restlichen null der mehr Schleifendurchläufe führen dann zum Zustand s'', das heißt, $\varphi_{b,c}(s') = s''$. Die Induktionsvoraussetzung ist, dass $s'' \models A \wedge \neg b$ aus $s' \models A$ folgt. Es verbleibt also $s' \models A$ zu zeigen. Weil der erste Durchlauf stattfindet, muss des Weiteren $B[\![b]\!](s) = \textbf{true}$, also $s \models b$ gelten, womit wir $s \models A \wedge b$ haben. Vermittels $\models \{A \wedge b\}c\{A\}$, was ja Kraft der Prämisse der Regel zur Verfügung steht, erhält man schließlich $s' \models A$. □

Regel zur Verstärkung der Vorbedingung

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}c\{B\}}{\vdash \{A'\}c\{B\}}$$

Regel zur Verstärkung der Vorbedingung

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}c\{B\}}{\vdash \{A'\}c\{B\}}$$

Beweis ihrer Gültigkeit. Es gelte $s \models A'$. Des Weiteren gelte $C[\![c]\!](s) = s'$. Zu zeigen ist $s' \models B$.

Regel zur Verstärkung der Vorbedingung

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}c\{B\}}{\vdash \{A'\}c\{B\}}$$

Beweis ihrer Gültigkeit. Es gelte $s \models A'$. Des Weiteren gelte $C[\![c]\!](s) = s'$. Zu zeigen ist $s' \models B$. Mit $\models A' \Rightarrow A$ erhält man zunächst $s \models A$. Mit $\models \{A\}c\{B\}$ daraufhin $s' \models B$. $\square$

Regel zur Abschwächung der Nachbedingung

$$\frac{\vdash \{A\}c\{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A\}c\{B'\}}$$

Regel zur Abschwächung der Nachbedingung

$$\frac{\vdash \{A\}c\{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A\}c\{B'\}}$$

Beweis ihrer Gültigkeit. Es gelte $s \models A$. Des Weiteren gelte $C[\![c]\!](s) = s'$. Zu zeigen ist $s' \models B'$.

Regel zur Abschwächung der Nachbedingung

$$\frac{\vdash \{A\}c\{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A\}c\{B'\}}$$

Beweis ihrer Gültigkeit. Es gelte $s \models A$. Des Weiteren gelte $C[\![c]\!](s) = s'$. Zu zeigen ist $s' \models B'$. Mit $\models \{A\}c\{B\}$ erhält man zunächst $s' \models B$. Mit $\models B \Rightarrow B'$ erhält man daraufhin $s' \models B'$. $\square$.

Ersetzungsregeln

$$\frac{\vdash A \Leftrightarrow A' \qquad \vdash \{A\}c\{B\}}{\vdash \{A'\}c\{B\}}, \qquad \frac{\vdash B \Leftrightarrow B' \qquad \vdash \{A\}c\{B\}}{\vdash \{A\}c\{B'\}}$$

Ersetzungsregeln

$$\frac{\vdash A \Leftrightarrow A' \qquad \vdash \{A\}c\{B\}}{\vdash \{A'\}c\{B\}}, \qquad \frac{\vdash B \Leftrightarrow B' \qquad \vdash \{A\}c\{B\}}{\vdash \{A\}c\{B'\}}$$

Beweis. Folgt unmittelbar aus der Regel zur Verstärkung der Vorbedingung bzw. Abschwächung der Nachbedingung. □

Verifikation des ersten Programms

Gesucht ist ein Beweis des Tripels:

```
{n ≥ 0}
y := 1; k := n;
while not k = 0 do
   y := y*x;
   k := k - 1
end
{y = x^n}
```

Vermittels der Regel zur Zuweisung – zuzüglich Reflexivität der Gleichheit und der Ersetzungsregel – ergibt sich erst einmal die Ableitung:

$$\dfrac{\dfrac{}{\vdash n \geq 0 \Leftrightarrow n \geq 0 \wedge 1 = 1} \quad \dfrac{}{\vdash \{n \geq 0 \wedge 1 = 1\}\ y\ :=\ 1\ \{n \geq 0 \wedge y = 1\}}}{\vdash \{n \geq 0\}\ y\ :=\ 1\ \{n \geq 0 \wedge y = 1\}}$$

Vermittels der Regel zur Zuweisung – zuzüglich Reflexivität der Gleichheit und der Ersetzungsregel – ergibt sich erst einmal die Ableitung:

$$\dfrac{\dfrac{}{\vdash n \geq 0 \Leftrightarrow n \geq 0 \wedge 1 = 1} \quad \dfrac{}{\vdash \{n \geq 0 \wedge 1 = 1\}\ y\ :=\ 1\ \{n \geq 0 \wedge y = 1\}}}{\vdash \{n \geq 0\}\ y\ :=\ 1\ \{n \geq 0 \wedge y = 1\}}$$

Wir fügen bereits ermittelte Zusicherungen in den Quelltext ein:

```
{n ≥ 0}
y := 1; k := n;
{k ≥ 0 ∧ y = 1}
while not k = 0 do
   y := y*x;
   k := k - 1
end
{y = x^n}?
```

Die Zielführende Schleifeninvariante ist hier $y = x^{n-k}$. Vor der Schleife gilt ja $n = k$. Mit $x^0 = 1$ kommt man somit auf $y = 1$. Bezüglich $0^0 := 1$ gilt dies auch im Fall $x = 0$.

Die Zielführende Schleifeninvariante ist hier $y = x^{n-k}$. Vor der Schleife gilt ja $n = k$. Mit $x^0 = 1$ kommt man somit auf $y = 1$. Bezüglich $0^0 := 1$ gilt dies auch im Fall $x = 0$.

Wir durchziehen das Programm nun gemäß den Regeln sukzessive mit Zusicherungen und gelangen daraufhin zum Abschluss der Verifikation:

$\{n \geq 0\}$
```
y := 1; k := n;
```
$\{k \geq 0 \wedge y = 1 \wedge y = x^{n-k}\}$
```
while not k = 0 do
```
$\quad \{y = x^{n-k}\}$
$\quad \{y \cdot x = x^{n-k} \cdot x\}$
```
    y := y*x;
```
$\quad \{y = x^{n-k} \cdot x = x^{n-(k-1)}\}$
```
    k := k - 1
```
$\quad \{y = x^{n-k}\}$
```
end
```
$\{y = x^{n-k} \wedge k = 0\}$
$\{y = x^n\}$

Verifikation des zweiten Programms

Gesucht ist der Beweis von:

$\{n \geq 0\}$

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

$\{y = x^n\}$

Gesucht ist der Beweis von:

$\{n \geq 0\}$
```
y := 1; a := x; k := n;
while 2 <= k do
   q := k/2; r := k - 2*q;
   if r = 1 then y := y*a else skip end;
   k := q; a := a*a
end;
if k = 1 then y := y*a else skip end
```
$\{y = x^n\}$

Wir betrachten das Programm zunächst vom Ende aus. Dort findet sich das Tripel

$\{(y = x^n \wedge k = 0) \vee (y \cdot a = x^n \wedge k = 1)\}$
```
if k = 1 then y := y*a else skip end
```
$\{y = x^n\}$.

Nimmt man bei der Vorbedingung nämlich die Fallunterscheidung in die beiden Seiten der Disjunktion vor, gelangt man in beiden Fällen zur Nachbedingung.

Am Anfang der Schleife gilt $a^k = x^n$, was etwas mit der Schleifeninvariante zu tun haben könnte. Ist diese Gleichung so modifizierbar, dass unter ihr die Zusicherung

$$(y = x^n \wedge k = 0) \vee (y \cdot a = x^n \wedge k = 1)$$

gilt?

Am Anfang der Schleife gilt $a^k = x^n$, was etwas mit der Schleifeninvariante zu tun haben könnte. Ist diese Gleichung so modifizierbar, dass unter ihr die Zusicherung

$$(y = x^n \wedge k = 0) \vee (y \cdot a = x^n \wedge k = 1)$$

gilt? Ja, nämlich zu $y \cdot a^k = x^n$, denn

im Fall $k = 0$ gilt $y \cdot a^k = y$,

im Fall $k = 1$ gilt $y \cdot a^k = y \cdot a$.

Am Anfang der Schleife gilt $a^k = x^n$, was etwas mit der Schleifeninvariante zu tun haben könnte. Ist diese Gleichung so modifizierbar, dass unter ihr die Zusicherung

$$(y = x^n \wedge k = 0) \vee (y \cdot a = x^n \wedge k = 1)$$

gilt? Ja, nämlich zu $y \cdot a^k = x^n$, denn

im Fall $k = 0$ gilt $y \cdot a^k = y$,

im Fall $k = 1$ gilt $y \cdot a^k = y \cdot a$.

Die Gleichung $y \cdot a^k = x^n$ gilt ebenfalls vor der Schleife, da dort $y = 1$ ist. Tatsächlich stellt sie eine zielführende Schleifeninvariante dar. Es verbleibt also zu bestätigen, dass sie auch am Ende des Schleifenrumpfs gilt.

Am Anfang der Schleife gilt $a^k = x^n$, was etwas mit der Schleifeninvariante zu tun haben könnte. Ist diese Gleichung so modifizierbar, dass unter ihr die Zusicherung

$$(y = x^n \wedge k = 0) \vee (y \cdot a = x^n \wedge k = 1)$$

gilt? Ja, nämlich zu $y \cdot a^k = x^n$, denn

im Fall $k = 0$ gilt $y \cdot a^k = y$,

im Fall $k = 1$ gilt $y \cdot a^k = y \cdot a$.

Die Gleichung $y \cdot a^k = x^n$ gilt ebenfalls vor der Schleife, da dort $y = 1$ ist. Tatsächlich stellt sie eine zielführende Schleifeninvariante dar. Es verbleibt also zu bestätigen, dass sie auch am Ende des Schleifenrumpfs gilt.

Zunächst ergibt sich

$\{y \cdot a^k = x^n\}$
`q := k/2; r := k - 2*q`
$\{y \cdot a^k = x^n \wedge q = \lfloor \frac{k}{2} \rfloor \wedge r = k - 2q\}$.

Wegen $r \in \{0, 1\}$ findet sich bei der Verzweigung nun

$\{y \cdot a^k = x^n \wedge q = \lfloor \frac{k}{2} \rfloor \wedge r = k - 2q\}$
if r = 1 **then** y := y∗a **else skip end**
$\{((y \cdot a^{k-1} = x^n \wedge r = 1) \vee (y \cdot a^k = x^n \wedge r = 0)) \wedge r = k - 2q\}$.

Wegen $r \in \{0, 1\}$ findet sich bei der Verzweigung nun

$\{y \cdot a^k = x^n \wedge q = \lfloor \frac{k}{2} \rfloor \wedge r = k - 2q\}$
if r = 1 **then** y := y∗a **else skip end**
$\{((y \cdot a^{k-1} = x^n \wedge r = 1) \vee (y \cdot a^k = x^n \wedge r = 0)) \wedge r = k - 2q\}$.

Die lange Nachbedingung kann man kompakter fassen als

$$y \cdot a^{k-r} = x^n \wedge r = k - 2q,$$

wobei sich des Weiteren $k - r = 2q$ ergibt.

Wegen $r \in \{0, 1\}$ findet sich bei der Verzweigung nun

$\{y \cdot a^k = x^n \wedge q = \lfloor \frac{k}{2} \rfloor \wedge r = k - 2q\}$
if `r = 1` **then** `y := y*a` **else skip end**
$\{((y \cdot a^{k-1} = x^n \wedge r = 1) \vee (y \cdot a^k = x^n \wedge r = 0)) \wedge r = k - 2q\}$.

Die lange Nachbedingung kann man kompakter fassen als

$$y \cdot a^{k-r} = x^n \wedge r = k - 2q,$$

wobei sich des Weiteren $k - r = 2q$ ergibt.

Zum Rest des Schleifenrumpfs findet sich somit schließlich

$\{y \cdot a^{2q} = x^n\}$
`k := q`
$\{y \cdot a^{2k} = x^n\}$
`a := a*a`
$\{y \cdot a^k = x^n\}$.

Quod erat demonstrandum.

Maschinengestützte Formalisierung des Kalküls

Im Anschluss will ich noch eine Formalisierung der Überlegungen in Coq ausführen. Dies setzt allerdings Grundwissen über die Sprache Gallina voraus, und wie logisches Schließen vermittels Taktiken abläuft.

Der vollständige Quelltext findet sich im Anhang. Um den Ablauf der Beweise zu betrachten, muss CoqIDE genutzt werden.

Wir indizieren die Programmvariablen durch die natürlichen Zahlen.
Hierbei steht `loc 0`, `loc 1`, `loc 2` usw. für x_0, x_1, x_2 usw.

Ein Zustand stellt sich daher schlicht als Funktion $s : \mathbb{N} \to \text{Int}$ dar. Für die ganzen Zahlen Int verwenden wir hierbei den Typ Z aus `ZArith`.

```
Require Import ZArith.ZArith.
Require Import Bool.Bool.

Definition Loc := nat.
Definition State := Loc -> Z.
```

Als nächstes erfolgt die Erklärung der Sprache IMP vermittels induktiver Tyen.

Die Division will ich der Prägnanz halber erst einmal entfallen lassen. Wir können sie später ggf. nachtragen.

```
Inductive Aexpr :=
| int (n: Z)
| loc (X: Loc)
| add (a1 a2: Aexpr)
| sub (a1 a2: Aexpr)
| mul (a1 a2: Aexpr).
```

```
Inductive Com :=
| Skip
| Assign (X: Loc) (a: Aexpr)
| Seq (c1 c2: Com)
| If (b: Bexpr) (c1 c2: Com)
| While (b: Bexpr) (c: Com).
```

```
Inductive Bexpr :=
| btrue
| bfalse
| beq (a1 a2: Aexpr)
| ble (a1 a2: Aexpr)
| bnot (b: Bexpr)
| band (b1 b2: Bexpr)
| bor (b1 b2: Bexpr).
```

Nun die Auswertungsfunktionen A, B. Für die Werte von B nehmen wir `bool` statt `Bexpr`, weil dieses bereits Funktionalität zum Argumentieren mit sich bringt.

```
Fixpoint evA (a: Aexpr) (s: State): Z :=
  match a with
  | int n => n
  | loc X => s X
  | add a1 a2 => evA a1 s + evA a2 s
  | sub a1 a2 => evA a1 s - evA a2 s
  | mul a1 a2 => evA a1 s * evA a2 s
  end.

Fixpoint evB (b: Bexpr) (s: State): bool :=
  match b with
  | btrue => true
  | bfalse => false
  | beq a1 a2 => Z.eqb (evA a1 s) (evA a2 s)
  | ble a1 a2 => Z.leb (evA a1 s) (evA a2 s)
  | bnot b => negb (evB b s)
  | band b1 b2 => andb (evB b1 s) (evB b2 s)
  | bor b1 b2 => orb (evB b1 s) (evB b2 s)
  end.
```

Es wäre nun denkbar, die Terme und Formeln der Zusicherungssprache ebenfalls vermittels induktiver Typen zu erklären. Dies führt allerdings zu einem Rattenschwanz von Umständlichkeiten, da es zu einer sogenannten *tiefen Einbettung* des logischen Systems führt. Wir müssten, wie für jede neue logische Sprache, einen Kalkül des natürlichen Schließens formulieren und streng genommen auch noch dessen Korrektheit beweisen.

Es wäre nun denkbar, die Terme und Formeln der Zusicherungssprache ebenfalls vermittels induktiver Typen zu erklären. Dies führt allerdings zu einem Rattenschwanz von Umständlichkeiten, da es zu einer sogenannten *tiefen Einbettung* des logischen Systems führt. Wir müssten, wie für jede neue logische Sprache, einen Kalkül des natürlichen Schließens formulieren und streng genommen auch noch dessen Korrektheit beweisen.

Wir unternehmen stattdessen den Ansatz, die Zusicherungen als Prädikate darzustellen, die Zustände als Argument bekommen. Eine Zusicherung A sei also eine Funktion

$$A : \texttt{State} \to \texttt{Prop}.$$

Es wäre nun denkbar, die Terme und Formeln der Zusicherungssprache ebenfalls vermittels induktiver Typen zu erklären. Dies führt allerdings zu einem Rattenschwanz von Umständlichkeiten, da es zu einer sogenannten *tiefen Einbettung* des logischen Systems führt. Wir müssten, wie für jede neue logische Sprache, einen Kalkül des natürlichen Schließens formulieren und streng genommen auch noch dessen Korrektheit beweisen.

Wir unternehmen stattdessen den Ansatz, die Zusicherungen als Prädikate darzustellen, die Zustände als Argument bekommen. Eine Zusicherung A sei also eine Funktion

$$A : \texttt{State} \to \texttt{Prop}.$$

Die Variante des Zustandes s wird auf die übliche Weise erklärt. Im Folgenden steht `variant s X n` also für $s[X := n]$ bzw. $s[n/X]$.

```
Definition variant (s: State) (X: Loc) (n: Z): State :=
  fun Y => if Nat.eqb X Y then n else s Y.
```

Die Auswertungsfunktion C ist wie gesagt eine partielle. Als Folge dessen tut sich das Problem auf, dass ihre Implementierung unmöglich wird, da in der Typentheorie nur solche Rekursionen fassbar sind, deren Terminierung geklärt ist.

Wir lösen dieses Problem in zwei Schritten. Erstens wird die partielle Funktion C als Relation formuliert. Statt $C[\![c]\!](s) = s'$ notieren wir also $C[\![c]\!](s)(s')$ bzw. ev`C c s s'`. Zweitens beschränken wir die rekursive Auswertung der Schleife über eine rekursive Konstruktion `iter` auf eine maximale Tiefe N. Da nicht klar ist, wie groß N bei

$$C[\![\textbf{while}\ b\ \textbf{do}\ c\ \textbf{end}]\!](s)(s')$$

mindestens sein muss, wird lediglich die Existenz von N gefordert.

```
Fixpoint iter (ev: State -> State -> Prop)
(N: nat) (b: Bexpr) (s s1: State): Prop :=
  match N with
  | 0 => False
  | S N => if evB b s then
      exists s0, ev s s0 /\ iter ev N b s0 s1
    else s1 = s
  end.

Fixpoint evC (c: Com) (s s1: State): Prop :=
  match c with
  | Skip => s1 = s
  | Assign X a => s1 = variant s X (evA a s)
  | Seq c1 c2 => exists s0, evC c1 s s0 /\ evC c2 s0 s1
  | If b c1 c2 => if evB b s then evC c1 s s1 else evC c2 s s1
  | While b c => exists N, iter (evC c) N b s s1
  end.
```

Nun kommen die Erklärungen zur Gültigkeit von Zusicherungen.

```
Definition Assertion := State -> Prop.
Definition sat (s: State) (A: Assertion): Prop := A s.

Definition valid (A: Assertion) (c: Com) (B: Assertion) :=
  forall s, sat s A -> forall s1, evC c s s1 -> sat s1 B.

Definition subst (A: Assertion) (X: Loc) (a: Aexpr): Assertion :=
  fun s => A (variant s X (evA a s)).
```

Diese Festlegungen sind folgendermaßen zu verstehen:

- Es steht `sat s A` für die Erfüllung $s \models A$.
- Es steht `valid A c B` für die Gültigkeit des Tripels $\{A\}c\{B\}$.
- Es steht `subst A X a` für die Substitution $A[X := a]$, die direkt als $s \mapsto A(s[X := A[\![a]\!](s)])$ definiert wird, insofern die Formel von A verborgen bleibt und das Substitutionslemma somit entfallen muss. Die beiden A sind von unterschiedlicher Bedeutung – das zweite steht für evA, siehe oben.

Die Formalisierung des Beweises der Gültigkeit der jeweiligen Schlussregel kann nun unternommen werden.

Zum leeren Kommando findet sich:

```
Theorem skip_intro_is_valid A:
  valid A Skip A.
Proof.
  unfold valid. intros s hs. intros s1 hs1.
  simpl evC in hs1. rewrite hs1. exact hs.
Qed.
```

Zur Zuweisung findet sich:

```
Theorem assign_intro_is_valid A X a:
  valid (subst A X a) (Assign X a) A.
Proof.
  unfold valid. intros s hs. intros s1 hs1.
  simpl evC in hs1.
  unfold sat in hs. unfold subst in hs.
  rewrite hs1. exact hs.
Qed.
```

Zur Sequenz von Kommandos findet sich:

```
Theorem seq_intro_is_valid A B C c1 c2:
  valid A c1 B -> valid B c2 C -> valid A (Seq c1 c2) C.
Proof.
  intros h1 h2. unfold valid. intros s hs. intros s2 hs2.
  simpl evC in hs2.
  destruct hs2 as (s1, (h11, h12)).
  unfold valid in h1. unfold valid in h2.
  assert (h1 := h1 s hs s1 h11).
  exact (h2 s1 h1 s2 h12).
Qed.
```

Zur Verzweigung findet sich:

```
Definition Conj (A: Assertion) (b: Bexpr) :=
  fun s => sat s A /\ evB b s = true.

Theorem if_intro_is_valid A b C c1 c2:
  valid (Conj A b) c1 C -> valid (Conj A (bnot b)) c2 C ->
  valid A (If b c1 c2) C.
Proof.
  intros h1 h2. unfold valid. intros s hs s1 hs1.
  simpl evC in hs1.
  destruct (evB b s) eqn:heq.
  * unfold valid in h1. apply (h1 s).
    - unfold sat. unfold Conj. rewrite heq.
      exact (conj hs (eq_refl true)).
    - exact hs1.
  * unfold valid in h2. apply (h2 s).
    - unfold sat. unfold Conj.
      simpl evB. rewrite heq. simpl.
      exact (conj hs (eq_refl true)).
    - exact hs1.
Qed.
```

Zur Schleife findet sich:

```
Theorem while_intro_is_valid A b c:
  valid (Conj A b) c A -> valid A (While b c) (Conj A (bnot b)).
Proof.
  intro h. unfold valid. intros s hs. intros s2 hs2.
  simpl evC in hs2. destruct hs2 as (N, hiter).
  revert s s2 hs hiter.
  induction N as [| N ih].
  * intros s s2 hs hiter. simpl iter in hiter.
    exfalso. exact hiter.
  * intros s s2 hs hiter. simpl iter in hiter.
    destruct (evB b s) eqn:heq.
    - destruct hiter as (s1, (h11, h12)).
      apply (ih s1 s2). clear ih.
      -- unfold valid in h. apply (h s). clear h.
         --- unfold sat. unfold Conj. exact (conj hs heq).
         --- exact h11.
      -- exact h12.
    - rewrite hiter. unfold sat. unfold Conj. split.
      -- exact hs.
      -- simpl evB. rewrite heq. simpl. reflexivity.
Qed.
```

Zur Verstärkung der Vorbedingung findet sich:

```
Theorem strengthen_precondition_is_valid {A1 A2 B c}:
  (forall s, A2 s -> A1 s) -> valid A1 c B -> valid A2 c B.
Proof.
  intros h1 h2. unfold valid. intros s hs s1 hs1.
  unfold sat in hs. apply (h1 s) in hs. clear h1.
  unfold valid in h2. apply (h2 s).
  * unfold sat. exact hs.
  * exact hs1.
Qed.
```

Zur Abschwächung der Nachbedingung findet sich:

```
Theorem weaken_postcondition_is_valid {A B1 B2 c}:
  (forall s, B1 s -> B2 s) -> valid A c B1 -> valid A c B2.
Proof.
  intros h1 h2. unfold valid. intros s hs s1 hs1.
  unfold sat. apply (h1 s1). clear h1.
  fold (sat s1 B1). unfold valid in h2. apply (h2 s).
  * exact hs.
  * exact hs1.
Qed.
```

Kurzes Anwendungsbeispiel

Beweis der Gültigkeit des Tripels $\{x = 0\}$ `x := x + 1` $\{x = 1\}$.

Kurzes Anwendungsbeispiel

Beweis der Gültigkeit des Tripels $\{x = 0\}\ x\ :=\ x\ +\ 1\ \{x = 1\}$.

```
Module Example1.
  Open Scope Z_scope.
  Definition x: nat := 0.

  Goal valid
    (fun s => s x = 0)
    (Assign x (add (loc x) (int 1)))
    (fun s => s x = 1).
  Proof.
    assert (h1 := assign_intro_is_valid
      (fun s => s x = 1) x (add (loc x) (int 1))).
    unfold subst in h1. unfold variant in h1. simpl in h1.
    assert (h2: forall s, s x = 0 -> s x + 1 = 1). {
      intros s heq. rewrite <- (Z.add_cancel_r _ _ 1) in heq.
      simpl in heq. exact heq.
    }
    exact (strengthen_precondition_is_valid h2 h1).
  Qed.
End Example1.
```

Literatur

- Glynn Winskel: *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.
- Krzysztof R. Apt, Ernst-Rüdiger Olderog: *Fifty years of Hoare's logic*. In: *Formal Aspects of Computing*. Band 31, Nr. 6, 2019, S. 751–807. doi:10.1007/s00165-019-00501-3.
- Edsger W. Dijkstra: *A Discipline of Programming*. Prentice Hall, 1976.
- Benjamin C. Pierce u. a.: *Software Foundations*.
- Nicolas Troquard, Philippe Balbiani: *Propositional Dynamic Logic*. In: *The Stanford Encyclopedia of Philosophy*.
- David Harel, Dexter Kozen, Jerzy Tiuryn: *Dynamic Logic*. The MIT Press, 2000.

Anlagen

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

Ende.

Januar 2025