

# Formale Grundlagen der Informatik 3 –

## *9. Axiome für $\mathcal{L}$ -Programme*

Christoph Walther  
TU Darmstadt

# 1 Axiome aus Datentypdefinitionen

- Um *Beweise* über Datentypen führen zu können, muß die Bedeutung von Datentypen formal durch *Axiome* erfaßt werden.
- **Vorgehen für formale Definition:** Erzeuge aus einer Datentypdefinition schematisch eine Menge  $AX_{\text{struc}[\text{@T}_1, \dots, \text{@T}_n]}$  von Gleichungen.
- *Konkret:* Gegeben Datentypdefinition

$$\begin{aligned}
 \text{structure struc}[\text{@T}_1, \dots, \text{@T}_n] &<= \\
 &\text{cons}_1(\text{sel}_{1,1}: \text{struc}_{1,1}, \dots, \text{sel}_{1,n_1}: \text{struc}_{1,n_1}), \\
 &\quad \vdots \\
 &\text{cons}_k(\text{sel}_{k,1}: \text{struc}_{k,1}, \dots, \text{sel}_{k,n_k}: \text{struc}_{k,n_k})
 \end{aligned} \tag{1}$$

## Bedeutung informell:

- Mit **Konstruktoren** wird Menge **aller Daten** von  $\text{struc}[\text{@T}_1, \dots, \text{@T}_n]$  gebildet
- Mit **Selektoren** können die Daten von  $\text{struc}[\text{@T}_1, \dots, \text{@T}_n]$  **zerlegt** werden
- $\text{eq}_{\text{struc}[\text{@T}_1, \dots, \text{@T}_n]}$  bezeichnet **Gleichheit** im Datentyp  $\text{struc}[\text{@T}_1, \dots, \text{@T}_n]$
- mit  $\text{if}_{\text{struc}[\text{@T}_1, \dots, \text{@T}_n]}$  **bedingte Ausdrücke** über  $\text{struc}[\text{@T}_1, \dots, \text{@T}_n]$

## Jetzt formal:

**Definition 1** (*Axiome einer Datentypdefinition,  $AX_{struct[@T_1, \dots, @T_n]}$* )

Einer Datentypdefinition für  $struct[@T_1, \dots, @T_n]$  wird die Gleichungsmenge  $AX_{struct[@T_1, \dots, @T_n]}$  wie folgt zugeordnet:

(1) für alle  $i \in \{1, \dots, k\}$  mit  $n_i = 0$  :

(a)  $eq_{struct[@T_1, \dots, @T_n]}(\mathbf{cons}_i, \mathbf{cons}_i) \equiv \mathbf{true}$

(2) für alle  $i \in \{1, \dots, k\}$  mit  $n_i > 0$  und für alle  $h \in \{1, \dots, n_i\}$  :

(a)  $\forall x_1:struct_{i,1}, \dots, x_{n_i}:struct_{i,n_i}. \mathbf{sel}_{i,h}(\mathbf{cons}_i(x_1, \dots, x_{n_i})) \equiv x_h$

(b)  $\forall x_1, y_1:struct_{i,1}, \dots, x_{n_i}, y_{n_i}:struct_{i,n_i}$   
 $eq_{struct[@T_1, \dots, @T_n]}(\mathbf{cons}_i(x_1, \dots, x_{n_i}), \mathbf{cons}_i(y_1, \dots, y_{n_i}))$   
 $\equiv \mathbf{EQ}(\langle x_1, \dots, x_{n_i} \rangle, \langle y_1, \dots, y_{n_i} \rangle)$

(3) für alle  $i, i' \in \{1, \dots, k\}$  mit  $i \neq i'$  :

(a)  $\forall x_1:struct_{i,1}, \dots, x_{n_i}:struct_{i,n_i}, y_1:struct_{i',1}, \dots, y_{n_{i'}}:struct_{i',n_{i'}}$   
 $eq_{struct[@T_1, \dots, @T_n]}(\mathbf{cons}_i(x_1, \dots, x_{n_i}), \mathbf{cons}_{i'}(y_1, \dots, y_{n_{i'}})) \equiv \mathbf{false}$

(4) (a)  $\forall x, y:struct[@T_1, \dots, @T_n]. \mathbf{if}_{struct[@T_1, \dots, @T_n]}(\mathbf{true}, x, y) \equiv x$

(b)  $\forall x, y:struct[@T_1, \dots, @T_n]. \mathbf{if}_{struct[@T_1, \dots, @T_n]}(\mathbf{false}, x, y) \equiv y$

(5) für alle  $i \in \{1, \dots, k\}$  :

(a)  $\forall x:struct[@T_1, \dots, @T_n]$   
 $? \mathbf{cons}_i(x) \equiv eq_{struct[@T_1, \dots, @T_n]}(x, \mathbf{cons}_i(\mathbf{sel}_{i,1}(x), \dots, \mathbf{sel}_{i,n_i}(x)))$

Dabei ist  $\text{EQ}(\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle)$  für  
 $x_1, y_1: \text{struct}_{i,1}, \dots, x_n, y_n: \text{struct}_{i,n}$  definiert durch

- $\text{EQ}(\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle) = \text{eq}_{\text{struct}_{i,1}}(x_1, y_1)$ , falls  $n = 1$ , und sonst
- $\text{EQ}(\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle)$   
 $= \text{if}_{\text{bool}}\{\text{eq}_{\text{struct}_{i,1}}(x_1, y_1), \text{EQ}(\langle x_2, \dots, x_n \rangle, \langle y_2, \dots, y_n \rangle), \text{false}\}$

D.h., mit EQ wird die **Injektivität** von **Konstruktoren** (bzgl. eq)  
mittels boolescher *if*-Ausdrücke formuliert.

### **Beobachtung:**

Jedes *Axiom* ist eine (unbedingte) *Gleichung* !

### **Übung:**

Vergleichen Sie die Axiome einer Datentypdefinition mit den Regeln des Berechnungskalküls aus **Kapitel 5**. Welche Zusammenhänge erkennen Sie ?

## Schreibweisen in Beispielen und in *VeriFun*:

- $t = r$  anstatt  $\text{eq}_{struct}(t, r)$
- $b$  anstatt  $b \equiv \text{true}$

### Achtung – Verwechslungsgefahr:

- $=$  ist die Gleichheit der *Metasprache*, d.h.  $=$  wird verwendet, wenn wir über die Vorlesungsinhalte sprechen (z.B. “... falls  $n = 1$ ”).
- In *Beispielen* und in *VeriFun* steht  $=$  jedoch für  $\text{eq}_{struct}$  !!!
- $\equiv$  ist die Gleichheit in der *Prädikatenlogik* 1. Stufe, also ein **Prädikatensymbol**.
- Dagegen ist  $\text{eq}_{struct}$  ein **Funktionssymbol** !!!
- Die **Bedeutung** von  $\text{eq}_{struct}$  wird durch Gleichungen, wie z.B.

$$(1) \quad (a) \quad \text{eq}_{nat}(0, 0) \equiv \text{true}$$

$$(2) \quad (b) \quad \forall x, y: \text{nat} \quad \text{eq}_{nat}(\text{succ}(x), \text{succ}(y)) \equiv \text{eq}_{nat}(x, y)$$

$$(3) \quad (a1) \quad \forall x: \text{nat} \quad \text{eq}_{nat}(0, \text{succ}(x)) \equiv \text{false}$$

$$(a2) \quad \forall x: \text{nat} \quad \text{eq}_{nat}(\text{succ}(x), 0) \equiv \text{false}$$

für  $\text{eq}_{nat}$  festgelegt.

## Frage: Wozu brauchen wir überhaupt eq ???

### Beispiel 1 (*Warum eq ?*)

- $dbl : nat \rightarrow nat$  bezeichne Funktion, die natürliche Zahlen verdoppelt.
- $half : nat \rightarrow nat$  bezeichne Funktion, die natürliche Zahlen (nach unten gerundet) halbiert.
- Beziehung zwischen  $half$  und  $dbl$  – Schreibweise Prädikatenlogik:

$$\forall x:nat \ half(dbl(x)) \equiv x$$

- Beziehung zwischen  $half$  und  $dbl$  – Schreibweise hier:

$$\forall x:nat \ eq_{nat}(half(dbl(x)), x) \equiv true.$$

- **Fazit:** Beide Schreibweisen möglich, also brauchen wir eq nicht.

*Jetzt weiter:*

- *EVEN* bezeichne Prädikat, das genau auf gerade natürliche Zahlen zutrifft.
- Weitere Beziehung zwischen *half* und *dbl* – Schreibweise Prädikatenlogik:

$$\forall x: \text{nat } \text{EVEN}(x) \rightarrow \text{dbl}(\text{half}(x)) \equiv x \quad (2)$$

- **Aber:** Formel (2) ist **keine Gleichung** !
- **Abhilfe:** Schreibe Implikation “ $\rightarrow$ ” als bedingten Ausdruck:

$$\forall x: \text{nat } \text{if}_{\text{bool}}(\text{EVEN}(x), \text{dbl}(\text{half}(x)) \equiv x, \text{true}) \equiv \text{true} \quad (3)$$

- **Aber:** Zwar ist Formel (3) eine Gleichung, aber **nicht syntaktisch korrekt:** *EVEN*(*x*) und *dbl*(*half*(*x*))  $\equiv$  *x* sind **atomare Formeln** – gefordert sind jedoch **boolsche Terme**.
- **Abhilfe:** Repräsentiere **Prädikate** *PR* als **Funktionen**  $pr : \dots \rightarrow \text{bool}$   
Hier *even* : *nat*  $\rightarrow$  *bool* und *eq<sub>nat</sub>* : *nat*  $\times$  *nat*  $\rightarrow$  *bool*.
- **Damit:** Anstatt Formel (2) hier **Gleichung**

$$\forall x: \text{nat } \text{if}_{\text{bool}}(\text{even}(x), \text{eq}_{\text{nat}}(\text{dbl}(\text{half}(x)), x), \text{true}) \equiv \text{true}$$

oder in “PrettyPrint”

$$\forall x: \text{nat } \text{if}\{\text{even}(x), \text{dbl}(\text{half}(x)) = x, \text{true}\} .$$

**Fazit:**

- Mit einer Datentypdefinition `structure struct <= ...` werden implizit neue Funktionssymbole

$$if_{struct} : bool \times struct \times struct \rightarrow struct \text{ , und}$$
$$eq_{struct} : struct \times struct \rightarrow bool$$

eingeführt.

- $if_{struct}$  und  $eq_{struct}$  werden benötigt um **universelle Formeln** als **Gleichungen** zu schreiben.

**Nächste Frage:**

Wozu brauchen wir die **Selektoren**  $sel_{i,h} : struct \rightarrow struct_{i,h}$  ???



## Beispiel 2 (Warum Selektoren ?)

- Wir definieren  $half : nat \rightarrow nat$  durch die Gleichung

$$\forall x:nat \quad half(x) \equiv \text{if} \{ eq(x, 0), \\ 0, \\ \text{if} \{ even(x), succ(half(pred(x))), half(pred(x)) \} \}. \quad (4)$$

- Die Bedingung “ $eq(x, 0)$ ” können wir direkt “durch Einsetzen” in die linke Gleichungsseite ( $\Rightarrow$  **Pattern Matching**) ausdrücken. Man erhält:

$$half(0) \equiv 0 \\ \forall y:nat \quad half(succ(y)) \equiv \text{if} \{ even(succ(y)), succ(half(y)), half(y) \} \quad (5)$$

- $eq$ -Bedingungen können hier direkt durch **Pattern Matching** kodiert werden:
  - $eq(x, 0) \equiv true \Rightarrow x \mapsto 0$ ,
  - $eq(x, 0) \equiv false \Rightarrow x \mapsto succ(y), pred(x) \mapsto y$

und damit sind **Selektoren überflüssig**.

- **Aber:** Andere Bedingungen können i.A. nicht durch **Pattern Matching** kodiert werden, z.B.  $even(x)$ .
- **Konsequenz:** Wir verwenden Schreibweise (4) anstatt (5) als **einheitliche Notation für bedingte Ausdrücke**. Damit **benötigen** wir dann **Selektoren**.

**Beispiel 3** (*Axiome aus Datentypdefinition*)

```
structure bool <= true, false
```

- Repräsentiert **Wahrheitswerte**.

- $AX_{bool}$  gegeben durch:

$$(4) \text{ (a) } \forall x, y: \text{bool } \text{if}_{bool}\{\text{true}, x, y\} \equiv x$$

$$\text{ (b) } \forall x, y: \text{bool } \text{if}_{bool}\{\text{false}, x, y\} \equiv y$$

- `bool` ist in *VeriFun* **vordefiniert** ( $\Rightarrow$  Ordner **Predefined**).
- **Sonderregelung:** `bool` ist Argumenttyp für *kein* Funktionssymbol; insbesondere damit auch kein  $\text{eq}_{bool}$  (und folglich auch kein `?true` und kein `?false`).
- **Grund:** Funktionen mit Ergebnistyp `bool` übernehmen die Rolle von Prädikaten.

**Beispiel 4** (*Axiome aus Datentypdefinition*)

```
structure nat <= 0, succ(pred : nat)
```

- Repräsentiert **natürliche Zahlen**.

- $AX_{\text{nat}}$  gegeben durch:

$$(1) \quad (a) \quad \text{eq}_{\text{nat}}(0, 0) \equiv \text{true}$$

$$(2) \quad (a) \quad \forall x:\text{nat} \text{ pred}(\text{succ}(x)) \equiv x$$

$$(b) \quad \forall x, y:\text{nat} \text{ eq}_{\text{nat}}(\text{succ}(x), \text{succ}(y)) \equiv \text{eq}_{\text{nat}}(x, y)$$

$$(3) \quad (a1) \quad \forall x:\text{nat} \text{ eq}_{\text{nat}}(0, \text{succ}(x)) \equiv \text{false}$$

$$(a2) \quad \forall x:\text{nat} \text{ eq}_{\text{nat}}(\text{succ}(x), 0) \equiv \text{false}$$

$$(4) \quad (a) \quad \forall x, y:\text{nat} \text{ if}_{\text{nat}}\{\text{true}, x, y\} \equiv x$$

$$(b) \quad \forall x, y:\text{nat} \text{ if}_{\text{nat}}\{\text{false}, x, y\} \equiv y$$

$$(5) \quad (a1) \quad \forall x:\text{nat} \text{ ?}0(x) \equiv \text{eq}_{\text{nat}}(x, 0)$$

$$(a2) \quad \forall x:\text{nat} \text{ ?succ}(x) \equiv \text{eq}_{\text{nat}}(x, \text{succ}(\text{pred}(x)))$$

**Beispiel 5** (*Axiome aus Datentypdefinition*)

```
structure list[@T] <=
  ∅, [infixr, 100] :: (hd:@T, tl:list[@T])
```

- Repräsentiert **endliche polymorphe Listen**.

- $AX_{list[@T]}$  gegeben durch:

- (1) (a)  $eq_{list[@T]}(\emptyset, \emptyset) \equiv \text{true}$
- (2) (a<sub>1</sub>)  $\forall n:@T, k:list[@T] \text{hd}(n :: k) \equiv n$   
 (a<sub>2</sub>)  $\forall n:@T, k:list[@T] \text{tl}(n :: k) \equiv k$   
 (b)  $\forall n_1, n_2:@T, k_1, k_2:list[@T] \text{eq}_{list[@T]}(n_1 :: k_1, n_2 :: k_2)$   
 $\equiv \text{if}_{\text{bool}}\{\text{eq}_{@T}(n_1, n_2), \text{eq}_{list[@T]}(k_1, k_2), \text{false}\}$
- (3) (a<sub>1</sub>)  $\forall n:@T, k:list[@T] \text{eq}_{list[@T]}(\emptyset, n :: k) \equiv \text{false}$   
 (a<sub>2</sub>)  $\forall n:@T, k:list[@T] \text{eq}_{list[@T]}(n :: k, \emptyset) \equiv \text{false}$
- (4) (a)  $\forall x, y:list[@T] \text{if}_{list[@T]}(\text{true}, x, y) \equiv x$   
 (b)  $\forall x, y:list[@T] \text{if}_{list[@T]}(\text{false}, x, y) \equiv y$
- (5) (a<sub>1</sub>)  $\forall k:list[@T] ?\emptyset(k) \equiv \text{eq}_{list[@T]}(k, \emptyset)$   
 (a<sub>2</sub>)  $\forall k:list[@T] ?::(k) \equiv \text{eq}_{list[@T]}(k, \text{hd}(k) :: \text{tl}(k))$

## Selektoren sind nur partiell definiert !

- Für jeden Term  $t$  gilt  $\text{pred}(\text{succ}(t)) \equiv t$  (siehe Axiom 2(a) aus Beispiel 4).
- **Also:** Selektor  $\text{pred}$  repräsentiert die Vorgängerfunktion auf  $\text{nat}$ .
- **Aber:** Kein Axiom für  $\text{pred}(0)$  !
- **Genauso:**  $\text{hd}(n :: k) \equiv n$  sowie  $\text{tl}(n :: k) \equiv k$  (siehe Axiome 2(a<sub>1</sub>, a<sub>2</sub>) aus Beispiel 5), aber keine Axiome für  $\text{hd}(\emptyset)$  und  $\text{tl}(\emptyset)$ .
- **Abhilfe** für  $\text{pred}$  und  $\text{tl}$  (und alle Selektoren *monomorpher* Datentypen): Willkürlich gewählter Wert, etwa  $\text{pred}(0) \equiv 0$  und  $\text{tl}(\emptyset) \equiv \emptyset$ .
- **Aber:** Wegen *Polymorphie* kann für  $\text{hd}(\emptyset)$  kein willkürlich gewählter Wert angegeben werden (denn dieser müßte vom Typ  $@T$  sein) !
- **Damit:**  $\text{hd}$  kann nur *partiell* definiert werden (also nur für *nicht-leere* Listen).
- **Konsequenz:**  $\text{sel}_{j,h}(\text{cons}_i(t_1, \dots, t_{n_i}))$  ist *immer* undefiniert, falls Selektor  $\text{sel}_{j,h}$  nicht zu Konstruktor  $\text{cons}_i$  gehört !

## 2 Axiome aus Prozedurdefinitionen

- Um *Beweise* über Prozeduren führen zu können, muß die Bedeutung von Prozeduren formal durch *Axiome* erfaßt werden.
- **Vorgehen für formale Definition:** Erzeuge aus einer Prozedurdefinition schematisch eine Gleichung  $AX_{proc}$ .
- *Konkret:* Gegeben

$$\text{function proc} (x_1 : \text{struct}_1, \dots, x_n : \text{struct}_n) : \text{struct} \leq R_{proc}$$

### Bedeutung informell:

- Mit einer Prozedurdefinition wird ein **neues Funktionssymbol**  $proc : \text{struct}_1 \times \dots \times \text{struct}_n \rightarrow \text{struct}$  definiert.
- **Absicht:** Definiere Funktionen durch Angabe einer Berechnungsvorschrift

### Definition 2 (Axiom einer Prozedurdefinition, $AX_{proc}$ )

Einer Prozedurdefinition für  $proc$  wird die Gleichung  $AX_{proc} =$

$$\forall x_1 : \text{struct}_1, \dots, x_n : \text{struct}_n \text{ proc}(x_1, \dots, x_n) \equiv R_{proc}$$

zugeordnet.

**Beispiel 6** (*Axiome aus Prozedurdefinitionen*)

```

function [infix1, 1] >(x:ℕ, y:ℕ): bool <=
  if ?0(x)
  then false
  else if ?0(y)
        then true
        else ¬(x) > ¬(y)
  end
end
end

```

- Berechnet “größer-als” Relation auf natürlichen Zahlen.
- in *VeriFun* vordefiniert ( $\Rightarrow$  Ordner *Predefined*)
- $AX_{>}$  gegeben durch:

$$\forall x, y: \text{nat } x > y \equiv$$

$$\text{if}_{\text{bool}} \{ ?0(x),$$

$$\quad \text{false},$$

$$\quad \text{if}_{\text{bool}} \{ ?0(y), \text{true}, \neg(x) > \neg(y) \} \}$$

**Beispiel 7** (*Axiome aus Prozedurdefinitionen*)

```
function dbl(x:nat):nat <=  
  if x = 0 then 0 else succ(succ(dbl(pred(x)))) end
```

- Berechnet Verdopplung auf natürlichen Zahlen.

- $AX_{dbl}$  gegeben durch:

$$\forall x:\text{nat } \text{dbl}(x) \equiv \text{if}_{\text{nat}}\{\text{eq}_{\text{nat}}(x, 0), 0, \text{succ}(\text{succ}(\text{dbl}(\text{pred}(x))))\}$$



**Beispiel 8** (*Axiome aus Prozedurdefinitionen*)

```

function half(x:nat):nat <=
  if x = 0
  then 0
  else if pred(x) = 0
        then 0
        else succ(half(pred(pred(x))))
  end
end

```

- Berechnet Halbieren natürlicher Zahlen, nach unten gerundet.

- $AX_{\text{half}}$  gegeben durch:

$$\forall x:\text{nat}. \text{half}(x) \equiv$$

$$\text{if}_{\text{nat}}\{\text{eq}_{\text{nat}}(x, 0),$$

$$0,$$

$$\text{if}_{\text{nat}}\{\text{eq}_{\text{nat}}(\text{pred}(x), 0), 0, \text{succ}(\text{half}(\text{pred}(\text{pred}(x))))\}$$

## Axiome aus Prozedurdefinitionen mit einseitigen Fallunterscheidungen

- Jedes Vorkommen von “ $\star$ ” im Prozedurrumpf wird im Axiom durch Prozeduraufruf (mit identischer Parameterliste) ersetzt

### Beispiel:

```
function last(k : list[@T]) : @T <=
  if ?∅(k)
    then  $\star$ 
    else if ?∅(tl(k)) then hd(k) else last(tl(k)) end_if
  end_if
```

$AX_{\text{last}}$  gegeben durch:

$$\forall k:\text{list}[@T]$$

$$\text{last}(k) \equiv \text{if}_{@T}\{?∅(k),$$

$$\quad \underline{\text{last}(k)},$$

$$\quad \text{if}_{@T}\{?∅(\text{tl}(k)), \text{hd}(k), \text{last}(\text{tl}(k))\}\}$$

**Beispiel:**

```

function log2(x : nat) : nat <=
if ?0(x)
  then ★
  else if ?0(¬(x))
    then 0
    else if even(x) then +(log2(half(x))) else ★ end_if
  end_if
end_if

```

$\forall x_{\log_2}$  gegeben durch:

$\forall x:\text{nat}$

$$\log_2(x) \equiv \text{if}_{\text{nat}} \{ ?0(x), \underbrace{\log_2(x)}, \text{if}_{\text{nat}} \{ ?0(\neg(x)), 0, \text{if}_{\text{nat}} \{ \text{even}(x), +(\log_2(\text{half}(x))), \underline{\log_2(x)} \} \} \}$$

### 3 Axiome aus Lemmadefinitionen

- Um bewiesene Lemmata in *Beweisen* verwenden zu können, müssen die Lemmata formal durch *Axiome* erfaßt werden.
- **Vorgehen für formale Definition:** Erzeuge aus einer Lemmadefinition schematisch eine Gleichung  $AX_{lem}$ .
- *Konkret:* Gegeben

$$\text{lemma } lem \leq \forall x_1 : \text{struct}_1, \dots, x_n : \text{struct}_n \text{ body}_{lem}$$

#### Bedeutung informell:

- Lemmata treffen Aussagen über Datentypen und Prozeduren.

#### **Definition 3** (*Axiom einer Lemmadefinition, $AX_{lem}$* )

Der Definition eines Lemmas wird die Gleichung  $AX_{lem} =$

$$\forall x_1 : \text{struct}_1, \dots, x_n : \text{struct}_n \text{ body}_{lem} \equiv \text{true}$$

zugeordnet.

**Beispiel:**

lemma insert\_keeps\_ordered <=

$\forall n:\text{nat}, k:\text{list}[\text{nat}] \text{ if}\{\text{ordered}(k), \text{ordered}(\text{insert}(n, k)), \text{true}\}$

$\mathcal{A}X_{\text{insert\_keeps\_ordered}}$  gegeben durch:

$\forall n:\text{nat}, k:\text{list}[\text{nat}]$

$\text{if}\{\text{ordered}(k), \text{ordered}(\text{insert}(n, k)), \text{true}\} \equiv \text{true}$

## 4 Zusammenfassung

- Um *Behauptungen* (= unbewiesene Lemmata) über ein  $\mathcal{L}$ -Programm  $P$  *beweisen* zu können, wird die Bedeutung der Datentyp- und Prozedurdefinitionen sowie der (bewiesenen) Lemmata von  $P$  formal durch eine Menge von *Axiomen* (= Gleichungen) erfaßt.
- Ausgehend von diesen Axiomen wird dann versucht, eine Behauptung *formal herzuleiten*.
- *Allgemeiner*: Wir versuchen  $\text{true}$  ausgehend von den Axiomen aus dem Zielterm  $\text{body}_{1\text{em}}$  einer *HPL*-Sequenz  $\langle \emptyset, \emptyset \Vdash \text{body}_{1\text{em}} \rangle$  herzuleiten.
- *Noch allgemeiner*: Wir versuchen ausgehend von den Axiomen aus einer *HPL*-Sequenz  $\langle H, IH \Vdash \text{goal} \rangle$  die *HPL*-Sequenz  $\langle H, IH \Vdash \text{true} \rangle$  herzuleiten.
- Diese Herleitungen werden durch einen *automatischen Beweiser*, den *Symbolic Evaluator* ( $\Rightarrow$  **Kapitel 10**), durchgeführt.