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 $AX_{\texttt{struc[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}$ von Gleichungen.
- *Konkret:* Gegeben Datentypdefinition

$$\begin{array}{l}\textbf{structure}\ \ \texttt{struc[@T}_1,\ldots,\texttt{@T}_n\texttt{]}\ \texttt{<=}\\ \quad \texttt{cons}_1\texttt{(sel}_{1,1}\texttt{: struc}_{1,1},\ldots,\texttt{sel}_{1,n_1}\texttt{: struc}_{1,n_1}\texttt{)},\\ \qquad\qquad\vdots\\ \quad \texttt{cons}_k\texttt{(sel}_{k,1}\texttt{: struc}_{k,1},\ldots,\texttt{sel}_{k,n_k}\texttt{: struc}_{k,n_k}\texttt{)}\end{array} \tag{1}$$

Bedeutung informell:

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

Jetzt formal:

Definition 1 *(Axiome einer Datentypdefinition, $AX_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}$)*
Einer Datentypdefinition für $\texttt{struct[@T}_1,\ldots,\texttt{@T}_n\texttt{]}$ *wird die Gleichungsmenge* $AX_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}$ *wie folgt zugeordnet:*

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

(a) $\texttt{eq}_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}\texttt{(cons}_i\texttt{, cons}_i\texttt{)} \equiv \texttt{true}$

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

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

(b) $\forall x_1, y_1{:}struct_{i,1},\ldots,x_{n_i}, y_{n_i}{:}struct_{i,n_i}$
$\texttt{eq}_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}\texttt{(cons}_i\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_{n_i}\texttt{),cons}_i\texttt{(}y_1\texttt{,}\ldots\texttt{,}y_{n_i}\texttt{))}$
$\equiv \mathsf{EQ}(\langle x_1,\ldots,x_{n_i}\rangle, \langle y_1,\ldots,y_{n_i}\rangle)$

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

(a) $\forall x_1{:}struct_{i,1},\ldots,x_{n_i}{:}struct_{i,n_i}, y_1{:}struct_{i',1},\ldots,y_{n_{i'}}{:}struct_{i',n_{i'}}$
$\texttt{eq}_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}\texttt{(cons}_i\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_{n_i}\texttt{),cons}_{i'}\texttt{(}y_1\texttt{,}\ldots\texttt{,}y_{n_{i'}}\texttt{))} \equiv \texttt{false}$

(4) (a) $\forall x, y{:}struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}.\ \texttt{if}_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}\{\texttt{true}, x, y\} \equiv x$
(b) $\forall x, y{:}struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}.\ \texttt{if}_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}\{\texttt{false}, x, y\} \equiv y$

(5) für alle $i \in \{1,\ldots,k\}$:

(a) $\forall x{:}struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}$
$\texttt{?cons}_i\texttt{(}x\texttt{)} \equiv \texttt{eq}_{struct\texttt{[@T}_1,\ldots,\texttt{@T}_n\texttt{]}}\texttt{(}x\texttt{,cons}_i\texttt{(}sel_{i,1}\texttt{(}x\texttt{)},\ldots,sel_{i,n_i}\texttt{(}x\texttt{)))}$

Dabei ist $\mathsf{EQ}(\langle x_1,\ldots,x_n\rangle, \langle y_1,\ldots,y_n\rangle)$ *für*
$x_1, y_1{:}struct_{i,1},\ldots,x_n, y_n{:}struct_{i,n}$ *definiert durch*

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

  D.h., mit EQ wird die **Injektivität** von **Konstruktoren** (bzgl. eq) mittels boolscher 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 $\texttt{eq}_{struct}\texttt{(}t\texttt{,}r\texttt{)}$
- b anstatt $b \equiv \texttt{true}$

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 $\texttt{eq}_{struct}$ **!!!**
- $\equiv$ ist die Gleichheit in der *Prädikatenlogik* 1. Stufe, also ein **Prädikatensymbol**.
- Dagegen ist $\texttt{eq}_{struct}$ ein **Funktionssymbol** !!!
- Die **Bedeutung** von $\texttt{eq}_{struct}$ wird durch Gleichungen, wie z.B.

  (1) (a) $\texttt{eq}_{nat}\texttt{(O,O)} \equiv \texttt{true}$
  (2) (b) $\forall x,y{:}nat\ \texttt{eq}_{nat}\texttt{(succ(}x\texttt{),succ(}y\texttt{))} \equiv \texttt{eq}_{nat}\texttt{(}x\texttt{,}y\texttt{)}$
  (3) (a1) $\forall x{:}nat\ \texttt{eq}_{nat}\texttt{(O,succ(}x\texttt{))} \equiv \texttt{false}$
  (a2) $\forall x{:}nat\ \texttt{eq}_{nat}\texttt{(succ(}x\texttt{),O)} \equiv \texttt{false}$

  für $\texttt{eq}_{nat}$ 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:

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

- **Aber:** Formel (2) ist **keine Gleichung !**
- **Abhilfe:** Schreibe Implikation "→" als bedingten Ausdruck:

$$\forall x{:}nat\ if_{bool}(EVEN(x), dbl(half(x)) \equiv x, true) \equiv true \qquad (3)$$

- **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** $pr : \ldots \rightarrow bool$ Hier $even : nat \rightarrow bool$ und $eq_{nat} : nat \times nat \rightarrow bool$.
- **Damit:** Anstatt Formel (2) hier **Gleichung**

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

oder in "PrettyPrint"

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

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 $half : nat \to nat$ durch die Gleichung

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

- Die Bedingung "$eq(x,0)$" können wir direkt "durch Einsetzen" in die linke Gleichungsseite (=> **Pattern Matching**) ausdrücken. Man erhält:

$$\begin{array}{ll} & half(0) \equiv 0 \\ \forall y{:}nat & half(succ(y)) \equiv if\{even(succ(y)), succ(half(y)), half(y)\} \end{array} \quad (5)$$

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

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

- `bool` ist in *VeriFun* **vordefiniert** (=> Ordner **Predefined**).
- **Sonderregelung:** `bool` ist Argumenttyp für *kein* Funktionssymbol; insbesondere damit auch kein $\texttt{eq}_{bool}$ (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**.
- $AX_{\texttt{nat}}$ gegeben durch:

(1) (a) $\texttt{eq}_{\texttt{nat}}(\texttt{0},\texttt{0}) \equiv \texttt{true}$
(2) (a) $\forall x{:}\texttt{nat}\ \texttt{pred}(\texttt{succ}(x)) \equiv x$
(b) $\forall x, y{:}\texttt{nat}\ \texttt{eq}_{nat}(\texttt{succ}(x),\texttt{succ}(y)) \equiv \texttt{eq}_{nat}(x,y)$
(3) (a1) $\forall x{:}\texttt{nat}\ \texttt{eq}_{nat}(\texttt{0},\texttt{succ}(x)) \equiv \texttt{false}$
(a2) $\forall x{:}\texttt{nat}\ \texttt{eq}_{nat}(\texttt{succ}(x),\texttt{0}) \equiv \texttt{false}$
(4) (a) $\forall x, y{:}\texttt{nat}\ \texttt{if}_{nat}\{\texttt{true}, x, y\} \equiv x$
(b) $\forall x, y{:}\texttt{nat}\ \texttt{if}_{nat}\{\texttt{false}, x, y\} \equiv y$
(5) (a1) $\forall x{:}\texttt{nat}\ \texttt{?0}(x) \equiv \texttt{eq}_{nat}(x,\texttt{0})$
(a2) $\forall x{:}\texttt{nat}\ \texttt{?succ}(x) \equiv \texttt{eq}_{nat}(x,\texttt{succ}(\texttt{pred}(x)))$

Beispiel 5 *(Axiome aus Datentypdefinition)*

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

- Repräsentiert **endliche polymorphe Listen**.
- $AX_{\texttt{list[@T]}}$ gegeben durch:

(1) (a) $\texttt{eq}_{\texttt{list[@T]}}(\texttt{ø},\texttt{ø}) \equiv \texttt{true}$
(2) (a_1) $\forall n{:}\texttt{@T}, k{:}\texttt{list[@T]}\ \texttt{hd}(n::k) \equiv n$
(a_2) $\forall n{:}\texttt{@T}, k{:}\texttt{list[@T]}\ \texttt{tl}(n::k) \equiv k$
(b) $\forall n_1, n_2{:}\texttt{@T}, k_1, k_2{:}\texttt{list[@T]}\ \texttt{eq}_{\texttt{list[@T]}}(n_1::k_1, n_2::k_2)$
$\equiv \texttt{if}_{\texttt{bool}}\{\texttt{eq}_{\texttt{@T}}(n_1,n_2), \texttt{eq}_{\texttt{list[@T]}}(k_1,k_2), \texttt{false}\}$
(3) (a_1) $\forall n{:}\texttt{@T}, k{:}\texttt{list[@T]}\ \texttt{eq}_{\texttt{list[@T]}}(\texttt{ø}, n::k) \equiv \texttt{false}$
(a_2) $\forall n{:}\texttt{@T}, k{:}\texttt{list[@T]}\ \texttt{eq}_{\texttt{list[@T]}}(n::k, \texttt{ø}) \equiv \texttt{false}$
(4) (a) $\forall x, y{:}\texttt{list[@T]}\ \texttt{if}_{\texttt{list[@T]}}\{\texttt{true}, x, y\} \equiv x$
(b) $\forall x, y{:}\texttt{list[@T]}\ \texttt{if}_{\texttt{list[@T]}}\{\texttt{false}, x, y\} \equiv y$
(5) (a1) $\forall k{:}\texttt{list[@T]}\ \texttt{?ø}(k) \equiv \texttt{eq}_{\texttt{list[@T]}}(k, \texttt{ø})$
(a2) $\forall k{:}\texttt{list[@T]}\ \texttt{?::}(k) \equiv \texttt{eq}_{\texttt{list[@T]}}(k, \texttt{hd}(k)::\texttt{tl}(k))$

Selektoren sind nur partiell definiert !

- Für jeden Term `t` gilt `pred(succ(t))`$\equiv$ `t` (siehe Axiom 2(a) aus Beispiel 4).
- **Also:** Selektor `pred` repräsentiert die Vorgängerfunktion auf `nat`.
- **Aber:** Kein Axiom für `pred(0)` **!**
- **Genauso**: `hd(n::k)`$\equiv$ `n` sowie `tl(n::k)`$\equiv$ `k` (siehe Axiome 2(a_1,a_2) aus Beispiel 5), aber keine Axiome für `hd(ø)` und `tl(ø)`.
- **Abhilfe** für `pred` und `tl` (und alle Selektoren *monomorpher* Datentypen): Willkürlich gewählter Wert, etwa `pred(0)`$\equiv$ `0` und `tl(ø)`$\equiv$`ø`.
- **Aber:** Wegen *Polymorphie* kann für `hd(ø)` kein willkürlich gewählter Wert angegeben werden (denn dieser müßte vom Typ `@T` sein) **!**
- **Damit:** `hd` kann nur *partiell* definiert werden (also nur für *nicht-leere* Listen).
- **Konsequenz:** $\texttt{sel}_{j,h}(\texttt{cons}_i(t_1,\ldots,t_{n_i}))$ ist *immer* undefiniert, falls Selektor $\texttt{sel}_{j,h}$ nicht zu Konstruktor $\texttt{cons}_i$ 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

  function $\texttt{proc(x}_1\texttt{:struct}_1,\ldots,\texttt{x}_n\texttt{:struct}_n\texttt{):struct} <= R_{proc}$

Bedeutung informell:

- Mit einer Prozedurdefinition wird ein **neues Funktionssymbol** $proc : struct_1 \times \ldots \times struct_n \rightarrow struct$ definiert.
- **Absicht:** Definiere Funktionen durch Angabe einer Berechnungsvorschrift

Definition 2 *(Axiom einer Prozedurdefinition, AX_{proc})*
Einer Prozedurdefinition für `proc` *wird die Gleichung* $AX_{proc} =$

$$\forall x_1{:}struct_1,\ldots,x_n{:}struct_n\ proc(x_1,\ldots,x_n) \equiv R_{proc}$$

zugeordnet.

Beispiel 6 *(Axiome aus Prozedurdefinitionen)*

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

- Berechnet "größer-als" Relation auf natürlichen Zahlen.
- in *VeriFun* vordefiniert (=> Ordner *Predefined*)
- $AX_{>}$ gegeben durch:
  $\forall x, y$:`nat` $x > y \equiv$
  $\texttt{if}_{\texttt{bool}}\{\texttt{?0}(x),$
  `false`,
  $\texttt{if}_{\texttt{bool}}\{\texttt{?0}(y),\texttt{true},\texttt{⁻(x)} > \texttt{⁻(y)}\}\}$

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

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.
- $AX_{\texttt{half}}$ gegeben durch:
  $\forall x$:nat. half$(x) \equiv$
  $\texttt{if}_{\texttt{nat}}\{\texttt{eq}_{\texttt{nat}}(x,0),$
  $0,$
  $\texttt{if}_{\texttt{nat}}\{\texttt{eq}_{\texttt{nat}}(\texttt{pred}(x),0),0,\texttt{succ}(\texttt{half}(\texttt{pred}(\texttt{pred}(x))))\}\}$

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 ?ø(k)
 then ⋆
 else if ?ø(tl(k)) then hd(k) else last(tl(k)) end_if
end_if
```

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

$\forall$k:list[@T]
  last(k)$\equiv$ $\texttt{if}_{\texttt{@T}}\{$?ø(k),
    last(k),
    $\texttt{if}_{\texttt{@T}}\{$?ø(tl(k)),hd(k),last(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
```

$AX_{\texttt{log}_2}$ gegeben durch:

$\forall$x:nat
$\texttt{log}_2$(x)$\equiv$ $\texttt{if}_{\texttt{nat}}\{$?0(x),
  $\underline{\texttt{log}_2(\texttt{x})}$,
  $\texttt{if}_{\texttt{nat}}\{$?0($^-$(x)),
    $0,$
    $\texttt{if}_{\texttt{nat}}\{\texttt{even(x)},{}^+(\texttt{log}_2(\texttt{half(x)})),\underline{\texttt{log}_2(\texttt{x})}\}\}\}$

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

  $$\textbf{lemma}\ \texttt{lem}\ \texttt{<=}\ \forall \texttt{x}_1\texttt{:struct}_1, \ldots, \texttt{x}_n\texttt{:struct}_n\ body_{\texttt{lem}}$$

Bedeutung informell:

- Lemmata treffen Aussagen über Datentypen und Prozeduren.

Definition 3 *(Axiom einer Lemmadefinition, $AX_{\texttt{lem}}$)*
Der Definition eines Lemmas wird die Gleichung $AX_{\texttt{lem}} =$

$$\forall x_1{:}struct_1, \ldots, x_n{:}struct_n\ body_{\texttt{lem}} \equiv \texttt{true}$$

zugeordnet.

Beispiel:

```
lemma insert_keeps_ordered <=
∀n:nat,k:list[nat] if{ordered(k),ordered(insert(n,k)),true}
```

$AX_{\texttt{insert\_keeps\_ordered}}$ gegeben durch:

```
∀n:nat,k:list[nat]
    if{ordered(k),ordered(insert(n,k)),true} ≡ 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 `true` ausgehend von den Axiomen aus dem Zielterm $body_{\texttt{lem}}$ einer *HPL*-Sequenz $\langle \emptyset, \emptyset \Vdash body_{\texttt{lem}} \rangle$ herzuleiten.
- *Noch allgemeiner:* Wir versuchen ausgehend von den Axiomen aus einer *HPL*-Sequenz $\langle H, IH \Vdash goal \rangle$ die *HPL*-Sequenz $\langle H, IH \Vdash \texttt{true} \rangle$ herzuleiten.
- Diese Herleitungen werden durch einen *automatischen Beweiser*, den *Symbolic Evaluator* (=> **Kapitel 10**), durchgeführt.