Formale Grundlagen der Informatik 3 –

12. Semantik von $\mathcal{L}$-Lemmata

Christoph Walther
TU Darmstadt

1 Übersicht

- Durch Definition der *Semantik* von Lemmata legen wir fest, wann ein *Lemma wahr* ist
- Die *Semantik* von Lemmata wird *nur für terminierende* $\mathcal{L}$-Programme definiert
- **Damit:** Wir betrachten nur *totale* Korrektheitsaussagen
- **Grund 1:** *kompliziertere Semantik* bei partiellen Korrektheitsaussagen
- **Grund 2:** *höherer Interaktionsbedarf* bei Nachweis von partiellen Korrektheitsaussagen

Vorgehensweise:

- Wir betrachten 2 Teilsprachen $\mathcal{L}^{=}$ und $\mathcal{L}^{-}$ von $\mathcal{L}$ (mit $\mathcal{L}^{=} \subset \mathcal{L}^{-} \subset \mathcal{L}$)
- Zuerst Semantikdefinition für Teilsprache $\mathcal{L}^{=}$
- Danach Verallgemeinerung auf Teilsprache $\mathcal{L}^{-}$
- Schließlich Verallgemeinerung auf Sprache $\mathcal{L}$

2 Die Sprache $\mathcal{L}^{=}$

$\mathcal{L}^{=}$ ist definiert wie $\mathcal{L}$ jedoch

(1) *ohne polymorphe* Datentypen (*Typvariable* @XYZ verboten)

(2) *ohne partiell definierte* Prozeduren (*unbestimmte Ergebnisse* $\star$ verboten)

(3) mit *total definierten* Selektoren

Grund:

- Wegen (1) nur *monomorphe* Datentypen τ
  - *Damit gilt*: τ ist einzige Instanz von τ und kann durch die Menge $\mathcal{C}_\tau$ der *Konstruktorgrundterme* des Typs τ (mit $\mathcal{C}_\tau \neq \emptyset$) dargestellt werden
- Wegen (2) und (3) ***keine*** *Stuck-Terme*

Total definierte Selektoren:

- Aus der Definition eines monomorphen Datentyps τ kann uniform ein *Beispielterm* $\nabla_\tau \in \mathcal{C}_\tau$ bestimmt werden
  - *Beispiel*: $\nabla_{\texttt{nat}} = \texttt{0}$
  - *Beispiel*: $\nabla_{\texttt{list}} = \texttt{ø}$ für
    `structure list <= ø, [infixr,100] ::(hd : nat, tl : list)`
  - *Beispiel*: $\nabla_{\texttt{sexpr}} = \texttt{atom(0)}$ für
    ```
    structure sexpr <=
      atom(data : nat), nil, cons(car : sexpr, cdr : sexpr)
    ```
- Der *Berechnungskalkül* wird *erweitert* um die Regel

$$\frac{sel_{j,h}(cons_i(q_1, \ldots, q_{n_i}))}{\nabla_\tau} \text{, falls } q_1, \ldots, q_{n_i} \in \mathcal{C}(P) \text{ und } j \neq i \qquad (1)$$

- **Damit**: Selektoren sind auch bei Anwendung auf Konstruktoren, zu denen sie nicht gehören, definiert
  - *Beispiel*: $\texttt{pred(0)} \Rightarrow_P \texttt{0}$
  - *Beispiel*: $\texttt{hd(ø)} \Rightarrow_P \texttt{0}$, $\texttt{tl(ø)} \Rightarrow_P \texttt{ø}$
  - *Beispiel*: $* \texttt{data(nil)} \Rightarrow_P \texttt{0}$, $\texttt{data(cons(...))} \Rightarrow_P \texttt{0}$,
    - $\texttt{car(atom(...))} \Rightarrow_P \texttt{atom(0)}$, $\texttt{cdr(atom(...))} \Rightarrow_P \texttt{atom(0)}$,
    - $\texttt{car(nil)} \Rightarrow_P \texttt{atom(0)}$, $\texttt{cdr(nil)} \Rightarrow_P \texttt{atom(0)}$

- **Konsequenz:** Ergebnis Interpreter = *Konstruktorgrundterm* (oder undefiniert)

$$eval_P(t) := \begin{cases} t_{\Downarrow_P} & \text{, falls } t \Rightarrow^!_P r \text{ für ein } r \in \underline{\mathcal{C}(P)} \\ undefiniert & \text{, sonst.} \end{cases}$$

- **Für *terminierende* $\mathcal{L}^=$-Programme P gilt:**
  Ergebnis Interpreter *immer* Konstruktorgrundterm, d.h. für alle $t \in \mathcal{G}(P)$ gibt es ein $r \in \mathcal{C}(P)$ so daß

$$eval_P(t) = r$$

Definition 1 *(Standardalgebra $\mathcal{M}_P$, Standardinterpretation $\mathcal{I}_P$)*

Für ein terminierendes $\mathcal{L}^=$-Programm P ist die Standardalgebra $\mathcal{M}_P$ *von P definiert durch $\mathcal{M}_P := (\mathcal{C}(P), \mu)$ mit*

$$\mu_{\mathtt{f}}(\mathtt{q}_1, \ldots, \mathtt{q}_n) := eval_P(\mathtt{f}(\mathtt{q}_1, \ldots, \mathtt{q}_n))$$

für alle Funktionssymbole $\mathtt{f} : \tau_1 \times \ldots \times \tau_n \to \tau$ von P und alle $\mathtt{q}_i \in \mathcal{C}(P)_{\tau_i}$.

Eine Standardinterpretation $\mathcal{I}_P$ *für P ist ein Paar $(\mathcal{M}_P, \upsilon)$ wobei $\upsilon : \mathcal{V} \to \mathcal{C}(P)$ eine totale Funktion (=* Variablenbelegung*) mit $\upsilon(x) \in \mathcal{C}(P)_\tau$ für alle $x : \tau$ ist.*

- **Standardalgebra:**
  - Signatur Σ für Funktionssymbole gegeben durch $\mathcal{L}^{=}$-Programm P
    - `eq`, `if`, `case`, *Konstruktoren*, *Selektoren*, *Prozedurfunktionssymbole*
  - Trägermenge = Menge $\mathcal{C}(P)$ der Konstruktorgrundterme von P
  - Funktionen $\mu_{\mathtt{f}}$ definiert durch Interpreter $eval_P$
  - $\equiv$ ist einziges Prädikatensymbol (wird als Identität auf $\mathcal{C}(P)$ gedeutet)

Es gilt:

Satz 2 *(Standardalgebra und Axiome)*
Für ein terminierendes $\mathcal{L}^{=}$-Programm P gilt

$$\mathcal{M}_P \models AX_P \cup AX_{Lem_{verified}} \cup \mathcal{E}_P.$$

Beweis: *Übung.*

Mit Standardalgebra *jetzt:*

Definition 3 *(Theorie Th_P)*
Für ein terminierendes $\mathcal{L}^{=}$-Programm P ist die Theorie Th_P von P definiert als $Th(\mathcal{M}_P)$.

Also: Die Theorie Th_P von P enthält alle geschlossenen Formeln über den Funktionssymbolen von P, die in der Standardalgebra $\mathcal{M}_P$ wahr sind.

Jetzt: Mit der Theorie Th_P definieren wir die Semantik von Formeln über $\Sigma(P)$ und damit insbesondere die Semantik von $\mathcal{L}^=$-Lemmata.

Definition 4 (Semantik von $\mathcal{L}^=$-Formeln)

Eine Formel φ über der Signatur eines terminierenden $\mathcal{L}^=$-Programms P ist wahr *gdw.* $\varphi \in Th_P$.

Korollar 5 (Semantik von $\mathcal{L}^=$-Lemmata)

Ein Lemma "`lemma lem <=` $\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ *body" eines terminierenden $\mathcal{L}^=$-Programms P ist* wahr *gdw.* $[\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n \; body \equiv \texttt{true}] \in Th_P$.

Damit gilt: "`lemma lem` $<=$ $\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ \mathit{body}$" ist *wahr* gdw. der *Interpreter* $eval_P$ *jede Instanz* body' des Lemmarumpfs body, die durch Ersetzung aller allquantifizierten Variablen $x_i : \tau_i$ durch *Konstruktorgrundterme* $\mathtt{q}_i \in \mathcal{C}(P)_{\tau_i}$ entsteht, zu `true` *ausrechnet*, denn

	"`lemma lem` $<=$ $\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ \mathit{body}$" ist wahr	
gdw.	$[\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ \mathit{body} \equiv \mathtt{true}] \in Th_P$	, mit Korollar 5
gdw.	$\mathcal{M}_P \models \forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ \mathit{body} \equiv \mathtt{true}$	, mit Definition 3
gdw.	$\mathcal{M}_P \models \mathit{body}\,[x_1/\mathtt{q}_1, \ldots, x_n/\mathtt{q}_n] \equiv \mathtt{true}$	, für alle $\mathtt{q}_i \in \mathcal{C}(P)_{\tau_i}$
gdw.	$\mathcal{M}_P(\mathit{body}\,[x_1/\mathtt{q}_1, \ldots, x_n/\mathtt{q}_n]) = \mathcal{M}_P(\mathtt{true})$	, für alle $\mathtt{q}_i \in \mathcal{C}(P)_{\tau_i}$
gdw.	$eval_P(\mathit{body}\,[x_1/\mathtt{q}_1, \ldots, x_n/\mathtt{q}_n]) = eval_P(\mathtt{true})$	, für alle $\mathtt{q}_i \in \mathcal{C}(P)_{\tau_i}$
gdw.	$eval_P(\mathit{body}\,[x_1/\mathtt{q}_1, \ldots, x_n/\mathtt{q}_n]) = \mathtt{true}$	, für alle $\mathtt{q}_i \in \mathcal{C}(P)_{\tau_i}$

3 Einschub: Erzeugte Algebren

Definition 6 *(Teilsignatur)*

Seien Σ^1 *und* Σ^2 $\mathcal{S}$*-Signaturen. Dann ist* Σ^1 *eine* Teilsignatur *von* Σ^2*, kurz* $\Sigma^1 \subset \Sigma^2$*, gdw.* $\Sigma^1_{w,s} \subset \Sigma^2_{w,s}$ *für alle* $w \in \mathcal{S}^*$ *und alle* $s \in \mathcal{S}$*.* $\Sigma^2 \setminus \Sigma^1$ *ist definiert als die* $\mathcal{S}$*-Signatur* $\left(\Sigma^2_{w,s} \setminus \Sigma^1_{w,s}\right)_{w \in \mathcal{S}^*, s \in \mathcal{S}}$*.*

Definition 7 *(Erzeugte* Σ*-Algebra)*

Seien Σ^c *und* Σ $\mathcal{S}$*-Signaturen mit* $\Sigma^c \subset \Sigma$ *und sei* $A = (\mathcal{A}, \alpha)$ *eine* Σ*-Algebra. Dann ist* A *durch* Σ^c ***erzeugt*** *gdw. gilt:*

für alle $s \in \mathcal{S}$ *und für alle* $a \in \mathcal{A}_s$ *existiert ein* $q \in \mathcal{T}(\Sigma^c)_s$ *mit* $a = A(q)$*.*

Die Funktionssymbole aus Σ^c *werden die* ***Konstruktorfunktionssymbole*** *(kurz:* ***Konstruktoren****) von* A *genannt,* $\Sigma^d := \Sigma \setminus \Sigma^c$ *ist die Signatur der* ***definierten Funktionssymbole*** *von* A*.*

A *ist durch* Σ^c ***frei*** *erzeugt gdw. zusätzlich gilt:*

für alle $s \in \mathcal{S}$ *und für alle* $q_1, q_2 \in \mathcal{T}(\Sigma^c)_s$*:* $A(q_1) = A(q_2) \Rightarrow q_1 = q_2$*.* ■

***Anschauung*:**

- Wenn $A = (\mathcal{A}, \alpha)$ durch Σ^c erzeugt ist, so kann jedes Element a einer Trägermenge $\mathcal{A}_s$ durch **mindestens** einen Grunderm $q \in \mathcal{T}(\Sigma^c)_s$ (=> **Konstruktorgrundterm**) benannt werden. Anders gesagt, jedes $a \in \mathcal{A}_s$ besitzt mindestens einen "endlich langen Namen".
- Ist $A = (\mathcal{A}, \alpha)$ durch Σ^c **frei erzeugt**, so kann jedes Element a einer Trägermenge $\mathcal{A}_s$ durch **genau einen** Konstruktorgrundterm $q \in \mathcal{T}(\Sigma^c)_s$ benannt werden.

***Bedeutung*:**

- Alle Elemente der Trägermengen einer erzeugten Σ-Algebra A können **endlich repräsentiert** werden, nämlich durch Konstruktorgrundterme, und somit in einem Rechner abgespeichert werden.
- Ist A **frei erzeugt**, so existieren nur triviale Gleichungen zwischen Konstruktorgrundtermen ("frei" $=$ es gelten keine Gesetze).

Satz 8 *($\mathcal{M}_P$ ist frei erzeugt)*

Sei P ein terminierendes $\mathcal{L}^=$-Programm, Σ die Signatur von P und $\Sigma^c \subset \Sigma$ die Signatur der Konstruktorfunktionssymbole in P. Dann ist $\mathcal{M}_P$ durch Σ^c frei erzeugt.

Beweis: Übung.

Satz 9 *In einer frei erzeugten Σ-Algebra $A = (\mathcal{A}, \alpha)$ werden*

(1) *die Konstruktoren als injektive Funktionen und*

(2) *Terme mit voneinander verschiedenen führenden Konstruktoren durch verschiedene Trägerelemente gedeutet.*

Beweis (1) Sei A durch Σ^c erzeugt, sei $cons \in \Sigma^c_{s_1,\ldots,s_n,s}$ und sei $a_i, a'_i \in \mathcal{A}_{s_i}$ für jedes $i \in \{1, \ldots, n\}$ mit

$$\alpha_{cons}(a_1, \ldots, a_n) = \alpha_{cons}(a'_1, \ldots, a'_n).$$

Dann existieren $q_i, q'_i \in \mathcal{T}(\Sigma^c)_{s_i}$ mit $a_i = A(q_i)$ und $a'_i = A(q'_i)$ für jedes $i \in \{1, \ldots, n\}$, denn A ist durch Σ^c erzeugt. Also gilt

$$\alpha_{cons}(A(q_1), \ldots, A(q_n)) = \alpha_{cons}(A(q'_1), \ldots, A(q'_n))$$

und folglich

$$A(cons\ q_1\ \ldots\ q_n) = A(cons\ q'_1\ \ldots\ q'_n).$$

Da A *frei* erzeugt ist, folgt

$$cons\ q_1\ \ldots\ q_n = cons\ q'_1\ \ldots\ q'_n$$

und damit $q_i = q'_i$ für jedes $i \in \{1, \ldots, n\}$. Also gilt $A(q_i) = A(q'_i)$ und damit $a_i = a'_i$ für jedes $i \in \{1, \ldots, n\}$. $\square$

(2) Seien $cons \in \Sigma^c_{s_1,\ldots,s_n,s}$ und $cons' \in \Sigma^c_{s'_1,\ldots,s'_m,s'}$ mit $cons \neq cons'$, seien $t_i \in \mathcal{T}(\Sigma)_{s_i}$ und $t'_j \in \mathcal{T}(\Sigma)_{s_j}$ für jedes $i \in \{1,\ldots,n\}$ und jedes $j \in \{1,\ldots,m\}$. Angenommen, es gilt

$$A(cons\ t_1\ \ldots\ t_n) = A(cons'\ t'_1\ \ldots\ t'_m).$$

Damit gilt

$$\alpha_{cons}(A(t_1),\ldots,A(t_n)) = \alpha_{cons'}(A(t'_1),\ldots,A(t'_m))$$

und folglich

$$\alpha_{cons}(A(q_1),\ldots,A(q_n)) = \alpha_{cons'}(A(q'_1),\ldots,A(q'_m))$$

für gewisse $q_i \in \mathcal{T}(\Sigma^c)_{s_i}$ und $q'_j \in \mathcal{T}(\Sigma^c)_{s'_j}$ mit $A(t_i) = A(q_i)$ und $A(t'_j) = A(q'_j)$ für jedes $i \in \{1,\ldots,n\}$ und jedes $j \in \{1,\ldots,m\}$, denn A ist durch Σ^c erzeugt. Also gilt auch

$$A(cons\ q_1\ \ldots\ q_n) = A(cons'\ q'_1\ \ldots\ q'_m)$$

und folglich

$$cons\ q_1\ \ldots\ q_n\ =\ cons'\ q'_1\ \ldots\ q'_m\,,$$

denn A ist *frei* erzeugt.▼

Mit

$$cons\ q_1\ \ldots\ q_n\ \neq\ cons'\ q'_1\ \ldots\ q'_m\,,$$

muß daher auch

$$A(cons\ t_1\ \ldots\ t_n) \neq A(cons'\ t'_1\ \ldots\ t'_m).$$

gelten. □■

Beispiel 1 *Sei* $\mathcal{S} := \{nat\}\,, \Sigma^c_{nat} := \{O\}\,, \Sigma^c_{nat,nat} := \{succ\}\,, \Sigma_{nat,nat,nat} := \{plus\}$ *und* $PLUS = (\mathcal{A}, \alpha)$ *definiert durch*

(1) $\mathcal{A}_{nat} = \mathbb{N}$

(2) $\alpha_O := 0$, $\alpha_{succ}(n) := n + 1$

(3) $\alpha_{plus}(n, m) := n + m$.

Für $n \in \mathbb{N}$ *sei* $succ^{(n)} \in \mathcal{T}(\Sigma^c)$ *definiert durch* $succ^{(0)} := O$ *und* $succ^{(n+1)} := succ\ succ^{(n)}$. *Es gilt offenbar* $PLUS(succ^{(n)}) = n$ *und damit ist* $PLUS$ ***erzeugt*** *durch* Σ^c.

Mit $PLUS(succ^{(n_1)}) = PLUS(succ^{(n_2)}) \Rightarrow n_1 = n_2 \Rightarrow succ^{(n_1)} = succ^{(n_2)}$ *ist* $PLUS$ *sogar* ***frei erzeugt***.

Beispiel 2 *Sei* $\mathcal{S} := \{int\}, \Sigma^c_{int} := \{O\}, \Sigma^c_{int,int} := \{succ, pred\}, \Sigma_{int,int,int} := \{plus\}$ *und* $INT = (\mathcal{A}, \alpha)$ *definiert durch*

(1) $\mathcal{A}_{int} = \mathbb{Z}$

(2) $\alpha_O := 0, \alpha_{succ}(i) := i + 1, \alpha_{pred}(i) := i - 1,$

(3) $\alpha_{plus}(i, j) := i + j.$

Für $z \in \mathbb{Z}$ *sei* $cons^{(z)} \in \mathcal{T}(\Sigma)$ *definiert durch* $cons^{(0)} := O$, $cons^{(z)} := succ\ cons^{(z-1)}$, *falls* $z > 0$, *und* $cons^{(z)} := pred\ cons^{(z+1)}$, *falls* $z < 0$. *Es gilt offenbar* $INT(cons^{(z)}) = z$, *also ist* INT ***erzeugt*** *durch* Σ^c.

Mit $INT(succ\ pred\ O) = 0 = INT(O)$ *und* $succ\ pred\ O \neq O$ *ist* INT ***nicht frei erzeugt*** *durch* Σ^c. *Tatsächlich ist* INT *nicht frei erzeugbar: Ohne* O *kann* 0 *nicht repräsentiert werden, ohne* $succ$ *verliert man die positiven und ohne* $pred$ *die negativen ganzen Zahlen.*

Fazit: Mit Konstruktorfunktionssymbolen aus Σ^c werden die Elemente der Trägermengen einer Σ-Algebra repräsentiert (=> Darstellung von Daten eines Programms).

Mit den definierten Funktionssymbolen aus Σ^d werden Funktionen repräsentiert (=> Namen für die Funktionen, die von Prozeduren berechnet werden).

Satz 10

Für eine erzeugte Σ-Algebra $A = (\mathcal{A}, \alpha)$ sind alle Trägermengen $\mathcal{A}_s$ abzählbar.

Konsequenz: Es gibt keine erzeugten Σ-Algebren mit Trägermenge $2^{\mathbb{N}}$, $\mathbb{R}$ usw. Folglich gibt es auch keine Programmiersprachen mit einem Datentyp für die reellen Zahlen $\mathbb{R}$. Datentypen wie *real*, *float*, ... sind lediglich Approximationen von $\mathbb{R}$, d.h. diese bezeichnen *echte Teilmengen* von $\mathbb{R}$, die dann formal mittels erzeugter Algebren dargestellt werden können.

Zusammenhang zwischen A**-Variablenbelegungen** und **Substitutionen:**

Satz 11 *(Substitutionslemma)*

Sei $\mathfrak{a}$ eine A-Variablenbelegung für eine Σ-Algebra A, sei $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ eine Substitution und sei $t \in \mathcal{T}(\Sigma, \mathcal{V})$. Dann gilt

$$\mathfrak{a}(\sigma(t)) = \mathfrak{a}\left[x_1/\mathfrak{a}(t_1), \ldots, x_n/\mathfrak{a}(t_n)\right](t)$$

Beweis: *Übung.*

Satz 12 *(Konstruktorinduktion)*

In einer (durch $\Sigma^c \subset \Sigma$) erzeugten Σ-Algebra $A = (\mathcal{A}, \alpha)$ gilt das ***Induktionsprinzip****, d.h. für jede Formel $\phi[x] \in \mathcal{F}(\Sigma, \mathcal{V})$ mit $\mathcal{V}_f(\phi[x]) = \{x\}$ und $x \in \mathcal{V}_s$ gilt:*

Wenn

(1) $A \models \forall x_1{:}s_1, \ldots, \forall x_n{:}s_n\ \phi[f x_1 \ldots x_n]$
für alle $f \in \Sigma^c_{s_1,\ldots,s_n,s}$ mit $s_i \neq s$ für alle $i \in \{1, \ldots, n\}$ und

(2) $A \models \forall x_1{:}s_1, \ldots, \forall x_n{:}s_n\ \phi[x_{j_1}] \wedge \ldots \wedge \phi[x_{j_m}] \rightarrow \phi[f x_1 \ldots x_n]$
für alle $f \in \Sigma^c_{s_1,\ldots,s_n,s}$ mit $\{i \mid s_i = s\} = \{j_1, \ldots, j_m\} \neq \emptyset$,

so gilt auch

(3) $A \models \forall x{:}s\ \phi[x]$.

Beweis:

Angenommen, (3) ist falsch. Dann gibt es ein $a \in \mathcal{A}_s$ mit $A[x/a] \not\models \phi[x]$. Sei $\mathcal{T}_a := \{q \in \mathcal{T}(\Sigma^c)_s \mid A \not\models \phi[q]\}$. Da A durch Σ^c erzeugt ist, gilt $\mathcal{T}_a \neq \emptyset$ mit dem Substitutionslemma (Satz 11). Sei $q' \in \mathcal{T}_a$ ein Term minimaler Länge unter allen Termen von $\mathcal{T}_a$. Es gilt $q' = f q_1 \ldots q_n$ für ein $f \in \Sigma^c_{s_1,\ldots,s_n,s}$ und gewisse Terme $q_i \in \mathcal{T}(\Sigma^c)_{s_i}$.

Fall $s_i \neq s$ *für alle* $i \in \{1, \ldots, n\}$*: Mit (1) gilt* $A\,[x_1/A(q_1), \ldots, x_n/A(q_n)] \models \phi\,[f\,x_1 \ldots x_n]$ *und mit dem Substitutionslemma dann* $A \models \phi\,[f\,q_1 \ldots q_n]$*, also* $A \models \phi\,[q']$*.* ▼

Fall $s_i = s$ *für ein* $i \in \{1, \ldots, n\}$*: Mit (2) gilt* $A\,[x_1/A(q_1), \ldots, x_n/A(q_n)] \models (\phi\,[x_{j_1}] \wedge \ldots \wedge \phi\,[x_{j_m}] \rightarrow \phi\,[f\,x_1 \ldots x_n])$ *und mit dem Substitutionslemma dann* $A \models (\phi\,[q_{j_1}] \wedge \ldots \wedge \phi\,[q_{j_m}] \rightarrow \phi\,[f\,q_1 \ldots q_n])$*.*

Fall (i) $A \models \phi\,[q_{j_h}]$ *für alle* q_{j_h}*: Dann gilt* $A \models \phi\,[f\,q_1 \ldots q_n]$*, d.h.* $A \models \phi\,[q']$*.* ▼

Fall (ii) $A \not\models \phi\,[q_{j_h}]$ *für ein* q_{j_h}*: Dann gilt* $q_{j_h} \in \mathcal{T}_a$ *und folglich ist* q' *nicht minimal gewählt.* ▼

Bemerkung 1

- Das Induktionsprinzip von Satz 12 wird **Konstruktorinduktion** (über s) genannt, da es sich an der Termstruktur in $\mathcal{T}(\Sigma^c)_s$ orientiert.
- **Konstruktorinduktion** ist ein Sonderfall der **Noetherschen Induktion** (=> **Kapitel 6**) sowie der **strukturellen Induktion** (=> **Kapitel 7**).
- Satz 12 definiert *kein neues/weiteres* Induktionsprinzip (!): Der Satz klärt lediglich die semantische Grundlage der Induktion, nämlich *erzeugte* Algebren.

Beispiel 3 *Sei* $\mathcal{S} := \{nat\}, \Sigma^c_{nat} := \{O\}, \Sigma^c_{nat,nat} := \{succ\}, \Sigma_{nat,nat,nat} := \{plus\}$ *und* A *eine durch* Σ^c *erzeugte* Σ*-Algebra. Dann gilt mit Satz 12*

$$A \models \forall x{:}nat\ \phi[x]$$

für jede Formel $\phi[x] \in \mathcal{F}(\Sigma, \mathcal{V})$ *mit* $\mathcal{V}_f(\phi[x]) = \{x\}$ *und* $x \in \mathcal{V}_{nat}$*, falls*

(1) $A \models \phi[O]$ und

(2) $A \models \forall x{:}nat\ \phi[x] \rightarrow \phi[succ\ x]$

gilt. Konstruktorinduktion bzgl. $\mathcal{T}(\Sigma^c)_{nat}$ *entspricht also genau der "üblichen" Induktion auf natürlichen Zahlen, der* ***Peano Induktion****.*

Beispiel 4 *Sei* $\mathcal{S} := \{int\}, \Sigma^c_{int} := \{O\}, \Sigma^c_{int,int} := \{succ, pred\}, \Sigma_{int,int,int} := \{plus\}$ *und* A *eine durch* Σ^c *erzeugte* Σ*-Algebra. Dann gilt mit Satz 12*

$$A \models \forall x{:}int\ \phi[x]$$

für jede Formel $\phi[x] \in \mathcal{F}(\Sigma, \mathcal{V})$ *mit* $\mathcal{V}_f(\phi[x]) = \{x\}$ *und* $x \in \mathcal{V}_{int}$*, falls gilt:*

(1) $A \models \phi[O]$,

(2) $A \models \forall x{:}int\ \phi[x] \rightarrow \phi[succ\ x]$ und

(3) $A \models \forall x{:}int\ \phi[x] \rightarrow \phi[pred\ x]$.

Definition 13 *(Induktive Folgerung)*

Eine Formel $\phi \in \mathcal{F}_g(\Sigma, \mathcal{V})$ ***folgt bzgl.*** $\Sigma^c \subset \Sigma$ ***induktiv*** *aus einer Formelmenge* $\Phi \subset \mathcal{F}_g(\Sigma, \mathcal{V})$ *(kurz:* $\Phi \models_{Ind(\Sigma^c)} \phi$*) gdw.* $A \models_{Alg(\Sigma)} \phi$ *für* ***alle durch*** Σ^c ***erzeugten*** Σ*-Algebren* A *mit* $A \models_{Alg(\Sigma)} \Phi$.

Eine Formelmenge $\Psi \subset \mathcal{F}_g(\Sigma, \mathcal{V})$ ***folgt bzgl.*** Σ^c ***induktiv*** *aus einer Formelmenge* $\Phi \subset \mathcal{F}_g(\Sigma, \mathcal{V})$ *(kurz:* $\Phi \models_{Ind(\Sigma^c)} \Psi$*) gdw.* $\Phi \models_{Ind(\Sigma^c)} \psi$ *für* ***alle*** $\psi \in \Psi$.

Beispiel 5 *(Assoziativität von* `plus`*)*

- Programm P, AX_P und φ (Assoziativität von `plus`) wie in Beispiel 9 von **Kapitel 11**
- *Es gilt:* $AX_P \not\models_{\mathcal{F}(\Sigma,\mathcal{V})} \varphi$ (vgl. Beispiel 9 von **Kapitel 11**)
  - Assoziativität von `plus` folgt nicht aus AX_P
- *Aber:* $AX_P \models_{Ind(\Sigma^c)} \varphi$
  - Assoziativität von `plus` folgt *induktiv* aus AX_P
- Warum $A \not\models \varphi$ für Algebra A aus Beispiel 9 von **Kapitel 11**?
  - Algebra A ist *nicht erzeugt*, denn $A(q) \notin \left\{z + \frac{1}{2} | z \in \mathbb{Z}\right\}$ für alle Konstruktorgrundterme q. ■

Wie können *wahre* $\mathcal{L}^{=}$-Lemmata bewiesen werden ?

- Durch *Induktion*, denn die *Standardalgebra* $\mathcal{M}_P$ ist durch Σ^c *erzeugt* und in erzeugten Algebren gilt das Induktionsprinzip (=> Sätze 8 und 12).
- Mit Induktion erhalten wir neue *Beweisverpflichtungen* (= Induktionsformeln = Basis- und Schrittfälle), deren Gültigkeit (im Standardmodell $\mathcal{M}_P$) *hinreichend* für die Gültigkeit Originalbehauptung ist (=> Sätze 2 und 12).
- Kann eine Induktionsformel durch *symbolische Auswertung* bewiesen werden, so gilt $AX_P \cup AX_{Lem_{verified}} \cup \mathcal{E}_P \models \forall \ldots [\ldots \rightarrow b \equiv true]$ (*allgemeingültig*, vgl. Satz 15 in **Kapitel 11**) und damit insbesondere $\mathcal{M}_P \models b \equiv true$ (=> fertig!).
- Kann eine Induktionsformel *nicht* durch symbolische Auswertung *bewiesen* werden, so kann
  (a) zwar $AX_P \cup AX_{Lem_{verified}} \cup \mathcal{E}_P \models \forall \ldots [\ldots \rightarrow b \equiv true]$ gelten, jedoch der Nachweis von $b \not\vdash_{AX_P, Lem_{verified}, seq} true$ wegen *Unvollständigkeit* scheitern
  => dann *Benutzerinteraktion* erforderlich (*HPL*-Regeln *Use Lemma*, *Apply Equation*, *Case Analysis*, ...)
  (b) $AX_P \cup AX_{Lem_{verified}} \cup \mathcal{E}_P \not\models \forall \ldots [\ldots \rightarrow b \equiv true]$ gelten
  => dann *Benutzerinteraktion* erforderlich (*HPL*-Regel *Induction* oder aber *Hilfslemma* erfinden und beweisen, vgl. Fallstudie *InsertionSort*).

4 Die Sprache $\mathcal{L}^-$

$\mathcal{L}^-$ ist definiert wie $\mathcal{L}$ jedoch

- *ohne polymorphe* Datentypen (*Typvariable* @XYZ verboten)

Also jetzt erlaubt:

- *partiell definierte* Prozeduren (*unbestimmte Ergebnisse* $\star$ möglich)
- *partiell definierte* Selektoren

Konsequenz: Es gibt *Stuck-Terme*

Problem: Deutung von *Stuck-Termen*

Lösungsansatz:

- Aus einem terminierenden $\mathcal{L}^-$-Programm P^- gewinnen wir durch eine sogenannte *Vervollständigung* (irgend)ein $\mathcal{L}^=$-Programm $P^=$ indem wir
  - für jeden monomorphen Datentyp τ einen Beispielterm $\nabla_\tau \in \mathcal{C}(P^-)_\tau$ *beliebig* wählen
  - jedem Prozeduraufruf $\mathtt{p(q}_1\mathtt{,...,q}_n\mathtt{)}$ mit $\mathtt{q}_i \in \mathcal{C}(P^-)$ und $eval_P(except_{\mathtt{p}}[\mathtt{q}_1\mathtt{,...,q}_n]) = \mathtt{true}$ [1] ein *beliebiges Ergebnis* aus $\mathcal{C}(P^-)$ zuordnen
- Für ein terminierendes $\mathcal{L}^-$-Programm P^- definieren wir

$$\mathcal{P}(P^-) := \left\{\mathcal{L}^=\text{-Programm } P^= \mid P^= \text{ entsteht durch Vervollständigung aus } P^-\right\}$$

Damit:

Definition 14 *(Theorie Th_{P^-})*

Für ein terminierendes $\mathcal{L}^-$-Programm P^- ist die Theorie Th_{P^-} von P^- definiert als $\bigcap_{P^= \in \mathcal{P}(P^-)} Th_{P^=}$.

Also: Die Theorie Th_{P^-} von P^- enthält alle geschlossenen Formeln über den Funktionssymbolen von P^-, die in **jeder** Standardalgebra $\mathcal{M}_{P^=}$ wahr sind.

[1] Siehe **Kapitel 5**, Abschnitt 1.3.3.

Jetzt: Mit der Theorie Th_{P^-} definieren wir die Semantik von Formeln über $\Sigma(P^-)$ und damit insbesondere die Semantik von $\mathcal{L}^-$-Lemmata.

Definition 15 (Semantik von $\mathcal{L}^-$-Formeln)

Eine Formel φ über der Signatur eines terminierenden $\mathcal{L}^-$-Programms P^- ist wahr *gdw.* $\varphi \in Th_{P^-}$.

Korollar 16 (Semantik von $\mathcal{L}^-$-Lemmata)

Ein Lemma "`lemma lem <=` $\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ *body" eines terminierenden $\mathcal{L}^-$-Programms P^- ist* wahr *gdw.*

$$[\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n \; body \equiv \texttt{true}] \in Th_{P^-}$$

Für $\mathcal{L}^-$-Programme P^- gilt (vgl. Satz 14 in **Kapitel 11**)**:**

(1) $Th_{P^-} \neq \emptyset$

(2) $\varphi \notin Th_{P^-}$ oder $\neg\varphi \notin Th_{P^-}$ für alle geschlossenen Formeln φ
(Theorie ist *konsistent* / *widerspruchsfrei*)

(3) $\psi \notin Th_{P^-}$ **und** $\neg\psi \notin Th_{P^-}$ für einige geschlossenen Formeln ψ
(Theorie ist ***un****vollständig*) **!!!**

Beispiel für *Unvollständigkeit*:

- Sei $\psi := \texttt{pred(0)=0}$
- $P_1^=$ sei eine Vervollständigung von P^- mit $\mathcal{M}_{P_1^=}(\texttt{pred(0)}) = \texttt{1}$
- $P_2^=$ sei eine Vervollständigung von P^- mit $\mathcal{M}_{P_2^=}(\texttt{pred(0)}) = \texttt{0}$
- **Dann gilt:**
  - $\texttt{pred(0)=0} \notin Th_{P^-}$ denn $\texttt{pred(0)=0} \notin Th_{P_1^=}$
  - $\neg\,\texttt{pred(0)=0} \notin Th_{P^-}$ denn $\neg\,\texttt{pred(0)=0} \notin Th_{P_2^=}$.

Achtung:

“Unvollständigkeit” wird in unterschiedlicher Bedeutung verwendet:

- Ein *Kalkül* (der Prädikatenlogik) ist *unvollständig* gdw. es allgemeingültige Formeln gibt, die mittels des Kalküls *nicht bewiesen* werden können.
  - **Beispiel:** *Symbolische Auswertung.*
- Eine *Formelmenge* F ist *unvollständig* gdw. es eine Formel φ gibt, so daß weder $\varphi \in F$ noch $\neg\varphi \in F$ gilt.
  - **Beispiel:** Die Menge aller allgemeingültigen Formeln.
  - **Beispiel:** Die Theorie Th_{P^-} eines terminierenden $\mathcal{L}^-$-Programm P^-.

Konsequenz der *Unvollständigkeit* von Th_{P^-}:

- Es gibt $\mathcal{L}^-$-Lemmata φ, die
  - *weder* ***wahr*** ($=> \varphi \in Th_{P^-}$) *noch* ***falsch*** ($=> \neg\varphi \in Th_{P^-}$) sind.
- *Damit*: Es gibt $\mathcal{L}^-$-Lemmata, so daß diese
  - *weder* ***beweisbar*** *noch deren* ***Negat beweisbar*** sind.

Für $\mathcal{L}^=$-Lemmata gilt:

(1) Wenn der *Interpreter* $eval_{P^=}$ *jede Konstruktorgrundinstanz* des Lemmarumpfs zu `true` *ausrechnet*, so ist das Lemma *wahr*.

(2) Ist das Lemma *wahr*, so *rechnet* der *Interpreter* $eval_{P^=}$ *jede Konstruktorgrundinstanz* des Lemmarumpfs zu `true` aus.

Für $\mathcal{L}^-$-Lemmata gilt:

(1) Wenn der *Interpreter* $eval_{P^-}$ *jede Konstruktorgrundinstanz* des Lemmarumpfs zu `true` *ausrechnet*, so ist das Lemma *wahr*.

(2) **Ist das Lemma *wahr*, so *rechnet* der *Interpreter* $eval_{P^-}$ im allgemeinen *nicht jede* *Konstruktorgrundinstanz* des Lemmarumpfs zu `true` aus.**

Ursache: *Stuck-Terme*

- Grundterme, die *Stuck-Terme* enthalten, können mitunter *nicht* zu *Konstruktorgrundtermen ausgerechnet* werden (s. Abschnitt 2.4 in **Kapitel 5**)
- Damit insbesondere: *boolsche Grundterme*, die *Stuck-Terme* enthalten, können mitunter nicht zu `true` ausgerechnet werden
- **Konsequenz:** Enthält $body[\mathtt{x}_1/\mathtt{q}_1, \ldots, \mathtt{x}_n/\mathtt{q}_n]$ *Stuck-Terme*, so muß nicht notwendigerweise gelten:

$$eval_{P^-}(body[\mathtt{x}_1/\mathtt{q}_1, \ldots, \mathtt{x}_n/\mathtt{q}_n]) = \mathtt{true}$$

Beispiel 6

`lemma trivial <= ∀ x : nat pred(x) = pred(x)`

- **Ersetze** im Lemmarumpf `x:nat` durch `succ(0)`
- **Ergebnis**: Grundterm `pred(succ(0)) = pred(succ(0))`
- **Es gilt:** $eval_{P^-}$(`pred(succ(0))= pred(succ(0))`) = `true`

Aber:

- **Ersetze** im Lemmarumpf `x:nat` durch `0`
- **Ergebnis**: Grundterm `pred(0) = pred(0)`
- **Es gilt:** $eval_{P^-}$(`pred(0)= pred(0)`) = `pred(0)= pred(0)` $\neq$ `true !`
- **Grund:**
  - `pred(0)` ist ein *Stuck-Term*, damit ist auch
  - `pred(0)= pred(0)` ein *Stuck-Term* und folglich kann
  - `pred(0)= pred(0)` nicht weiter ausgerechnet werden
    (s. Abschnitt 2.4 in **Kapitel 5**)
- **Fazit:** Das $\mathcal{L}^-$-Lemma `trivial` ist zwar *wahr*, jedoch kann $eval_{P^-}$ nicht jede Konstruktorgrundinstanz des Lemmarumpfs zu `true` ausrechnen

5 Die Sprache $\mathcal{L}$

Jetzt alles erlaubt:

- *partiell definierte* Prozeduren (*unbestimmte Ergebnisse* $\star$ möglich)
- *partiell definierte* Selektoren
- *polymorphe* Datentypen (*Typvariable* @XYZ erlaubt)

Problem: Für *polymorphe* Datentypen τ kann τ ***nicht*** durch die Menge $\mathcal{C}_\tau$ der *Konstruktorgrundterme* des Typs τ dargestellt werden

Lösungsansatz:

- In einem Lemma

$$\texttt{lemma lem} <= \forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ \mathit{body}$$

  eines terminierenden $\mathcal{L}$-Programms P *ersetzen* wir die *Typvariable* in den Typen τ_i durch *beliebige monomorphe Typen* (außer solchen, die `bool` enthalten) und erhalten so die monomorphen Instanzen τ_i' der Typen τ_i
  - **Grund** *beliebige* monomorphe Typen:
    Typvariable sind *implizit allquantifiziert*
  - **Grund** beliebige *monomorphe* Typen:
    Diese können durch die Menge ihrer *Konstruktorgrundterme* repräsentiert werden

- **Ergebnis:** $\texttt{lemma lem}' <= \forall x_1{:}\tau_1', \ldots, x_n{:}\tau_n'\ \mathit{body}$

- In den *Prozeduren* und *Datentypen*, die in `lem` verwendet werden, werden die entsprechenden *Ersetzungen* der *Typvariablen* durchgeführt

- **Ergebnis:** $\mathcal{L}^-$-Programm P' (gegebenenfalls erweitert durch “neue” monomorphe Typen)

Damit:

Definition 17 (Semantik von $\mathcal{L}$-Formeln)

Eine Formel φ über der Signatur eines terminierenden *$\mathcal{L}$-Programms P ist* wahr *gdw.*

$$\varphi' \in Th_{P'}$$

für jede *monomorphe Instanz φ' der Formel und der entsprechenden monomorphen Instanz P' des $\mathcal{L}$-Programms P (monomorphe Instantiierung der Prozeduren und Datentypen, die in φ verwendet werden).*

Korollar 18 (Semantik von $\mathcal{L}$-Lemmata)

Ein Lemma "`lemma lem` $<=$ $\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ *body" eines* terminierenden *$\mathcal{L}$-Programms P ist* wahr *gdw.*

$$[\forall x_1{:}\tau'_1, \ldots, x_n{:}\tau'_n \; body \equiv \texttt{true}] \in Th_{P'}$$

für jede *monomorphe Instanz* `lem`$'$ *des Lemmas und der entsprechenden monomorphen Instanz P' des $\mathcal{L}$-Programms P (monomorphe Instantiierung der Prozeduren und Datentypen, die in* `lem` *verwendet werden).*

Damit Behauptung

Jedes Lemma, das in ***VeriFun*** *bewiesen* wurde, ist *wahr*.

Weiter gilt:

Für jedes *terminierende* $\mathcal{L}$-Programm P sind die Axiome AX_P sowie die Formeln aus $\mathcal{E}_P$ *wahr*.

Satz 19 *(Gültigkeit von Formeln aus $AX_P \cup \mathcal{E}_P$)*

Für jede *monomorphe Instanz φ' einer Formel $\varphi \in AX_P \cup \mathcal{E}_P$ eines* terminierenden *$\mathcal{L}$-Programms P und die entsprechende monomorphe Instanz P' des $\mathcal{L}$-Programms P (monomorphe Instantiierung der Prozeduren und Datentypen, die in φ verwendet werden), gilt*

$$\varphi' \in Th_{P'} .$$

Für *terminierende* $\mathcal{L}$-Programme P gilt damit:

- Ist ein Lemma

$$\texttt{lemma lem} <= \forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ body$$

in *VeriFun* bewiesen, so gilt

$$\boxed{eval_P(body[x_1/q_1, \ldots, x_n/q_n]) \neq \texttt{false}}$$

  - für alle *monomorphen Instanzen* $\tau'_1, \ldots, \tau'_n$ (die nicht `bool` enthalten) von $\tau_1, \ldots, \tau_n$ sowie
  - für alle *Konstruktorgrundterme* $q_1 \in \mathcal{C}(P)_{\tau'_1}, \ldots, q_n \in \mathcal{C}(P)_{\tau'_n}$.

Anschaulich:

- *Ersetze* die *Typvariable* in den Typen τ_i durch *beliebige monomorphe Typen* (außer solchen, die `bool` enthalten). *Ergebnis*: monomorphe Typen τ'_i
- *Ersetze* die Vorkommen der Variablen $x_i{:}\tau'_i$ im Lemmarumpf $body$ durch *beliebige Konstruktorgrundterme* aus $\mathcal{C}(P)_{\tau'_i}$.
- **Ergebnis** beider Ersetzungen: beliebiger *Grundterm* $body[x_1/q_1, \ldots, x_n/q_n]$
- **Es gilt:** $body[x_1/q_1, \ldots, x_n/q_n]$
  - kann durch $eval_P$ nie zu `false` *ausgerechnet* werden,
  - *vorausgesetzt* `lemma lem` wurde *verifiziert*.

Beispiel 7

```
lemma (k \ j) \ i = (k \ i) \ j <=
∀k : list[@ITEM], i, j : @ITEM (k \ j) \ i = (k \ i) \ j
```

*(aus **Kapitel 6**)*

- *Ersetze* `@ITEM` durch `list[nat]`. *Ergebnis*: `list[list[nat]]`
- *Ersetze* im Lemmarumpf
  - `k:list[list[nat]]` durch `(4 :: 1 :: ø) :: (3 :: 1 :: ø ) :: (4 :: 3 :: ø) :: ø`
  - `i:list[nat]` durch `4 :: 1 :: ø`
  - `j:list[nat]` durch `3 :: 1 :: ø`
- Ergebnis beider Ersetzungen: Grundterm $body' =$ `((4 :: 1 :: ø) :: (3 :: 1 :: ø ) :: (4 :: 3 :: ø) :: ø \ 3 :: 1 :: ø) \ 4 :: 1 :: ø = ((4 :: 1 :: ø) :: (3 :: 1 :: ø) :: (4 :: 3 :: ø ) :: ø \ 4 :: 1 :: ø) \ 3 :: 1 :: ø`
- **Es gilt:** $eval_P(body') =$ `true`

6 Widerlegungen von Lemmata

Für *terminierende* $\mathcal{L}$-Programme P gilt:

- Ein Lemma

  $$\texttt{lemma lem} <= \forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n\ body$$

  ist *widerlegt*, falls

  $$\boxed{eval_P(body[x_1/q_1, \ldots, x_n/q_n]) = \texttt{false}} \qquad (2)$$

  – für *einige* monomorphen Instanzen $\tau'_1, \ldots, \tau'_n$ (außer solchen, die `bool` enthalten) von $\tau_1, \ldots, \tau_n$ sowie

  – für *einige* Konstruktorgrundterme $q_1{:}\tau'_1, \ldots, q_n{:}\tau'_n$

- **Vorgehen:**
  – *Ersetze* die *Typvariable* in den Typen τ_i durch `nat`.
  Ergebnis: monomorphe Typen τ'_i
  – *Finde* Konstruktorgrundterme $q_1{:}\tau'_1, \ldots, q_n{:}\tau'_n$ so daß (2) gilt.
  – Aufgabe des *Disprovers*: Auffinden solcher Konstruktorgrundterme

Es gilt:

- Es gibt ein *vollständiges Verfahren*, daß das *Widerlegungsproblem löst*
- **Naiver Ansatz:** Erzeuge Schritte-für-Schritt alle Konstruktorgrundterme $q_1{:}\tau'_1, \ldots, q_n{:}\tau'_n$ und überprüfe nach jedem Schritt (2)
- **Offensichtlich:** Das Verfahren terminiert nicht, falls das Lemma wahr ist
- **Problem:** Auch bei nicht-naiven Verfahren können *sehr große Suchräume* entstehen
- **Daher:** Der *Disprover* wird durch Heuristiken gesteuert, um die Suchräume einzugrenzen
- **Konsequenz:** *Unvollständigkeit* des *Disprovers*, d.h. es gibt falsche Lemmata für die der *Disprover* keine Widerlegung findet