

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 ** 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}_τ 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:* $\nabla_{\text{nat}} = 0$
 - *Beispiel:* $\nabla_{\text{list}} = \emptyset$ für
`structure list <= \emptyset , [infixr, 100] :: (hd : nat, tl : list)`
 - *Beispiel:* $\nabla_{\text{sexpr}} = \text{atom}(0)$ für
`structure sexpr <= atom(data : nat), nil, cons(car : sexpr, cdr : sexpr)`
- Der *Berechnungskalkül* wird *erweitert* um die Regel

$$\frac{\text{sel}_{j,h}(\text{cons}_i(q_1, \dots, q_{n_i}))}{\nabla_\tau}, \text{ falls } q_1, \dots, q_{n_i} \in \mathcal{C}(P) \text{ und } j \neq i \quad (1)$$
- **Damit:** Selektoren sind auch bei Anwendung auf Konstruktoren, zu denen sie nicht gehören, definiert
 - *Beispiel:* $\text{pred}(0) \Rightarrow_P 0$
 - *Beispiel:* $\text{hd}(\emptyset) \Rightarrow_P 0, \text{tl}(\emptyset) \Rightarrow_P \emptyset$
 - *Beispiel:* $* \text{data}(\text{nil}) \Rightarrow_P 0, \text{data}(\text{cons}(\dots)) \Rightarrow_P 0,$
 $* \text{car}(\text{atom}(\dots)) \Rightarrow_P \text{atom}(0), \text{cdr}(\text{atom}(\dots)) \Rightarrow_P \text{atom}(0),$
 $* \text{car}(\text{nil}) \Rightarrow_P \text{atom}(0), \text{cdr}(\text{nil}) \Rightarrow_P \text{atom}(0)$

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

$$\text{eval}_P(t) := \begin{cases} t \downarrow_P & , \text{ falls } t \Rightarrow_P^! r \text{ für ein } r \in \mathcal{C}(P) \\ \text{undefiniert} & , \text{ sonst.} \end{cases}$$

- **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

$$\text{eval}_P(t) = r$$

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

$$\mu_{\mathbf{f}}(\mathbf{q}_1, \dots, \mathbf{q}_n) := \text{eval}_P(\mathbf{f}(\mathbf{q}_1, \dots, \mathbf{q}_n))$$

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

Eine Standardinterpretation \mathcal{I}_P für P ist ein Paar (\mathcal{M}_P, v) wobei $v : \mathcal{V} \rightarrow \mathcal{C}(P)$ eine totale Funktion (= Variablenbelegung) mit $v(x) \in \mathcal{C}(P)_\tau$ 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 $\mu_{\mathbf{f}}$ 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

$$\mathcal{M}_P \models AX_P \cup AX_{\text{Lem_verf}} \cup \mathcal{E}_P.$$

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 " $\text{lemma lem} \Leftarrow \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$ " eines terminierenden \mathcal{L}^- -Programms P ist wahr gdw. $[\forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body} \equiv \text{true}] \in Th_P$.

Damit gilt: " $\text{lemma lem} \Leftarrow \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$ " 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 $\mathbf{q}_i \in \mathcal{C}(P)_{\tau_i}$ entsteht, zu true ausrechnet, denn

| | | |
|------|---|---|
| | " $\text{lemma lem} \Leftarrow \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$ " ist wahr | |
| gdw. | $[\forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body} \equiv \text{true}] \in Th_P$ | , mit Korollar 5 |
| gdw. | $\mathcal{M}_P \models \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body} \equiv \text{true}$ | , mit Definition 3 |
| gdw. | $\mathcal{M}_P \models \text{body} [x_1/\mathbf{q}_1, \dots, x_n/\mathbf{q}_n] \equiv \text{true}$ | , für alle $\mathbf{q}_i \in \mathcal{C}(P)_{\tau_i}$ |
| gdw. | $\mathcal{M}_P(\text{body} [x_1/\mathbf{q}_1, \dots, x_n/\mathbf{q}_n]) = \mathcal{M}_P(\text{true})$ | , für alle $\mathbf{q}_i \in \mathcal{C}(P)_{\tau_i}$ |
| gdw. | $\text{eval}_P(\text{body} [x_1/\mathbf{q}_1, \dots, x_n/\mathbf{q}_n]) = \text{eval}_P(\text{true})$ | , für alle $\mathbf{q}_i \in \mathcal{C}(P)_{\tau_i}$ |
| gdw. | $\text{eval}_P(\text{body} [x_1/\mathbf{q}_1, \dots, x_n/\mathbf{q}_n]) = \text{true}$ | , für alle $\mathbf{q}_i \in \mathcal{C}(P)_{\tau_i}$ |

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. $\Sigma_{w,s}^1 \subset \Sigma_{w,s}^2$ für alle $w \in \mathcal{S}^*$ und alle $s \in \mathcal{S}$. $\Sigma^2 \setminus \Sigma^1$ definiert als die \mathcal{S} -Signatur $(\Sigma_{w,s}^2 \setminus \Sigma_{w,s}^1)_{w \in \mathcal{S}^*, s \in \mathcal{S}}$.

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 Grundterm $q \in \mathcal{T}(\Sigma^c)_s$ (\Rightarrow **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 Konstruktorfunktionen als injektive Funktionen und
- (2) Terme mit voneinander verschiedenen führenden Konstruktoren durch verschiedene Trägerelemente gedeutet.

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

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

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 $i \in \{1, \dots, n\}$, denn A ist durch Σ^c erzeugt. Also gilt

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

und folglich

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

Da A frei erzeugt ist, folgt

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

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

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

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

Damit gilt

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

und folglich

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

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 $i \in \{1, \dots, n\}$ und jedes $j \in \{1, \dots, m\}$, denn A ist durch Σ^c erzeugt. Also gilt auch

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

und folglich

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

denn A ist frei erzeugt. ▼

Mit

$$\text{cons } q_1 \dots q_n \neq \text{cons}' q'_1 \dots q'_m,$$

muß daher auch

$$A(\text{cons } t_1 \dots t_n) \neq A(\text{cons}' t'_1 \dots t'_m).$$

gelten. $\square \blacksquare$

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

- (1) $\mathcal{A}_{\text{nat}} = \mathbb{N}$
- (2) $\alpha_O := 0$, $\alpha_{\text{succ}}(n) := n + 1$
- (3) $\alpha_{\text{plus}}(n, m) := n + m$.

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

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

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

- (1) $\mathcal{A}_{\text{int}} = \mathbb{Z}$
- (2) $\alpha_O := 0$, $\alpha_{\text{succ}}(i) := i + 1$, $\alpha_{\text{pred}}(i) := i - 1$,
- (3) $\alpha_{\text{plus}}(i, j) := i + j$.

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

Mit $\text{INT}(\text{succ } \text{pred } O) = 0 = \text{INT}(O)$ und $\text{succ } \text{pred } O \neq O$ 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 (\Rightarrow Darstellung von Daten eines Programms).

Mit den definierten Funktionssymbolen aus Σ^d werden Funktionen repräsentiert (\Rightarrow 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 \mathbf{a} eine A-Variablenbelegung für eine Σ -Algebra A , sei $\sigma = \{x_1/t_1, \dots, x_n/t_n\}$ eine Substitution und sei $t \in \mathcal{T}(\Sigma, \mathcal{V})$. Dann gilt

$$\mathbf{a}(\sigma(t)) = \mathbf{a}[x_1/\mathbf{a}(t_1), \dots, x_n/\mathbf{a}(t_n)](t)$$

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) $A \models \forall x_1:s_1, \dots, \forall x_n:s_n \phi[f x_1 \dots x_n]$
für alle $f \in \Sigma_{s_1, \dots, s_n, s}^c$ mit $s_i \neq s$ für alle $i \in \{1, \dots, n\}$ und
- (2) $A \models \forall x_1:s_1, \dots, \forall x_n:s_n \phi[x_{j_1}] \wedge \dots \wedge \phi[x_{j_m}] \rightarrow \phi[f x_1 \dots x_n]$
für alle $f \in \Sigma_{s_1, \dots, s_n, s}^c$ mit $\{i \mid s_i = s\} = \{j_1, \dots, j_m\} \neq \emptyset$,

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 $q' = f q_1 \dots q_n$ für ein $f \in \Sigma_{s_1, \dots, s_n, s}^c$ und gewisse Terme $q_i \in \mathcal{T}(\Sigma^c)_{s_i}$.

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

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

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

Fall (ii) $A \not\models \phi[q_{j_i}]$ für ein q_{j_i} : Dann gilt $q_{j_i} \in \mathcal{T}_a$ 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** (\Rightarrow **Kapitel 6**) sowie der **strukturellen Induktion** (\Rightarrow **Kapitel 7**).
- Satz 12 definiert *kein neues/weiteres* Induktionsprinzip (!): Der Satz klärt lediglich die semantische Grundlage der Induktion, nämlich *erzeugte* Algebren.

Beispiel 3 Sei $\mathcal{S} := \{nat\}$, $\Sigma_{nat}^c := \{O\}$, $\Sigma_{nat,nat}^c := \{succ\}$, $\Sigma_{nat,nat,nat} := \{plus\}$ 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 $\mathcal{S} := \{int\}$, $\Sigma_{int}^c := \{O\}$, $\Sigma_{int,int}^c := \{succ, pred\}$, $\Sigma_{int,int,int} := \{plus\}$ 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 Formelmengemenge $\Phi \subset \mathcal{F}_g(\Sigma, \mathcal{V})$ (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 Formelmengemenge $\Psi \subset \mathcal{F}_g(\Sigma, \mathcal{V})$ **folgt bzgl.** Σ^c **induktiv** aus einer Formelmengemenge $\Phi \subset \mathcal{F}_g(\Sigma, \mathcal{V})$ (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 $A(q) \notin \{z + \frac{1}{2} | z \in \mathbb{Z}\}$ 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 (\Rightarrow 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 (\Rightarrow Sätze 2 und 12).
- Kann eine Induktionsformel durch *symbolische Auswertung* bewiesen werden, so gilt $AX_P \cup AX_{Lem_{verified}} \cup \mathcal{E}_P \models \forall \dots [\dots \rightarrow b \equiv true]$ (allgemeingültig, vgl. Satz 15 in **Kapitel 11**) und damit insbesondere $\mathcal{M}_P \models b \equiv true$ (\Rightarrow fertig!).
- Kann eine Induktionsformel *nicht* durch symbolische Auswertung bewiesen werden, so kann
 - (a) zwar $AX_P \cup AX_{Lem_{verified}} \cup \mathcal{E}_P \models \forall \dots [\dots \rightarrow b \equiv true]$ gelten, jedoch der Nachweis von $b \not\models_{AX_P, Lem_{verified}, seq} true$ wegen *Unvollständigkeit* scheitern \Rightarrow dann *Benutzerinteraktion* erforderlich (*HPL-Regeln Use Lemma, Apply Equation, Case Analysis, ...*)
 - (b) $AX_P \cup AX_{Lem_{verified}} \cup \mathcal{E}_P \not\models \forall \dots [\dots \rightarrow b \equiv true]$ gelten \Rightarrow 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 * 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 $p(q_1, \dots, q_n)$ mit $q_i \in \mathcal{C}(P^-)$ und $eval_P(except_p[q_1, \dots, q_n]) = \text{true}$ ¹ ein beliebiges Ergebnis aus $\mathcal{C}(P^-)$ zuordnen
- Für ein terminierendes \mathcal{L}^- -Programm P^- definieren wir

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

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 “ $\text{lemma lem} \Leftarrow \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$ ” eines terminierenden \mathcal{L}^- -Programms P^- ist wahr gdw.

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

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 **unvollständig**) !!!

Beispiel für Unvollständigkeit:

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

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** ($\Rightarrow \varphi \in Th_{P^-}$) noch **falsch** ($\Rightarrow \neg\varphi \in Th_{P^-}$) 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 $body[x_1/q_1, \dots, x_n/q_n]$ *Stuck-Terme*, so muß nicht notwendigerweise gelten:

$$eval_{P^-}(body[x_1/q_1, \dots, x_n/q_n]) = true$$

Beispiel 6

lemma trivial $\leq \forall x : nat \text{ pred}(x) = \text{pred}(x)$

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

Aber:

- **Ersetze** im Lemmarumpf $x : nat$ durch 0
- **Ergebnis:** Grundterm $\text{pred}(0) = \text{pred}(0)$
- **Es gilt:** $eval_{P^-}(\text{pred}(0) = \text{pred}(0)) = \text{pred}(0) = \text{pred}(0) \neq true !$
- **Grund:**
 - $\text{pred}(0)$ ist ein *Stuck-Term*, damit ist auch
 - $\text{pred}(0) = \text{pred}(0)$ ein *Stuck-Term* und folglich kann
 - $\text{pred}(0) = \text{pred}(0)$ 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* ★ möglich)
- *partiell definierte* Selektoren
- *polymorphe* Datentypen (*Typvariable* @XYZ erlaubt)

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

Lösungsansatz:

- In einem Lemma

$$\text{lemma lem} \leq \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$$

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:** $\text{lemma lem}' \leq \forall x_1:\tau'_1, \dots, x_n:\tau'_n \text{ body}$
- 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 “ $\text{lemma lem} \leq \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$ ” eines terminierenden \mathcal{L} -Programms P ist wahr gdw.

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

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

$$\text{lemma lem} \leq \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$$

in *VeriFun* bewiesen, so gilt

$$\text{eval}_P(\text{body}[x_1/q_1, \dots, x_n/q_n]) \neq \text{false}$$

- für alle *monomorphen Instanzen* τ'_1, \dots, τ'_n (die nicht `bool` enthalten) von τ_1, \dots, τ_n sowie
- für alle *Konstruktorgrundterme* $q_1 \in \mathcal{C}(P)_{\tau'_1}, \dots, q_n \in \mathcal{C}(P)_{\tau'_n}$.

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* $\text{body}[x_1/q_1, \dots, x_n/q_n]$
- **Es gilt:** $\text{body}[x_1/q_1, \dots, x_n/q_n]$
 - kann durch eval_P nie zu `false` *ausgerechnet* werden,
 - *vorausgesetzt* lemma `lem` wurde *verifiziert*.

Beispiel 7

$$\text{lemma } (k \setminus j) \setminus i = (k \setminus i) \setminus j \leq$$

$$\forall k : \text{list}[\text{@ITEM}], i, j : \text{@ITEM} \quad (k \setminus j) \setminus i = (k \setminus i) \setminus j$$

(aus **Kapitel 6**)

- Ersetze `@ITEM` durch `list[nat]`. *Ergebnis:* `list[list[nat]]`
- Ersetze im Lemmarumpf
 - `k:list[list[nat]]` durch `(4 :: 1 :: \emptyset) :: (3 :: 1 :: \emptyset) :: (4 :: 3 :: \emptyset) :: \emptyset`
 - `i:list[nat]` durch `4 :: 1 :: \emptyset`
 - `j:list[nat]` durch `3 :: 1 :: \emptyset`
- Ergebnis beider Ersetzungen: Grundterm $\text{body}' =$

$$((4 :: 1 :: \emptyset) :: (3 :: 1 :: \emptyset) :: (4 :: 3 :: \emptyset) :: \emptyset \setminus 3 :: 1 :: \emptyset) \setminus 4 :: 1 :: \emptyset = ((4 :: 1 :: \emptyset) :: (3 :: 1 :: \emptyset) :: (4 :: 3 :: \emptyset) :: \emptyset \setminus 4 :: 1 :: \emptyset) \setminus 3 :: 1 :: \emptyset$$
- **Es gilt:** $\text{eval}_P(\text{body}') = \text{true}$

6 Widerlegungen von Lemmata**Für terminierende \mathcal{L} -Programme P gilt:**

- Ein Lemma

$$\text{lemma lem} \leq \forall x_1:\tau_1, \dots, x_n:\tau_n \text{ body}$$

ist *widerlegt*, falls

$$\text{eval}_P(\text{body}[x_1/q_1, \dots, x_n/q_n]) = \text{false} \quad (2)$$

- für *einige* monomorphen Instanzen τ'_1, \dots, τ'_n (außer solchen, die `bool` enthalten) von τ_1, \dots, τ_n sowie
- für *einige* Konstruktorgrundterme $q_1:\tau'_1, \dots, q_n:\tau'_n$
- **Vorgehen:**
 - Ersetze die *Typvariable* in den Typen τ_i durch `nat`. *Ergebnis:* monomorphe Typen τ'_i
 - *Finde* Konstruktorgrundterme $q_1:\tau'_1, \dots, q_n:\tau'_n$ 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 $q_1:\tau'_1, \dots, q_n:\tau'_n$ 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