

Semantik: Einführung

Prof. Dr. David Sabel

LFE Theoretische Informatik



Ziele und Inhalte zunächst:

- Begriffe verstehen: Syntax, Semantik, kontextuelle Gleichheit
- Beispielhafte operationale Semantik für **sequentielle** Programmiersprache: λ -Kalkül

Danach: Betrachtung der Semantik von nebenläufigen Sprachen

- Betrachtung von **Kalkülen** als Modelle für Programmiersprachen
- Können als **Kernsprachen** solcher Programmiersprachen aufgefasst werden
- Abgespeckte Varianten der Programmiersprachen, dafür einfacher mathematisch handhabbar
- Kalkül: **Syntax** und **Semantik**

Syntax

- Legt fest, welche Programme (Ausdrücke) gebildet werden dürfen
- Welche **Konstrukte** stellt der Kalkül zu Verfügung?

Semantik

- Legt die **Bedeutung** der Programme fest
- Gebiet der **formalen Semantik** kennt verschiedene Ansätze

Axiomatische Semantik

- Beschreibung von Eigenschaften von Programmen mithilfe **logischer Axiome** und **Schlussregeln**
- **Herleitung** neuer Eigenschaften mit den Schlussregeln
- Prominentes Beispiel: Hoare-Logik, z.B.

Hoare-Tripel $\{P\}S\{Q\}$:

Vorbedingung P , Programm S , Nachbedingung Q

Schlussregel z.B.:

$$\text{Sequenz: } \frac{\{P\}S\{Q\}, \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

- Erfasst i.a. nur einige Eigenschaften, nicht alle, von Programmen

Denotationale Semantik

- **Abbildung** von Programmen in mathematische Räume durch **Semantische Funktion**
- Oft Verwendung von partiell geordneten Mengen (Domains)
- Im Nichtdeterministischen: Power-Domains
- Z.B. $\llbracket \cdot \rrbracket$ als semantische Funktion:

$$\llbracket \text{if } a \text{ then } b \text{ else } c \rrbracket = \begin{cases} \llbracket b \rrbracket, & \text{falls } \llbracket a \rrbracket = \textit{True} \\ \llbracket c \rrbracket, & \text{falls } \llbracket a \rrbracket = \textit{False} \\ \perp, & \text{sonst} \end{cases}$$

- Gilt i.a. als **mathematisch elegant**
- Schwierig bei vielen Konstrukten
- Sehr schwierig bei Nichtdeterminismus

Operationale Semantik

- definiert genau die **Auswertung/Ausführung** von Programmen
- definiert quasi einen Interpreter
- Verschiedene Formalismen:
 - Zustandsübergangssysteme
 - Abstrakte Maschinen
 - Ersetzungssysteme
- Unterscheidung in **small-step** und **big-step** Semantiken
- Wir verwenden operationale Semantiken

- Als einleitendes Beispiel betrachten wir den **Lambda-Kalkül**
- Modell für **sequentielle** Programmiersprachen
- Insbesondere für **funktionale** Programmiersprachen wie Haskell
- Von Alonzo Church in den 1930er Jahren eingeführt
- Der Lambda-Kalkül ist **Turing**-mächtig.

Syntax des Lambda-Kalküls

Expr	::=	V	Variable (unendliche Menge)
		$\lambda V.$ Expr	Abstraktion
		(Expr Expr)	Anwendung (Applikation)

- $\lambda x.s$: x ist in s **gebunden**, in Haskell: $\backslash x \rightarrow s$
- Abstraktionen sind **anonyme Funktionen**
 $id(x) = x$ in Lambda-Notation $\lambda x.x$
- $(s t)$ erlaubt die Anwendung von Funktionen auf Argumente
- s, t dürfen beliebige Ausdrücke sein
- deswegen **Higher-Order Lambda Kalkül**
- Bsp.: $id(id)$ kann in Lambda-Notation: $(\lambda x.x) (\lambda x.x)$

Gebundene und freie Variablen

- $FV(t)$: Freie Variablen von t
- $BV(t)$: Gebundene Variablen von t

$$FV(x) = x$$

$$FV(\lambda x.s) = FV(s) \setminus \{x\}$$

$$FV(s t) = FV(s) \cup FV(t)$$

$$BV(x) = \emptyset$$

$$BV(\lambda x.s) = BV(s) \cup \{x\}$$

$$BV(s t) = BV(s) \cup BV(t)$$

- $FV(t) = \emptyset \implies t$ geschlossen, t Programm, sonst t offen

Substitution

- $s[t/x]$ = Ausdruck der entsteht nach **Ersetzung** aller **freien Vorkommen** von x in s durch t
- Vermeidung von Namenskonflikten dabei: $x \notin BV(s)$

$$\begin{aligned}x[t/x] &= t \\y[t/x] &= y, \text{ falls } x \neq y \\(\lambda y.s)[t/x] &= \lambda y.(s[t/x]) \\(s_1 s_2)[t/x] &= (s_1[t/x] s_2[t/x])\end{aligned}$$

$$\text{Z.B. } (\lambda x.z x)[(\lambda y.y)/z] = (\lambda x.((\lambda y.y) x))$$

- **Kontext** = Ausdruck der an einer Position ein **Loch** $[\cdot]$ anstelle eines Unterausdrucks hat

Als Grammatik:

$$C ::= [\cdot] \mid \lambda V.C \mid (C \text{ Expr}) \mid (\text{Expr } C)$$

- In Kontexte kann man Ausdrücke einsetzen
- Kontext C , Ausdruck s : $C[s]$ ergibt Ausdruck, indem das Loch in C durch s ersetzt wird
- Beispiel: $C = ([\cdot] (\lambda x.x))$, dann: $C[\lambda y.y] = ((\lambda y.y) (\lambda x.x))$.
- Das Einsetzen darf/kann freie Variablen einfangen, z.B. $C = (\lambda x.[\cdot])$, dann $C[\lambda y.x] = (\lambda x.\lambda y.x)$.

Alpha-Umbenennungsschritt

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ falls } y \notin BV(\lambda x.s) \cup FV(\lambda x.s)$$

Alpha-Äquivalenz

$=_{\alpha}$ ist die **reflexiv-transitive Hülle** von $\xrightarrow{\alpha}$

- Wir betrachten α -äquivalente Ausdrücke als **gleich**.
- z.B. $\lambda x.x =_{\alpha} \lambda y.y$
- Distinct Variable Convention: Alle gebundenen Variablen sind verschieden und gebundene Variablen sind verschieden von freien.
- α -Umbenennungen ermöglichen, dass die DVC stets erfüllt werden kann.

Beta-Reduktion

$$(\beta) \quad (\lambda x.s) t \rightarrow s[t/x]$$

- Wenn $s_0 = (\lambda x.s) t \xrightarrow{\beta} s[t/x] = t_0$, dann sagt man s_0 reduziert unmittelbar zu t_0 .
- Für die Festlegung der operationalen Semantik, muss man noch definieren, wo die β -Reduktion angewendet wird
- Betrachte $((\lambda x.xx)((\lambda y.y)(\lambda z.z)))$.
- $((\lambda x.xx)((\lambda y.y)(\lambda z.z)))$ $\rightarrow ((\lambda y.y)(\lambda z.z)) ((\lambda y.y)(\lambda z.z))$
oder
- $((\lambda x.xx)$ $((\lambda y.y)(\lambda z.z)))$ $\rightarrow ((\lambda x.xx)(\lambda z.z))$.

Definition

Call-by-name **Reduktionskontexte** R :

$$R ::= [\cdot] \mid (R \textit{ Expr})$$

Wenn $s_0 \xrightarrow{\beta} t_0$, dann ist $R[s_0] \xrightarrow{\textit{name}} R[t_0]$
ein **call-by-name-Reduktionsschritt**

Beispiel

$$\begin{aligned} & ((\lambda x.(x x)) (\lambda y.y)) ((\lambda w.w) (\lambda z.(z z))) \\ \xrightarrow{\beta} & (x x)[(\lambda y.y)/x] ((\lambda w.w) (\lambda z.(z z))) \\ = & ((\lambda y.y) (\lambda y.y)) ((\lambda w.w) (\lambda z.(z z))) \\ & \text{hier ist } R = ([\cdot] ((\lambda w.w) \lambda z.(z z))) \end{aligned}$$

Call-by-Name Reduktion (2)

- Die call-by-name Reduktion ist **deterministisch**: Für jeden Ausdruck s gibt es höchstens einen Ausdruck t , so dass $s \xrightarrow{\text{name}} t$.
- Es gibt auch Ausdrücke für die keine Reduktion möglich ist:
- Reduktion stößt auf freie Variable: z.B. $(x (\lambda y.y))$
- Ausdruck ist ein **Wert**: Wert = Abstraktion
- $\xrightarrow{\text{name},+}$ = transitive Hülle von $\xrightarrow{\text{name}}$
- $\xrightarrow{\text{name},*}$ = reflexiv-transitive Hülle von $\xrightarrow{\text{name}}$

Definition

Ein Ausdruck s (call-by-name) *konvergiert* ($s \downarrow_{\text{name}}$) gdw.

\exists Abstraktion $v : s \xrightarrow{\text{name},*} v$.

Andernfalls **divergiert** s , Notation $s \uparrow_{\text{name}}$

Call-by-Name Reduktion (3)

- Haskell verwendet den call-by-name Lambda-Kalkül als semantische Grundlage,
- Implementierung verwenden call-by-need Variante:
Vermeidung von Doppelauswertungen
- Call-by-name (und auch call-by-need) sind optimal bzgl. Konvergenz:

Aussage

Sei s ein Lambda-Ausdruck und s kann mit beliebigen β -Reduktionen (an beliebigen Positionen) in eine Abstraktion v überführt werden. Dann gilt $s \downarrow_{name} v$.

Hauptidee: Argumentauswertung vor Einsetzung

Call-by-value Beta-Reduktion

$(\beta_{cbv}) \quad (\lambda x.s) v \rightarrow s[v/x]$, wobei v Abstraktion oder Variable

Definition

Call-by-value Reduktionsskontexte E :

$$E ::= [\cdot] \mid (E \text{ Expr}) \mid ((\lambda V. \text{Expr}) \text{ Expr})$$

Wenn $s_0 = (\lambda x.s) v \rightarrow s[v/x] \xrightarrow{\beta_{cbv}} s[v/x] = t_0$, dann ist $E[s_0] \xrightarrow{\text{value}} E[t_0]$ ein **call-by-value Reduktionsschritt**.

Call-by-Value Reduktion (2)

- Auch die call-by-value Reduktion ist deterministisch.

Definition

Ein Ausdruck s (call-by-value) konvergiert ($s \downarrow_{value}$), gdw.

\exists Abstraktion $v : s \xrightarrow{value,*} v$. Ansonsten (call-by-value) divergiert s ,

Notation: $s \uparrow_{value}$.

- es gilt: $s \downarrow_{value} \implies s \downarrow_{name}$.
- Die Umkehrung gilt nicht!
- Call-by-value Vorteil: Seiteneffekte können direkt eingebaut werden, da die Auswertungsreihenfolge fest liegt.
- Einige Programmiersprachen mit call-by-value Auswertung (strikte funktionale Programmiersprachen): ML (mit den Dialekten SML, OCaml), Scheme und Microsofts F#.

- $\Omega := (\lambda x.x x) (\lambda x.x x)$.
- $\Omega \xrightarrow{\text{name}} \Omega$. Daraus folgt: $\Omega \uparrow_{\text{name}}$
- $\Omega \xrightarrow{\text{value}} \Omega$. Daraus folgt: $\Omega \uparrow_{\text{value}}$.
- $t := ((\lambda x.(\lambda y.y)) \Omega)$.
- $t \xrightarrow{\text{name}} \lambda y.y$, d.h. $t \downarrow_{\text{name}}$.
- Da die call-by-value Auswertung jedoch zunächst das Argument Ω auswerten muss, gilt $t \uparrow_{\text{value}}$.

Bisher zwei Kalküle:

- Call-by-Name Lambda-Kalkül: Ausdrücke, $\xrightarrow{\textit{name}}$, $\downarrow_{\textit{name}}$
- Call-by-Value Lambda-Kalkül: Ausdrücke, $\xrightarrow{\textit{value}}$, $\downarrow_{\textit{value}}$

D.h. Syntax + Operationale Semantik.

Es fehlt:

- Begriff: Wann sind zwei Ausdrücke gleich
- D.h. insbesondere: Wann darf ein Compiler einen Ausdruck durch einen anderen ersetzen?

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt.
- Formaler: s und t sind gleich, wenn für alle C : gilt $C[s]$ und $C[t]$ verhalten sich gleich.
- Verhalten muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt.
- Formaler: s und t sind gleich, wenn für alle C : gilt $C[s]$ und $C[t]$ verhalten sich gleich.
- Verhalten muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt.
- Formaler: s und t sind gleich, wenn für alle C : gilt $C[s]$ und $C[t]$ verhalten sich gleich.
- Verhalten muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt.
- Formaler: s und t sind gleich, wenn für alle C : gilt $C[s]$ und $C[t]$ verhalten sich gleich.
- Verhalten muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung

Kontextuelle Approximation und Gleichheit

Call-by-Name Lambda-Kalkül:

- $s \leq_{c,name} t$ gdw. $\forall C : C[s] \downarrow_{name} \implies C[t] \downarrow_{name}$
- $s \sim_{c,name} t$ gdw. $s \leq_{c,name} t$ und $t \leq_{c,name} s$

Call-by-Value Lambda-Kalkül:

- $s \leq_{c,value} t$ gdw. $\forall C : C[s] \downarrow_{value} \implies C[t] \downarrow_{value}$
- $s \sim_{c,value} t$ gdw. $s \leq_{c,value} t$ und $t \leq_{c,value} s$

Gleichheit (4)

- $\sim_{c,name}$ und $\sim_{c,value}$ sind **Kongruenzen**
- **Kongruenz** = Äquivalenzrelation + kompatibel mit Kontexten, d.h. $s \sim t \implies C[s] \sim C[t]$.
- Gleichheit beweisen i.a. schwer, widerlegen i.a. einfach.

Gleichheit (4)

- $\sim_{c,name}$ und $\sim_{c,value}$ sind **Kongruenzen**
- **Kongruenz** = Äquivalenzrelation + kompatibel mit Kontexten, d.h. $s \sim t \implies C[s] \sim C[t]$.
- Gleichheit beweisen i.a. schwer, widerlegen i.a. einfach.

Beispiele für Gleichheiten:

- $(\beta) \subseteq \sim_{c,name}$
- $(\beta_{cbv}) \subseteq \sim_{c,value}$ aber $(\beta) \not\subseteq \sim_{c,value}$
- $\sim_{c,name} \not\subseteq \sim_{c,value}$ und $\sim_{c,value} \not\subseteq \sim_{c,name}$