

Semantik funktionaler Programme

- ❖ Die Bedeutung und Wirkungsweise der Programme einer Programmiersprache wird durch Angabe einer **Semantik** präzisiert.
- ❖ Die Semantik legt die Bedeutung einzelner Sprachelemente fest.
- ❖ Speziell bei funktionalen Programmiersprachen legt sie also die Bedeutung fest von
 - Funktionsdeklarationen
 - und Ausdrücken
 - arithmetische, Boolesche und Vergleichs-Operationen
 - bedingte Ausdrücke
 - Funktionsaufrufe

Operationale vs. funktionale Semantik

- ❖ Zwei mögliche Sichtweisen bei der Angabe einer Semantik
 - **operationale Semantik:**
Beschreibung der Abfolge der einzelnen Berechnungsschritte bei der Ausführung des Programms
 - **funktionale Semantik:**
Beschreibung der Funktion eines Programmes durch Festlegung des Ein- /Ausgabeverhaltens (extensionales bzw. beobachtbares Verhalten)
- ❖ Speziell für funktionale Programmiersprachen wählen wir:
 - für die operationale Semantik:
 - Ein Programm ist ein Termersetzungssystem.
 - für die funktionale Semantik:
 - Eine Interpretationsfunktion I ordnet
 - einer Funktionsdeklaration eine Funktion und
 - einem Ausdruck einen Wert

ZU.

Operationale Semantik: ein funktionales Programm als Termersetzungssystem

- ❖ ein Programm als Termersetzungssystem:
 - Programmteile definieren Termersetzungsregeln
 - ein Berechnungsschritt ist ein Termersetzungsschritt
 - das Ergebnis ist ein terminaler Term
(der nicht weiter abgeleitet werden kann).

- ❖ Die Termersetzungssemantik ist i.W. eine Formalisierung dessen, was wir bisher als „Auswerten von Ausdrücken“ bezeichnet haben.

Operationale Semantik: arithmetische, Boolesche und Vergleichsoperatoren

❖ Arithmetische Operatoren:

- Den Ausdruck $3+2$ betrachten wir als Kurzschreibweise des Terms $\text{add}(\text{succ}(\text{succ}(\text{succ}(\text{zero}))), \text{succ}(\text{succ}(\text{zero})))$
- Die arithmetischen Operatoren beschreiben Termersetzungsgesetze wie
 - $\text{add}(x, \text{succ}(y)) \rightarrow \text{succ}(\text{add}(x,y))$
 - $\text{add}(x, \text{zero}) \rightarrow x$
- Das Ergebnis der Termersetzung sind Terme in Normalform wie $\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{zero}))))$.
- Vgl. Kapitel Termersetzungssysteme

❖ Boolesche Operatoren und Vergleichsoperatoren:

- beschreiben Termersetzungsgesetze analog zu den arithmetischen Operatoren

Operationale Semantik: bedingte Ausdrücke

- ❖ der bedingte Ausdruck $B ? A_1 : A_2$ wird ausgewertet mit
 - folgender Metaregel:
 - Falls $B \rightarrow B'$ eine Termersetzungsregel ist,
dann ist auch $B ? A_1 : A_2 \rightarrow B' ? A_1 : A_2$ eine Regel.
 - und den folgenden Termersetzungsregeln:
 - $\text{true} ? A_1 : A_2 \rightarrow A_1$
 - $\text{false} ? A_1 : A_2 \rightarrow A_2$
- ❖ Diese Strukturierung in Metaregel und Regeln formalisiert folgende Auswertungsstrategie:
 - zuerst die Bedingung auswerten, bis entweder true oder false als Ergebnis erreicht ist;
 - erst danach genau einen der beiden Teilausdrücke A_1 oder A_2 auswerten.

Operationale Semantik:

Funktionsdeklaration und Funktionsaufruf

- ❖ Die Funktionsdeklaration $f(T_1 x_1, T_2 x_2, \dots, T_n x_n) \{ \text{return } A; \}$ wird zusammen mit dem Funktionsaufruf $f(A_1, A_2, \dots, A_n)$ über folgende Metaregeln ausgewertet:
 - Metaregel für die Parameterauswertung:
 - Falls $A_i \rightarrow A'_i$ für ein i eine Termersetzungsregel ist, dann ist auch $f(A_1, \dots, A_i, \dots, A_n) \rightarrow f(A_1, \dots, A'_i, \dots, A_n)$ eine Termersetzungsregel.
 - Metaregel für die Parametersubstitution:
 - Falls a_1, a_2, \dots, a_n terminale Terme sind, dann ist $f(a_1, a_2, \dots, a_n) \rightarrow A[a_1/x_1, a_2/x_2, \dots, a_n/x_n]$ eine Termersetzungsregel.
- ❖ Die Struktur der Metaregeln formalisiert die Auswertungsstrategie Call-by-Value.

Funktionale Semantik: eine Interpretationsfunktion für Ausdrücke

- ❖ Gegeben seien analog zu den Vorlesungsblöcken „Boolesche Ausdrücke“ und „Termersetzungssysteme“
 - eine Belegung β , die den in Ausdrücken frei vorkommenden Identifikatoren einen Wert zuordnet
 - eine Interpretationsfunktion I_β die jedem Ausdruck mit einem arithmetischen, Booleschen oder Vergleichs-Operator einen Wert zuordnet, z.B.

- $I_\beta[\text{true}] = L$

- $I_\beta[A_1 + A_2] = I_\beta[A_1] + I_\beta[A_2]$

Ein Zeichen im
Java-Programm

Die Addition auf z.B.
ganzen Zahlen

- ❖ Für die funktionale Semantik müssen wir nun noch angeben, wie I_β für bedingte Ausdrücke und für Funktionsaufrufe definiert sein soll.

Funktionale Semantik: Interpretation von bedingten Ausdrücken

❖ Die Interpretation bedingter Ausdrücke:

$$I_{\beta}[B \text{ ? } A_1 : A_2] = \begin{cases} I_{\beta}[A_1], & \text{falls } I_{\beta}[B] = L \\ I_{\beta}[A_2], & \text{falls } I_{\beta}[B] = 0 \\ \perp, & \text{falls } I_{\beta}[B] = \perp \end{cases}$$

Funktionale Semantik:

Interpretation von Funktionsaufrufen

❖ Zunächst setzen wir voraus:

- dass wir bereits wissen, wie Funktionsvereinbarungen zu interpretieren sind;
- wir bezeichnen mit $I_\beta[f]$ die Funktion, die dem Identifikator f mittels der Deklaration
$$T \ f (T_1 \ x_1, T_2 \ x_2, \dots, T_n \ x_n) \ \{ \text{return } A; \}$$
zugeordnet wird.

❖ Dann geben wir als Interpretation des Funktionsaufrufes

$f(A_1, A_2, \dots, A_n)$ an:

$$I_\beta[f(A_1, A_2, \dots, A_n)] = \begin{cases} \perp & \text{falls } I_\beta[A_i] = \perp \text{ für ein } i \\ I_\beta[f] (I_\beta[A_1], I_\beta[A_2], \dots, I_\beta[A_n]) & \text{sonst} \end{cases}$$

❖ Das heißt:

- wir wenden die Funktion $I_\beta[f]$ auf die Interpretationen der aktuellen Parameter an.
- Die Funktionsanwendung ist strikt.

Funktionale Semantik:

Interpretation nicht rekursiver Funktionsvereinbarungen

- ❖ Nun betrachten wir die Funktionsvereinbarung

$T \ f (T_1 \ x_1, T_2 \ x_2, \dots, T_n \ x_n) \ \{ \text{return } A; \}$

- ❖ Wir setzen dabei voraus, dass die Funktionsvereinbarung **nicht rekursiv** ist, d.h. dass A keinen Aufruf von f enthält.

- ❖ Die Interpretation der Funktionsvereinbarung liefert eine Funktion

– $I_\beta[f] : T_1 \times T_2 \times \dots \times T_n \rightarrow T$

- ❖ Wir definieren diese Funktion, indem wir für jede Kombination von Parameterwerten a_1, a_2, \dots, a_n angeben, welchen Wert $I_\beta[f](a_1, a_2, \dots, a_n)$ liefert (sog. punktweise Definition).

- ❖ $I_\beta[f](a_1, a_2, \dots, a_n)$ ist der Wert des Ausdruckes A , wobei sich die Belegung der formalen Parameter x_i zu a_i verändert:

Mit der neuen Belegung $\gamma(v) = \begin{cases} a_i, & \text{falls } v \text{ einer der Parameter } x_i \text{ ist} \\ \beta(v), & \text{sonst} \end{cases}$

gilt also: $I_\beta[f](a_1, a_2, \dots, a_n) = I_\gamma[A]$

Funktionale Semantik:

Interpretation rekursiver Funktionsvereinbarungen

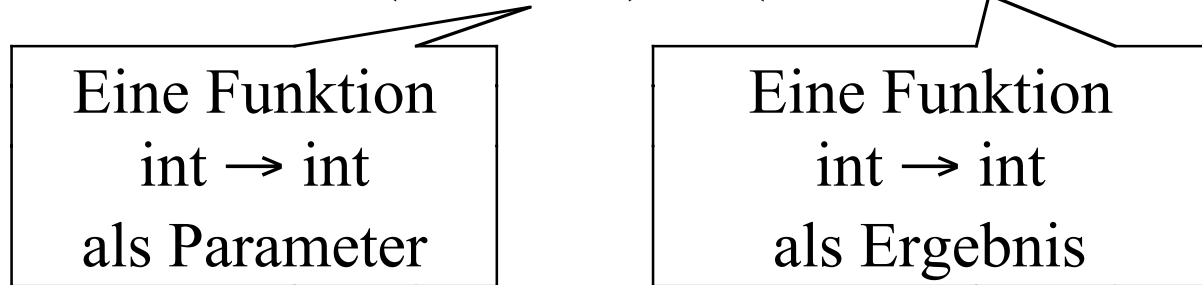
- ❖ Problem:
 - $I_{\beta}[f](a_1, a_2, \dots, a_n) = I_{\gamma}[A]$ ist keine gültige Definition, wenn f in A wieder auftritt, d.h. wenn f rekursiv ist.
- ❖ Für das Folgende wollen wir immer annehmen, dass A einen Aufruf von f enthält.
- ❖ Zur Vereinfachung betrachten wir nur noch folgende Funktionsdeklaration:
 - `int f (int n) { return A ;}`
- ❖ Diese Deklaration weist offensichtlich dem Identifikator f eine Funktion zu, die durch `int (int n) { return A ;}` beschrieben ist.
- ❖ `int (int n) { return A ;}` wird **Funktionsabstraktion** genannt.
- ❖ Was bedeutet aber `int (int n) { return A ;}` ?

Funktionsabstraktion und Funktionale

- ❖ Was bedeutet $\text{int } (\text{int } n) \{ \text{return } A ; \}$?
 - A enthält f , kann also als Ausdruck aufgefasst werden, der eine Funktion f als Parameter enthält
 - $\text{int } (\text{int } n) \{ \text{return } A ; \}$ ist dann eine Vorschrift, die in Abhängigkeit von einem Parameter f eine Funktion beschreibt.
- ❖ $\text{int } (\text{int } n) \{ \text{return } A ; \}$ ist also eine Abbildung, die in Abhängigkeit einer Funktion f wiederum eine Funktion liefert.
- ❖ Abbildungen, die Funktionen als Eingabe und Funktionen als Werte haben, nennen wir **Funktionale**.
- ❖ Wir kennen bereits Funktionale, z.B.
 - Differenzieren (ordnet einer Funktion ihre Ableitung zu)
 - Integrieren
 - Modellieren (ordnet einer Funktion der Wirklichkeit eine Funktion im Modell zu)

Funktionsabstraktion und Funktionale (Fortsetzung)

- ❖ Die Funktionsabstraktion $\text{int } (\text{int } n) \{ \text{return } A ; \}$ als Funktional:
 - Wir bezeichnen das Funktional, das durch diese Funktionsabstraktion beschrieben wird, im folgenden mit τ .
 - τ hat folgende Funktionalität: $\tau: (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$



- ❖ Erinnerung:
 - Wir suchen die Funktion $I_\beta[f]$, die durch die Vereinbarung $\text{int } f (\text{int } n) \{ \text{return } A ; \}$ dem Identifikator f zugewiesen wird.
 - Diese Funktion $I_\beta[f]$ wollen wir nun mit F abkürzen.
- ❖ Die entscheidende Idee:
 - Es muss gelten: $\tau(F) = F$
 - Das gesuchte F ist also ein sog. **Fixpunkt** des Funktionals τ .

Induktive Deutung rekursiver Funktionsdeklaration

- ❖ Entscheidende Idee:
 - Es muss gelten: $\tau(F) = F$
- ❖ Frage: welches F hat diese Eigenschaft, d.h. ist Fixpunkt von τ ?
- ❖ Pragmatische Vorgehensweise:
 - Am Anfang wissen wir gar nichts von F .
 - Wir starten deshalb mit einer Funktion F_0 , die an allen Stellen den Wert \perp liefert:
$$F_0(n) = \perp \text{ für alle } n \text{ aus } \text{int}$$
 - (Die Funktion, die an allen Stellen \perp liefert, wird oft mit Ω bezeichnet.)
 - Dann untersuchen wir, welche Information eigentlich durch τ geliefert wird:
 - Wir setzen F_0 in τ ein und nennen das Ergebnis F_1 :
 - $F_1 = \tau(F_0)$
 - Diesen Schritt wiederholen wir dann: $F_{i+1} = \tau(F_i)$

Induktive Deutung der Funktion Summe

❖ Beispiel:

```
int summe(int n) {return n == 0 ? 0 : summe(n-1) + n;}
```

❖ In diesem Beispiel wird τ durch die Funktionsabstraktion

```
int (int n) {return n == 0 ? 0 : f(n-1) + n;}
```

bestimmt.

❖ Wir starten mit $F_0 = \Omega$

❖ F_1 wird dann dadurch bestimmt, dass Ω in das Funktional eingesetzt wird, d.h. F_1 entspricht der Funktionsabstraktion:

```
- int (int n) {return n == 0 ? 0 :  $\Omega$ (n-1) + n;}
```

- und dies ist gleichwertig zu:

```
- int (int n) {return n == 0 ? 0 :  $\perp$ ;}
```

❖ F_1 ist also die Funktion, die für den Parameter 0 den Wert 0 liefert, und sonst überall \perp .

❖ F_1 ist aber immerhin schon etwas mehr definiert als Ω .

❖ Dies macht Mut zum Weitermachen.

Induktive Deutung der Funktion Summe (Fortsetzung)

❖ Beispiel:

```
int (int n) {return n == 0 ? 0 : f(n-1) + n;}
```

❖ F_1 : `int (int n) {return n == 0 ? 0 : \perp ;}`

❖ F_2 ergibt sich nun durch Einsetzen von F_1 in die Funktionsabstraktion.
Dies ergibt:

```
int (int n) {return n == 0 ? 0 :  $F_1$ (n-1) + n;}
```

❖ für $n = 0$:

$$- F_2(0) = 0$$

❖ für $n = 1$:

$$- F_2(1) = F_1(0) + 1 = 0 + 1 = 1$$

❖ für $n > 1$:

$$- F_2(n) = F_1(n-1) + n = \perp + n = \perp$$

❖ F_2 liefert also bereits für 0 und 1 die korrekten Werte und ist an allen anderen Stellen undefiniert.

Induktive Deutung der Funktion Summe (Fortsetzung)

❖ Beispiel:

```
int (int n) {return n == 0 ? 0 : f(n-1) + n;}
```

❖ Wir beschreiben die F_i durch eine Wertetabelle:

| $i \backslash n$ | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
|------------------|----|----|----|---|---|---|---|----|----|
| 0 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| 1 | ⊥ | ⊥ | ⊥ | 0 | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| 2 | ⊥ | ⊥ | ⊥ | 0 | 1 | ⊥ | ⊥ | ⊥ | ⊥ |
| 3 | ⊥ | ⊥ | ⊥ | 0 | 1 | 3 | ⊥ | ⊥ | ⊥ |
| 4 | ⊥ | ⊥ | ⊥ | 0 | 1 | 3 | 6 | ⊥ | ⊥ |
| 5 | ⊥ | ⊥ | ⊥ | 0 | 1 | 3 | 6 | 10 | ⊥ |
| 6 | ⊥ | ⊥ | ⊥ | 0 | 1 | 3 | 6 | 10 | 15 |

Induktive Deutung der Funktion Summe (Fortsetzung)

❖ Beobachtung:

- Mit jedem Schritt wird die Funktion an einer weiteren Stelle definiert.
- Wo die Funktion bereits definiert ist, ändert sich der Wert nicht mehr.
- Wo die Funktion bereits definiert ist, liefert sie den korrekten Wert $\text{summe}(n)$
- An den Stellen, an denen die Rekursion nicht terminiert, bleibt der Wert \perp .
- Die Funktion F_i berechnet die Funktionswerte korrekt, für die höchstens i Aufrufe von summe nötig sind.

Formalisierung der Beobachtung: Die Ordnung \sqsubseteq

- ❖ Die Ordnung \sqsubseteq auf Elementen:
 - Sind x und y zwei Elemente einer Trägermenge, dann gilt $x \sqsubseteq y$ genau dann, wenn $x = y$ oder $x = \perp$.
 - D.h. wenn x nicht undefiniert ist, dann ist es identisch mit y
- ❖ Die Ordnung \sqsubseteq auf Funktionen:
 - Sind f und g zwei Funktionen $f, g: X \rightarrow Y$, dann gilt $f \sqsubseteq g$ genau dann, wenn $f(x) \sqsubseteq g(x)$ für alle $x \in X$.
 - d.h. $f \sqsubseteq g$, wenn überall dort, wo f definiert ist, f und g dieselben Werte liefern.
 - Wir sagen: f ist schwächer (definiert) als g
- ❖ Unsere Beobachtung ist formalisierbar als Satz:
 - Für alle Funktionen f gilt: $f \sqsubseteq \tau(f)$, d.h. τ ist monoton.
- ❖ Für die F_i bedeutet das: $F_i \sqsubseteq F_{i+1}$

Induktive Deutung rekursiver Funktionen: Ergebnis

- ❖ **Satz:** Unter den gegebenen Voraussetzungen (insbesondere der Striktheit der Funktionsauswertung) gilt:
 - Die Funktionen F_i konvergieren gegen eine Funktion F .
 - Für F gilt: $\tau(F) = F$
 - F ist unter den Fixpunkten von τ der schwächste, d.h. für jeden weiteren Fixpunkt G mit $\tau(G) = G$ gilt: $F \sqsubseteq G$.
- ❖ Die Existenz des schwächsten Fixpunktes folgt aus einem Satz von Knaster-Tarski, die Übereinstimmung mit dem Grenzwert F aus einem Satz von Kleene.
- ❖ **Ergebnis:**
 - für die rekursive Funktion f mit `int f (int n) { return A; }` gilt:
 - $I_\beta[f] = F$
 - Dabei ist F der Grenzwert der Funktionen F_i , die dadurch entstehen, dass beginnend mit Ω die schon berechneten Funktionen in das Funktional τ eingesetzt werden:
 - $F_{i+1} = \tau(F_i)$
 - Es gilt: $\tau(F) = F$