Theorembeweiser und ihre Anwendungen

Prof. Dr.-Ing. Gregor Snelting
Dipl.-Inf. Univ. Daniel Wasserrab

Lehrstuhl Programmierparadigmen
IPD Snelting
Universität Karlsruhe (TH)

Teil V

Formale Semantik und Typsysteme

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), Syntax und Semantik

Syntax:
- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Semantik:
- Bedeutung der einzelnen Sprachkonstrukte
- Bei Programmiersprachen verschiedenste Darstellungsweisen:
  - informal (Beispiele, erläuternder Text etc.)
  - formal (Regelsysteme, Funktionen etc.)
- Angegeben im Sprachstandard (oft sehr vermischt mit Syntax)

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), Syntax und Semantik

Syntax:

- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Semantik:

- Bedeutung der einzelnen Sprachkonstrukte
- Bei Programmiersprachen verschiedenste Darstellungsweisen:
  - informal (Beispiele, erläuternder Text etc.)
  - formal (Regelsysteme, Funktionen etc.)
- Angegeben im Sprachstandard (oft sehr vermischt mit Syntax)

Was ist Semantik?

Zwei Konzepte bei Programmiersprachen (analog zu natürlicher Sprache), Syntax und Semantik

Syntax:

- Regeln für korrekte Anordnung von Sprachkonstrukten
- In Programmiersprachen meist durch Grammatik, vor allem in BNF (Backus-Naur-Form) gegeben
- Angegeben im Sprachstandard

Semantik:

- Bedeutung der einzelnen Sprachkonstrukte
- Bei Programmiersprachen verschiedenste Darstellungsweisen:
  - informal (Beispiele, erläuternder Text etc.)
  - formal (Regelsysteme, Funktionen etc.)
- Angegeben im Sprachstandard (oft sehr vermischt mit Syntax)

Wozu formale Semantik?

Weitreichende Einsatzbereiche:

- Spezifikation von Programmverhalten
  (z.B. Design/Verständnis von Programmiersprachen)
- Compilerbau und -optimierung (z.B. Korrektheitsanalyse)
- Programmanalyse (z.B. Language Based Security)
- Korrektheitsbeweise (z.B. Typsicherheit)
- Testen (z.B. Prototyp-Entwicklung)
- ...

Verschiedenste Methodik:

- "Pen and Paper"
- Theorembeweiser
- Model Checking
- Typsysteme
- ...

Wozu formale Semantik?

Weitreichende Einsatzbereiche:

- Spezifikation von Programmverhalten
  (z.B. Design/Verständnis von Programmiersprachen)
- Compilerbau und -optimierung (z.B. Korrektheitsanalyse)
- Programmanalyse (z.B. Language Based Security)
- Korrektheitsbeweise (z.B. Typsicherheit)
- Testen (z.B. Prototyp-Entwicklung)
- ...

Verschiedenste Methodik:

- "Pen and Paper"
- Theorembeweiser
- Model Checking
- Typsysteme
- ...

Wozu formale Semantik?

Weitreichende Einsatzbereiche:

- Spezifikation von Programmverhalten
  (z.B. Design/Verständnis von Programmiersprachen)
- Compilerbau und -optimierung (z.B. Korrektheitsanalyse)
- Programmanalyse (z.B. Language Based Security)
- Korrektheitsbeweise (z.B. Typsicherheit) ⟵
- Testen (z.B. Prototyp-Entwicklung)
- ...

Verschiedenste Methodik:

- "Pen and Paper"
- Theorembeweiser ⟵
- Model Checking
- Typsysteme
- ...

Übersicht formale Semantiken

verschiedene Arten von Semantiken:

operational: simuliert Zustandsübergänge auf abstrakter Maschine
nahe an tatsächlichem Programmverhalten

denotational: Programm Funktion, bildet Anfangs- auf Endzustand ab
sehr mathematisch, gern für Beweise verwendet
heute etwas aus der Mode

axiomatisch: auch bekannt als Hoare-Logik
Zusicherungen als Vor-/Nachbedingungen, Invarianten

Termersetzung: Programmfortschritt als Termersetzungsregeln

Game semantics: nahe an denotational, "Spiel" zwischen Programm und
System, aktuell viel Forschung in diesem Bereich

Im folgenden nur operationale Semantik

Übersicht formale Semantiken

verschiedene Arten von Semantiken:

operational: simuliert Zustandsübergänge auf abstrakter Maschine
nahe an tatsächlichem Programmverhalten

denotational: Programm Funktion, bildet Anfangs- auf Endzustand ab
sehr mathematisch, gern für Beweise verwendet
heute etwas aus der Mode

axiomatisch: auch bekannt als Hoare-Logik
Zusicherungen als Vor-/Nachbedingungen, Invarianten

Termersetzung: Programmfortschritt als Termersetzungsregeln

Game semantics: nahe an denotational, "Spiel" zwischen Programm und
System, aktuell viel Forschung in diesem Bereich

Im folgenden nur operationale Semantik

Übersicht formale Semantiken

verschiedene Arten von Semantiken:

operational: simuliert Zustandsübergänge auf abstrakter Maschine
nahe an tatsächlichem Programmverhalten

denotational: Programm Funktion, bildet Anfangs- auf Endzustand ab
sehr mathematisch, gern für Beweise verwendet
heute etwas aus der Mode

axiomatisch: auch bekannt als Hoare-Logik
Zusicherungen als Vor-/Nachbedingungen, Invarianten

Termersetzung: Programmfortschritt als Termersetzungsregeln

Game semantics: nahe an denotational, "Spiel" zwischen Programm und System, aktuell viel Forschung in diesem Bereich

Im folgenden nur operationale Semantik

Übersicht formale Semantiken

verschiedene Arten von Semantiken:

operational: simuliert Zustandsübergänge auf abstrakter Maschine
nahe an tatsächlichem Programmverhalten

denotational: Programm Funktion, bildet Anfangs- auf Endzustand ab
sehr mathematisch, gern für Beweise verwendet
heute etwas aus der Mode

axiomatisch: auch bekannt als Hoare-Logik
Zusicherungen als Vor-/Nachbedingungen, Invarianten

Termersetzung: Programmfortschritt als Termersetzungsregeln

Game semantics: nahe an denotational, "Spiel" zwischen Programm und System, aktuell viel Forschung in diesem Bereich

Im folgenden nur operationale Semantik

Übersicht formale Semantiken

verschiedene Arten von Semantiken:

operational:	simuliert Zustandsübergänge auf abstrakter Maschine nahe an tatsächlichem Programmverhalten
denotational:	Programm Funktion, bildet Anfangs- auf Endzustand ab sehr mathematisch, gern für Beweise verwendet heute etwas aus der Mode
axiomatisch:	auch bekannt als Hoare-Logik Zusicherungen als Vor-/Nachbedingungen, Invarianten
Termersetzung:	Programmfortschritt als Termersetzungsregeln
Game semantics:	nahe an denotational, "Spiel" zwischen Programm und System, aktuell viel Forschung in diesem Bereich

Im folgenden nur operationale Semantik

Übersicht formale Semantiken

verschiedene Arten von Semantiken:

operational:	simuliert Zustandsübergänge auf abstrakter Maschine nahe an tatsächlichem Programmverhalten
denotational:	Programm Funktion, bildet Anfangs- auf Endzustand ab sehr mathematisch, gern für Beweise verwendet heute etwas aus der Mode
axiomatisch:	auch bekannt als Hoare-Logik Zusicherungen als Vor-/Nachbedingungen, Invarianten
Termersetzung:	Programmfortschritt als Termersetzungsregeln
Game semantics:	nahe an denotational, "Spiel" zwischen Programm und System, aktuell viel Forschung in diesem Bereich

Im folgenden nur operationale Semantik

aktuelles Forschungsgebiet

Typsicherheitsbeweise für “akademische Spielzeugsprachen”
schon vor Jahrzehnten (von Hand)
echte Sprachen für diesen Ansatz zu komplex

vor 10 Jahren erste formale Semantik für (Untermenge von) Java
Gruppe von Tobias Nipkow an TU München
formalisiert in Isabelle/HOL
dazu später mehr. . .

aktuelles Forschungsgebiet

Typsicherheitsbeweise für “akademische Spielzeugsprachen”
schon vor Jahrzehnten (von Hand)
echte Sprachen für diesen Ansatz zu komplex

vor 10 Jahren erste formale Semantik für (Untermenge von) Java
Gruppe von Tobias Nipkow an TU München
formalisiert in Isabelle/HOL
dazu später mehr. . .

aktuelles Forschungsgebiet

Typsicherheitsbeweise für “akademische Spielzeugsprachen”
schon vor Jahrzehnten (von Hand)
echte Sprachen für diesen Ansatz zu komplex

vor 10 Jahren erste formale Semantik für (Untermenge von) Java
Gruppe von Tobias Nipkow an TU München
formalisiert in Isabelle/HOL
dazu später mehr. . .

Beispiel: einfache While-Sprache

Programmanweisungen:

- `Skip`
- Variablenzuweisung $V := e$
- sequentielle Komposition (Hintereinanderausführung) $c_1; ; c_2$
- if-then-else `if` (b) c_1 `else` c_2
- while-Schleifen `while` (b) c'

Zustand:

beschreibt, welche Werte aktuell in den Variablen (partielle Funktion)

arithmetische/boole'sche Ausdrücke:

Wert des Ausdrucks abhängig von Zustand (wg. Variablen)

Beispiel: einfache While-Sprache

Programmanweisungen:

- `Skip`
- Variablenzuweisung $V := e$
- sequentielle Komposition (Hintereinanderausführung) $c_1; ; c_2$
- if-then-else `if` (b) c_1 `else` c_2
- while-Schleifen `while` (b) c'

Zustand:

beschreibt, welche Werte aktuell in den Variablen (partielle Funktion)

arithmetische/boole'sche Ausdrücke:

Wert des Ausdrucks abhängig von Zustand (wg. Variablen)

Beispiel: einfache While-Sprache

Programmanweisungen:

- `Skip`
- Variablenzuweisung $V := e$
- sequentielle Komposition (Hintereinanderausführung) $c_1; ; c_2$
- if-then-else `if` (b) c_1 `else` c_2
- while-Schleifen `while` (b) c'

Zustand:

beschreibt, welche Werte aktuell in den Variablen (partielle Funktion)

arithmetische/boole'sche Ausdrücke:

Wert des Ausdrucks abhängig von Zustand (wg. Variablen)

Big Step Regeln

Syntax: $\langle s, \sigma \rangle \Rightarrow \sigma'$ Anweisung s in Zustand σ liefert Endzustand σ'

$A(a)\sigma$ Auswertung von arithm. Ausdruck a in Zustand σ

$B(b)\sigma$ Auswertung von boole'schem. Ausdruck b in Zustand σ

$\Rightarrow$-Regeln :

$$\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x := a, \sigma \rangle \Rightarrow \sigma(x \mapsto A(a)\sigma)$$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{B(b)\sigma = \texttt{false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{while } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{B(b)\sigma = \texttt{false}}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c; ; c', \sigma \rangle \Rightarrow \sigma''}$$

Big Step Regeln

Syntax: $\langle s, \sigma \rangle \Rightarrow \sigma'$ Anweisung s in Zustand σ liefert Endzustand σ'

$A(a)\sigma$ Auswertung von arithm. Ausdruck a in Zustand σ

$B(b)\sigma$ Auswertung von boole'schem. Ausdruck b in Zustand σ

$\Rightarrow$ -Regeln :

$$\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x := a, \sigma \rangle \Rightarrow \sigma(x \mapsto A(a)\sigma)$$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{B(b)\sigma = \texttt{false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{while } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{B(b)\sigma = \texttt{false}}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c; ; c', \sigma \rangle \Rightarrow \sigma''}$$

Big Step Regeln

Syntax: $\langle s, \sigma \rangle \Rightarrow \sigma'$ Anweisung s in Zustand σ liefert Endzustand σ'
$A(a)\sigma$ Auswertung von arithm. Ausdruck a in Zustand σ
$B(b)\sigma$ Auswertung von boole'schem. Ausdruck b in Zustand σ

$\Rightarrow$ -Regeln : $\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma$ $\quad \langle x := a, \sigma \rangle \Rightarrow \sigma(x \mapsto A(a)\sigma)$

$$\frac{B(b)\sigma = \texttt{true} \quad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{B(b)\sigma = \texttt{false} \quad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{B(b)\sigma = \texttt{true} \quad \langle c', \sigma \rangle \Rightarrow \sigma' \quad \langle \texttt{while } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{B(b)\sigma = \texttt{false}}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \quad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c; ; c', \sigma \rangle \Rightarrow \sigma''}$$

Big Step Regeln

Syntax: $\langle s, \sigma \rangle \Rightarrow \sigma'$ Anweisung s in Zustand σ liefert Endzustand σ'

$A(a)\sigma$ Auswertung von arithm. Ausdruck a in Zustand σ

$B(b)\sigma$ Auswertung von boole'schem. Ausdruck b in Zustand σ

$\Rightarrow$ -Regeln : $\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma$ $\quad \langle x := a, \sigma \rangle \Rightarrow \sigma(x \mapsto A(a)\sigma)$

$$\frac{B(b)\sigma = \texttt{true} \quad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{B(b)\sigma = \texttt{false} \quad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{B(b)\sigma = \texttt{true} \quad \langle c', \sigma \rangle \Rightarrow \sigma' \quad \langle \texttt{while } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{B(b)\sigma = \texttt{false}}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \quad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c; ; c', \sigma \rangle \Rightarrow \sigma''}$$

Big Step Regeln

Syntax: $\langle s, \sigma \rangle \Rightarrow \sigma'$ Anweisung s in Zustand σ liefert Endzustand σ'

$A(a)\sigma$ Auswertung von arithm. Ausdruck a in Zustand σ

$B(b)\sigma$ Auswertung von boole'schem. Ausdruck b in Zustand σ

$\Rightarrow$ -Regeln :

$$\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x := a, \sigma \rangle \Rightarrow \sigma(x \mapsto A(a)\sigma)$$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{B(b)\sigma = \texttt{false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{while } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{B(b)\sigma = \texttt{false}}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c; ; c', \sigma \rangle \Rightarrow \sigma''}$$

Big Step Regeln

Syntax: $\langle s, \sigma \rangle \Rightarrow \sigma'$ Anweisung s in Zustand σ liefert Endzustand σ'

$A(a)\sigma$ Auswertung von arithm. Ausdruck a in Zustand σ

$B(b)\sigma$ Auswertung von boole'schem. Ausdruck b in Zustand σ

$\Rightarrow$ -Regeln : $\langle \texttt{Skip}, \sigma \rangle \Rightarrow \sigma \qquad \langle x := a, \sigma \rangle \Rightarrow \sigma(x \mapsto A(a)\sigma)$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'} \qquad \frac{B(b)\sigma = \texttt{false} \qquad \langle c', \sigma \rangle \Rightarrow \sigma'}{\langle \texttt{if } (b) \ c \ \texttt{else } c', \sigma \rangle \Rightarrow \sigma'}$$

$$\frac{B(b)\sigma = \texttt{true} \qquad \langle c', \sigma \rangle \Rightarrow \sigma' \qquad \langle \texttt{while } (b) \ c', \sigma' \rangle \Rightarrow \sigma''}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma''}$$

$$\frac{B(b)\sigma = \texttt{false}}{\langle \texttt{while } (b) \ c', \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle c', \sigma' \rangle \Rightarrow \sigma''}{\langle c; ; c', \sigma \rangle \Rightarrow \sigma''}$$

Formalisierung in Isabelle

$[\![a]\!]\sigma$ wertet Ausdruck a in σ aus, ersetzt $A(a)\sigma$ und $B(b)\sigma$ von vorher

```
inductive eval :: "com ⇒ state ⇒ state ⇒ bool"
    ("⟨_,_⟩ ⇒ _" [0,0,0] 81)
where Skip: "⟨Skip,σ⟩ ⇒ σ"
 | Assign: "⟨x:=a,σ⟩ ⇒ σ(x:=([[a]]σ))"
 | Seq: "[[ ⟨c,σ⟩ ⇒ σ'; ⟨c',σ'⟩ ⇒ σ'' ]] ⟹ ⟨c;;c',σ⟩ ⇒ σ''"
 | CondT:
   "[[ [[b]]σ = true; ⟨c,σ⟩ ⇒ σ' ]] ⟹ ⟨if (b) c else c',σ⟩ ⇒ σ'"
 | CondF:
   "[[ [[b]]σ = false; ⟨c',σ⟩ ⇒ σ' ]] ⟹ ⟨if (b) c else c',σ⟩ ⇒ σ'"
 | WhileF: "[[b]]σ = false ⟹ ⟨while (b) c',σ⟩ ⇒ σ"
 | WhileT: "[[ [[b]]σ = true; ⟨c,σ⟩ ⇒ σ'; ⟨while (b) c',σ'⟩ ⇒ σ'' ]]
     ⟹ ⟨while (b) c',σ⟩ ⇒ σ''"
```

Jinja

G. Klein and T. Nipkow.
A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler.
Transaction on Programming Languages and Systems, 28(4):619-695, ACM, 2006.
`http://afp.sf.net/entries/Jinja.shtml`

Jinja

Ziel: formale Semantik von Java

Überblick:

- beschränkt auf Basiskonstrukte in Java
- bietet Semantik und Typsystem für diese Konstrukte
- darauf basierend Typsicherheitsbeweis
- auch JVM formalisiert, Semantik, Typsystem und Typsicherheitsbeweis

Durch deutlich komplexere Sprache mehr Formalisierungsarbeit nötig

Jinja

Ziel: formale Semantik von Java

Überblick:

- beschränkt auf Basiskonstrukte in Java
- bietet Semantik und Typsystem für diese Konstrukte
- darauf basierend Typsicherheitsbeweis
- auch JVM formalisiert, Semantik, Typsystem und Typsicherheitsbeweis

Durch deutlich komplexere Sprache mehr Formalisierungsarbeit nötig

Das Programm

Formalisierung der Klassenhierarchie als Liste von Klassen bezeichnet als Programm

jede Klasse

- eindeutigen Namen
- Oberklasse (falls keine Oberklasse, Oberklasse `Object`)
- Liste von Feldern
- Liste von Methoden (jeweils mit Methodenrümpfen)

Das Programm

Formalisierung der Klassenhierarchie als Liste von Klassen bezeichnet als Programm

jede Klasse

- eindeutigen Namen
- Oberklasse (falls keine Oberklasse, Oberklasse `Object`)
- Liste von Feldern
- Liste von Methoden (jeweils mit Methodenrümpfen)

Exceptions

Jinja besitzt analog zu Java Exceptions
Problem: wie semantisch darstellen, dass Exception ausgelöst?

Lösung:

- alle Anweisungen sind Ausdrücke, d.h. werten zu etwas aus
- keine Exception: Programm wertet zu einem Wert aus
  (für alle Anweisungen zu dummy-Wert *unit*)
- Exception ausgelöst: Programm wertet zu dieser Exception aus

3 vordefinierte Exceptions:

- *ClassCastException*
- *OutOfMemoryException*
- *NullPointerException*

Exceptions

Jinja besitzt analog zu Java Exceptions
Problem: wie semantisch darstellen, dass Exception ausgelöst?

Lösung:

- alle Anweisungen sind Ausdrücke, d.h. werten zu etwas aus
- keine Exception: Programm wertet zu einem Wert aus
  (für alle Anweisungen zu dummy-Wert *unit*)
- Exception ausgelöst: Programm wertet zu dieser Exception aus

3 vordefinierte Exceptions:

- *ClassCastException*
- *OutOfMemoryException*
- *NullPointerException*

Exceptions

Jinja besitzt analog zu Java Exceptions
Problem: wie semantisch darstellen, dass Exception ausgelöst?

Lösung:

- alle Anweisungen sind Ausdrücke, d.h. werten zu etwas aus
- keine Exception: Programm wertet zu einem Wert aus
  (für alle Anweisungen zu dummy-Wert *unit*)
- Exception ausgelöst: Programm wertet zu dieser Exception aus

3 vordefinierte Exceptions:

- *ClassCastException*
- *OutOfMemoryException*
- *NullPointerException*

Ausdrücke

Jinja bietet:

- Objekterstellung
- casten von Objekten
- binäre Operatoren
- Variablenzugriff und -zuweisung
- Feldzugriff und -zuweisung
- Methodenaufruf (inkl. dynamischer Bindung)
- Blockstrukturen mit lokalen Variablen
- sequentielle Komposition
- if-then-else
- while-Schleifen
- Werfen und Fangen von beliebigen Exceptions

Laufzeitstruktur

wie in Java: Objekte auf dem Heap

Heap: Abbildung Adressen (natürliche Zahlen) nach Objekte

Objekt: Tupel aus Klassenname und Feldern

Felder: Abbildung Tupel (Feldname, definierende Klasse) nach Wert

Lookup kann je nach stat. Typ Feld gleichen Namens in verschiedenen Klassen finden (Feldredefinition)

alle solche Werte müssen gespeichert werden können, also definierende Klasse nötig

Laufzeitstruktur

wie in Java: Objekte auf dem Heap

Heap: Abbildung Adressen (natürliche Zahlen) nach Objekte

Objekt: Tupel aus Klassenname und Feldern

Felder: Abbildung Tupel (Feldname, definierende Klasse) nach Wert

Lookup kann je nach stat. Typ Feld gleichen Namens in verschiedenen Klassen finden (Feldredefinition)

alle solche Werte müssen gespeichert werden können, also definierende Klasse nötig

Laufzeitstruktur

wie in Java: Objekte auf dem Heap

Heap: Abbildung Adressen (natürliche Zahlen) nach Objekte

Objekt: Tupel aus Klassenname und Feldern

Felder: Abbildung Tupel (Feldname, definierende Klasse) nach Wert

Lookup kann je nach stat. Typ Feld gleichen Namens in verschiedenen Klassen finden (Feldredefinition)

alle solche Werte müssen gespeichert werden können, also definierende Klasse nötig

Laufzeitstruktur

wie in Java: Objekte auf dem Heap

- **Heap:** Abbildung Adressen (natürliche Zahlen) nach Objekte
- **Objekt:** Tupel aus Klassenname und Feldern
- **Felder:** Abbildung Tupel (Feldname, definierende Klasse) nach Wert

Lookup kann je nach stat. Typ Feld gleichen Namens in verschiedenen Klassen finden (Feldredefinition)

alle solche Werte müssen gespeichert werden können, also definierende Klasse nötig

Laufzeitstruktur

wie in Java: Objekte auf dem Heap

Heap: Abbildung Adressen (natürliche Zahlen) nach Objekte

Objekt: Tupel aus Klassenname und Feldern

Felder: Abbildung Tupel (Feldname, definierende Klasse) nach Wert
Lookup kann je nach stat. Typ Feld gleichen Namens
in verschiedenen Klassen finden (Feldredefinition)
alle solche Werte müssen gespeichert werden können,
also definierende Klasse nötig

Laufzeitstruktur

wie in Java: Objekte auf dem Heap

Heap: Abbildung Adressen (natürliche Zahlen) nach Objekte

Objekt: Tupel aus Klassenname und Feldern

Felder: Abbildung Tupel (Feldname, definierende Klasse) nach Wert
Lookup kann je nach stat. Typ Feld gleichen Namens
in verschiedenen Klassen finden (Feldredefinition)
alle solche Werte müssen gespeichert werden können,
also definierende Klasse nötig

Semantik

Zustand: modelliert aus Tupel

Heap

lokale Variablen wie bisher, part. Funktion Variable nach Wert

Semantik braucht Programm als Parameter
ausgeführter Ausdruck entspricht `main`-Methode

Zwei Semantiken definiert als induktive Mengen:

Big Step: wie bei While, Auswertung zu Endzustand

Small Step: Auswertung zu Restausdruck und -zustand

Äquivalenz der beiden Semantiken bewiesen

Semantik

Zustand: modelliert aus Tupel

Heap

lokale Variablen wie bisher, part. Funktion Variable nach Wert

Semantik braucht Programm als Parameter
ausgeführter Ausdruck entspricht `main`-Methode

Zwei Semantiken definiert als induktive Mengen:

Big Step: wie bei While, Auswertung zu Endzustand

Small Step: Auswertung zu Restausdruck und -zustand

Äquivalenz der beiden Semantiken bewiesen

Semantik

Zustand: modelliert aus Tupel

Heap

lokale Variablen wie bisher, part. Funktion Variable nach Wert

Semantik braucht Programm als Parameter
ausgeführter Ausdruck entspricht `main`-Methode

Zwei Semantiken definiert als induktive Mengen:

Big Step: wie bei While, Auswertung zu Endzustand

Small Step: Auswertung zu Restausdruck und -zustand

Äquivalenz der beiden Semantiken bewiesen

Semantik

auf diesem Level wichtig: Abstraktion
damit Regeln lesbar und verständlich, Auslagerung der Berechnungen und Zusicherungen in Funktionen bzw. Prädikate

Beispiel: dynamischer Lookup mittels Prädikat
$P \vdash C \texttt{ sees } M : Ts \to T = (pns, body) \texttt{ in } D$

> "In Programm P sieht Klasse C Methode M mit Parametern pns mit Typen Ts, Rückgabetyp T und Methodenrumpf $body$ in Klasse D"

Im Endeffekt geschickte Mengenmanipulation

Semantik

auf diesem Level wichtig: Abstraktion
damit Regeln lesbar und verständlich, Auslagerung der Berechnungen und Zusicherungen in Funktionen bzw. Prädikate

Beispiel: dynamischer Lookup mittels Prädikat
$P \vdash C \texttt{ sees } M : Ts \to T = (pns, body) \texttt{ in } D$

> "In Programm P sieht Klasse C Methode M mit Parametern pns mit Typen Ts, Rückgabetyp T und Methodenrumpf $body$ in Klasse D"

Im Endeffekt geschickte Mengenmanipulation

Semantik

auf diesem Level wichtig: Abstraktion
damit Regeln lesbar und verständlich, Auslagerung der Berechnungen und Zusicherungen in Funktionen bzw. Prädikate

Beispiel: dynamischer Lookup mittels Prädikat
$P \vdash C \texttt{ sees } M : Ts \to T = (pns, body) \texttt{ in } D$

> "In Programm P sieht Klasse C Methode M mit Parametern pns mit Typen Ts, Rückgabetyp T und Methodenrumpf $body$ in Klasse D"

Im Endeffekt geschickte Mengenmanipulation

Semantik

auf diesem Level wichtig: Abstraktion
damit Regeln lesbar und verständlich, Auslagerung der Berechnungen und Zusicherungen in Funktionen bzw. Prädikate

Beispiel: dynamischer Lookup mittels Prädikat
$P \vdash C \texttt{ sees } M : Ts \to T = (pns, body) \texttt{ in } D$

> "In Programm P sieht Klasse C Methode M mit Parametern pns mit Typen Ts, Rückgabetyp T und Methodenrumpf $body$ in Klasse D"

Im Endeffekt geschickte Mengenmanipulation

Typsystem

Typsystem ordnet jedem Ausdruck Typ zu
parametrisiert mit Programm und Typumgebung: Fkt. Variablen nach Typ
Syntax: $P, E \vdash e : T$
auch Typsystem definiert als induktive Menge

prüft auch weitere Eigenschaften, die zu Compilezeit gelten müssen:

- Cast nur auf Klasse ober- oder unterhalb des gecasteten Ausdrucks
- Feldzugriff sieht statisch Klasse mit Felddefinition
- Methodenaufruf sieht statisch Klasse mit Methodendefinition (garantiert, dass dynamischer Lookup Ziel findet)

Typsystem

Typsystem ordnet jedem Ausdruck Typ zu
parametrisiert mit Programm und Typumgebung: Fkt. Variablen nach Typ
Syntax: $P, E \vdash e : T$
auch Typsystem definiert als induktive Menge

prüft auch weitere Eigenschaften, die zu Compilezeit gelten müssen:

- Cast nur auf Klasse ober- oder unterhalb des gecasteten Ausdrucks
- Feldzugriff sieht statisch Klasse mit Felddefinition
- Methodenaufruf sieht statisch Klasse mit Methodendefinition (garantiert, dass dynamischer Lookup Ziel findet)

Typsicherheitsbeweis

Typsicherheit durch Kombination Small Step Semantik und Typsystem (plus einige weitere Plausibilitätsannahmen)

Typsicherheit traditionell durch diese Aussagen: [Wright,Felleisen]

Progress: Semantik darf nicht steckenbleiben

"wohlgetypter Ausdruck, der nicht fertig ausgewertet zu Wert oder Exception, muss weiter ausgewertet werden können"

Preservation: Typ des Ausdrucks darf nur kleiner werden

"durch Anwendung einer Semantikregel darf resultierender Restausdruck nur Typ $\leq$ dem Typ des Ausgangsausdruck haben"

Typsicherheitsbeweis

Typsicherheit durch Kombination Small Step Semantik und Typsystem (plus einige weitere Plausibilitätsannahmen)

Typsicherheit traditionell durch diese Aussagen: [Wright,Felleisen]

Progress: Semantik darf nicht steckenbleiben

"wohlgetypter Ausdruck, der nicht fertig ausgewertet zu Wert oder Exception, muss weiter ausgewertet werden können"

Preservation: Typ des Ausdrucks darf nur kleiner werden

"durch Anwendung einer Semantikregel darf resultierender Restausdruck nur Typ $\leq$ dem Typ des Ausgangsausdruck haben"

Typsicherheitsbeweis

Typsicherheit durch Kombination Small Step Semantik und Typsystem (plus einige weitere Plausibilitätsannahmen)

Typsicherheit traditionell durch diese Aussagen: [Wright,Felleisen]

Progress: Semantik darf nicht steckenbleiben

"wohlgetypter Ausdruck, der nicht fertig ausgewertet zu Wert oder Exception, muss weiter ausgewertet werden können"

Preservation: Typ des Ausdrucks darf nur kleiner werden

"durch Anwendung einer Semantikregel darf resultierender Restausdruck nur Typ $\leq$ dem Typ des Ausgangsausdruck haben"

Typsicherheitsbeweis

Beweise durch Regelinduktion über das Typsystem
Problem: Typsystem dafür zu restriktiv!
Beispiel: Methodenlookup durch Semantikregelanwendung anderes Ziel

Lösung: Laufzeit-Typsystem
Induktionsinvariante, die Haupteigenschaften des Typsystems beibehält
prüft auch Heapinhalt im Gegensatz zu statischem Typsystem

Typsicherheitsbeweis

Beweise durch Regelinduktion über das Typsystem
Problem: Typsystem dafür zu restriktiv!
Beispiel: Methodenlookup durch Semantikregelanwendung anderes Ziel

Lösung: Laufzeit-Typsystem
Induktionsinvariante, die Haupteigenschaften des Typsystems beibehält
prüft auch Heapinhalt im Gegensatz zu statischem Typsystem

Erweiterungen

Ziel: Jinja noch näher an Java
aktuell Arbeit an:

- Arrays (bereits abgeschlossen)
- Threads (in Arbeit):

Andreas Lochbihler.
Type Safe Nondeterminism - A Formal Semantics of Java Threads.
Workshop on Foundations of Object-Oriented Languages, 2008.
`http://afp.sf.net/entries/JinjaThreads.shtml`