Kapitel 21

Semantik

bisher: statisches Verhalten (Subtyping, Verhaltenskonformanz)

nun: dynamisches Verhalten

Ziel: Beschreibung des Berechnungsverhaltens in der Zeit

Mittel: *Operationale Semantik*, simuliert Programmausführung mit best. Programmzustand auf abstrakter Maschine

Vorteil: Validierung von Programmen gegen Spezifikation, Sicherheitseigenschaften:

- Ablaufeigenschaften
- Verifikationsregeln
- Typsicherheitsbeweis

21.1 Grundbegriffe

leicht vereinfacht:

(Programm)zustand: Abbildung von Variablen auf den aktuell gespeicherten Wert:
$\Sigma : Var \to Value, \sigma \in \Sigma$
Zustand aktualisieren, Variable x zugewiesen:
$\sigma[x \mapsto y] = \lambda z.$ if $z = x$ then y else $\sigma(z)$

Syntax einer vereinfachten OO-Sprache:

- Ausdrücke (Variablen, Memberzugriffe):
  $e ::= v \mid x.v$
- Anweisungen: $c ::= x.v := e \mid x.m(e) \mid$
  $x :=$ new $\tau \mid c; c' \mid$ if (e) $c \mid$ while (e) c

Auswerten von Ausdrücken:
Wert von e im Zustand σ: $[\![e]\!]\sigma$

Schritt in der Semantik: $\langle c, \sigma \rangle \Rightarrow \sigma'$
"Anweisung c ausgewertet in Zustand σ führt zu Endzustand σ'"
deshalb: *Big Step Semantik*
(oder "natural semantics")

21.2 Semantikregeln

Zuweisung: $\langle x.v := e, \sigma\rangle \Rightarrow \sigma[x.v \mapsto [\![e]\!]\sigma]$

Objekterzeugung: c_τ Konstruktor von τ,
initObj(τ) legt leeres Objekt vom Typ τ an

$$\frac{\langle c_\tau, \sigma[x \mapsto \text{ initObj}(\tau)]\rangle \Rightarrow \sigma'}{\langle x := \text{new } \tau, \sigma\rangle \Rightarrow \sigma'}$$

Methodenaufruf:
x sieht m mit Rumpf c_m und Parameter p

$$\frac{\langle c_m, \sigma[p \mapsto [\![e]\!]\sigma]\rangle \Rightarrow \sigma'}{\langle x.m(e), \sigma\rangle \Rightarrow \sigma'[p \mapsto \sigma(p)]}$$

Der lokale Parameter p muss unterschieden werden von einem evtl an der Aufrufstelle definierten (anderen) p

While-Schleife: 2 Regeln

$$\frac{[\![e]\!]\sigma = \texttt{False}}{\langle \text{while } (e)\ c, \sigma\rangle \Rightarrow \sigma}$$

$$\frac{[\![e]\!]\sigma = \texttt{True} \quad \langle c, \sigma\rangle \Rightarrow \sigma' \quad \langle \text{while } (e)\ c, \sigma'\rangle \Rightarrow \sigma''}{\langle \text{while } (e)\ c, \sigma\rangle \Rightarrow \sigma''}$$

Komposition:
$$\frac{\langle c, \sigma\rangle \Rightarrow \sigma' \qquad \langle c', \sigma'\rangle \Rightarrow \sigma''}{\langle c; c', \sigma\rangle \Rightarrow \sigma''}$$

Übung. Geben Sie Regeln für if (b) c an!

Beispiel: “Programm” `x=new c; x.inc(); v=x.n`
wobei `class c{int n; c(){n=0;}; void inc(){n=n+1;}}`

Einzelschritte:

A: $$\frac{\langle n := 0, \sigma[x \mapsto \text{initObj}(c)]\rangle \Rightarrow \sigma'[x.n \mapsto 0]}{\langle x := \text{new } c, \sigma\rangle \Rightarrow \sigma'[x.n \mapsto 0]}$$

B: $$\frac{\langle n := n+1, \sigma'[x.n \mapsto 0]\rangle \Rightarrow \sigma'[x.n \mapsto 1]}{\langle x.inc(), \sigma'[x.n \mapsto 0]\rangle \Rightarrow \sigma'[x.n \mapsto 1]}$$

C: $$\langle v := x.n, \sigma'[x.n \mapsto 1]\rangle \Rightarrow \sigma'[x.n \mapsto 1, v \mapsto [\![x.n]\!](\sigma'[x.n \mapsto 1])]$$

Komposition: (σ'' sei $\sigma'[x.n \mapsto 1, v \mapsto 1]$)

$$\text{A}\quad \frac{\text{B}\qquad \text{C}}{\langle x.inc(); v := x.n, \sigma'[x.n \mapsto 0]\rangle \Rightarrow \sigma''}}{\langle x := \text{new } c; x.inc(); v := x.n, \sigma\rangle \Rightarrow \sigma''}$$

Übung. Berechnen sie die Semantikschritte für
`x=new c; while(even(x.n)){x.inc();}`

21.3 Hoare-Kalkül

Hoare-Tripel: $\{P\}\ c\ \{Q\}$
P Vorbedingung, Q Nachbedingung von c

Beweis der Gültigkeit des Hoare-Tripel mittels Semantik:
$\{P\}\ c\ \{Q\} \iff (\forall \sigma\, \sigma'.\ P(\sigma) \wedge (\langle c, \sigma\rangle \Rightarrow \sigma') \Longrightarrow Q(\sigma'))$

Bem. Für nichtterminierendes c gilt jedes Tripel,
da kein $\langle c, \sigma\rangle \Rightarrow \sigma'$ existiert und somit linke Seite
immer `False`

Beispiel: $c = i := i + 5$,
$P \equiv \lambda\sigma.\ \sigma(i) = 5 \Longrightarrow P(\sigma[i \mapsto 5])$,
$Q \equiv \lambda\sigma.\ \sigma(i) = 10 \Longrightarrow Q(\sigma[i \mapsto 10])$,
$\langle i := i + 5, \sigma\rangle \Rightarrow \sigma[i \mapsto [\![i + 5]\!]\sigma]$

$P(\sigma) \wedge (\langle i := i + 5, \sigma\rangle \Rightarrow \sigma[i \mapsto 10]) \Longrightarrow Q(\sigma[i \mapsto 10])$
also $\{i = 5\}\ i := i + 5\ \{i = 10\}$ gültig

weitere Beispiele:

$\{\texttt{True}\}\ i := 10\ \{i = 10\}$	gültig
$\{x = y\}\ x := x + 1\ \{x \neq y\}$	gültig
$\{y \leq x\}\ x := y + z\ \{y \leq x\}$	nicht gültig (z negativ)
$\{\texttt{True}\}\ c\ \{\texttt{False}\}$	gültig, falls c nicht terminiert

21.4 Small Step Semantik

auch genannt “structural operational semantics”

nicht ein großer, sondern viele kleine Einzelschritte

Syntax: $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$

“Ein Teil der Anweisungen in c wird in Anfangszustand σ abgearbeitet, so dass Restanweisungen c' in neuem Zustand σ' bleiben”

Val v: Rückgabewerts eines Statements (nicht mehr weiter auswertbar)

Beispielregeln:

Zuweisung:

$\langle x := e, \sigma \rangle \to \langle \text{Val } (\llbracket e \rrbracket \sigma), \sigma[x \mapsto \llbracket e \rrbracket \sigma] \rangle$

Komposition:

$$\frac{\langle c, \sigma \rangle \to \langle c', \sigma' \rangle}{\langle c; c_2, \sigma \rangle \to \langle c'; c_2, \sigma' \rangle} \qquad \langle \text{Val } v; c_2, \sigma \rangle \to \langle c_2, \sigma \rangle$$

Methodenaufruf:

durch Inlining des Methodenrumpfs

While-Schleife:

$\langle \text{while}(b)\ c, \sigma \rangle \to \langle \text{if}(b)\ (c; \text{while}(b)\ c) \text{ else Skip}, \sigma \rangle$

Übung. Regeln für $x := \text{new } \tau$ und $\text{if } (b)\ c$

Einzelschritte können Phänomene beschreiben,
die ein großer Schritt nicht kann

Beispiel: parallele Ausführung ||:

$$\frac{\langle c_1, \sigma\rangle \to \langle c_1', \sigma'\rangle}{\langle c_1 \mid\mid c_2, \sigma\rangle \to \langle c_1' \mid\mid c_2, \sigma'\rangle} \qquad \frac{\langle c_2, \sigma\rangle \to \langle c_2', \sigma'\rangle}{\langle c_1 \mid\mid c_2, \sigma\rangle \to \langle c_1 \mid\mid c_2', \sigma'\rangle}$$

$$\langle \text{Val } v \mid\mid c_2, \sigma\rangle \to \langle c_2, \sigma\rangle \qquad \langle c_1 \mid\mid \text{Val } v, \sigma\rangle \to \langle c_1, \sigma\rangle$$

Beispiel: $\langle x := 7 \mid\mid x := x + 5, \sigma[x \mapsto 0]\rangle \to$
$\langle \text{Val } 7 \mid\mid x := x + 5, \sigma[x \mapsto 7]\rangle \to$
$\langle x := x + 5, \sigma[x \mapsto 7]\rangle \to \sigma[x \mapsto 12]$

Alternative Regel im 1.Schritt:
$\langle x := 7 \mid\mid x := x + 5, \sigma[x \mapsto 0]\rangle \to$
$\langle x := 7 \mid\mid \text{Val } 5, \sigma[x \mapsto 5]\rangle \to$
$\langle x := 7, \sigma[x \mapsto 5]\rangle \to \sigma[x \mapsto 7]$

Zwei mögliche Lösungen wegen alternativen Ableitungen (Interleaving)

Mit Big Step Semantik nur Nichtdeterminismus, aber kein Interleaving beschreibbar

Nichttermination:

Programm `x=1; while(x>0){x:=x+1;}`

In Small Step:
$\langle x := 1; \text{while}(x > 0)(x := x + 1), \sigma\rangle \rightarrow$
$\langle \text{while}(x > 0)(x := x + 1), \sigma[x \mapsto 1]\rangle \rightarrow$
$\langle \text{if}(x > 0)(x := x + 1; \text{while}(x > 0)(x := x + 1))$
$\text{else Skip}, \sigma[x \mapsto 1]\rangle \rightarrow$
$\langle x := x + 1; \text{while}(x > 0)(x := x + 1), \sigma[x \mapsto 1]\rangle \rightarrow$
$\langle \text{while}(x > 0)(x := x + 1), \sigma[x \mapsto 2]\rangle \rightarrow \ldots$

unendliche Auswertung, in jedem Schritt eine Regel anwendbar

In Big Step:
$\langle x := 1; \text{while}(x > 0)(x := x + 1), \sigma\rangle \Rightarrow ?$

Endzustand kann nicht angegeben werden

Äquivalent: keine anwendbare Regel existiert

Also in Big Step Unterscheidung “Steckenbleiben” und “Nichttermination” nicht möglich!

21.5 Typsicherheit

"Well-typed programs cannot go wrong", genauer:
wohlgetypte Programme enthalten keine unbehandelten Laufzeitfehler/Abstürze (Exceptions sind erlaubt, da Fehler an Auftrittstelle behandelt)

insbesondere Cardelli-Typsystem: garantiert keine illegalen Downcasts

Das muss man aber beweisen: Typsicherheitsbeweis!

Beweistechnik: Kombination von *Typsystem* und *Semantik*

Beobachtung: Programm wohlgetypt, wenn Typ eines Ausdrucks durch Auswertung gleich bleibt (auch kleinere Typen erlaubt)

Standardverfahren von Wright und Felleisen:
Progress und *Preservation*

Progress: c nicht komplett ausgewertet
$\Longrightarrow \forall \sigma.\ \exists c' \sigma'.\ \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$

Preservation: $(\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle) \wedge (c : T)$
$\Longrightarrow \exists T'.\ (c' : T') \wedge (T' \leq T)$

Beide Eigenschaften bewiesen $\Rightarrow$ Typsicherheit

Beweis der Eigenschaften mittels Regelinduktion über Small Step Semantik

Beispiel: Beweis Preservation für Zuweisungsregel
$\langle x := e, \sigma \rangle \rightarrow \langle \text{Val}\ (\llbracket e \rrbracket \sigma), \sigma[x \mapsto \llbracket e \rrbracket \sigma] \rangle$

$(x := e) : \text{Class}\ C$, falls V Variable vom Typ Class C und e vom Typ Class D mit Class $D \leq$ Class C (wg. eventueller Typkonversion/Upcast)

$\text{Val}\ (\llbracket e \rrbracket \sigma) : \text{Class}\ D$
also gesuchtes $T' = \text{Class}\ D \leq \text{Class}\ C = T$

Bem. Es gibt maschinengeprüfte Typsicherheitsbeweise für Java [Nipkow et al.] (incl. Threads) und C++ [Wasserrab et al.] nach diesem Muster (also mit Semantik und Typsystem)