Kapitel 2
Grundlagen und Semantik von Programmiersprachen

Wolf Zimmermann

Verifikation von Übersetzern

Inhalt

Ziele

- Kennenlernen der theoretischen Grundlagen
- Erwerben der Fähigkeit zur Formalisierung der Semantik von Programmiersprachen
- Umsetzen des Korrektheitsbegriffs auf formale Semantiken

2.1 Motivation

Aufbau einer Programmiersprache

Trennt Syntax und Semantik

Syntax einer Programmiersprache

Was ist wohlgeformtes Programm?

Semantik einer Programmiersprache

- Welche abstrakte Bedeutung hat ein wohlgeformtes Programm?
- Selbstbezug

Pragmatik einer Programmiersprache

- Welche konkrete Bedeutung hat ein wohlgeformtes Programm?
- Bezug auf Umwelt und Rechner

Statische und dynamische Semantik

Statische Semantik einer Programmiersprache

- Definiert Eigenschaften von Programmen, die ohne Ausführung bestimmt werden können
  - ☞ Typisierung
  - ☞ Gültigkeitsbereich von Variablen

Dynamische Semantik einer Programmiersprache

- Ordnet Sprachelementen Bedeutung zu
- Definiert Ausführung des Programms

Arten formaler Semantiken

Denotationale Semantik

Jedem Sprachelement wird eine (evtl. rekursiv) Zustandsübergangsfunktion zugeorndet

- Mathematische Basis: vollständige Halbordnungen, stetige Funktionen

Strukturell operationale Semantik

Jedem Sprachelement wird eine Zustandstransformation zugeordnet

- Mathematische Basis: Inferenzregeln

Abstrakte Maschine

- Jedem Sprachelement werden Befehlsfolgen (evtl. rekursiv) zugeordnet
- Jedem Befehl wird eine Zustandstransformation zugeordnet
- Basis: Algebren und Abstrakte Zustandsmaschinen

Kriterien für die Übersetzungsverifikation

- Einheitliche Art der formalen Semantik für alle beteiligten Sprachen
- Modularer Aufbau der Sprachsemantiken ermöglicht Modularisierung der Korrektheitsbeweise
  - Formale Semantik für einen einfachen Sprachkern
  - Sukzessive Anreicherung um neue Sprachkonstrukte bis komplette Sprache erreicht wird **ohne** Änderung der bisher definierten Semantik der Teilsprachen
- Prozessorhandbücher ordnen jedem Maschinenbefehl Registertransfers zu

$\Rightarrow$ Operationale Semantik

- Zwischensprachsemantiken und Programmiersprachsemantiken lassen sich mit operationalen Semantiken mindestens so leicht formalisieren wie mit denotationalen Semantiken
- Abstrakte Maschinen sind für Maschinensprachen besser geeignet als strukturell operationale Semantiken
- Zwischensprachen und Programmiersprachsemantiken lassen sich modular über abstrakte Maschinen oft leichter aufbauen als strukturell operationale Semantiken

Fazit und Ausblick auf den Rest des Kapitels

Fazit

Formalisierung der Semantiken der beteiligten Programmiersprachen über abstrakte Maschinen

Durchführung

Definition der abstrakten Maschinen mit Spezifikationsmethode **Abstrakte Zustandsmaschinen** (engl. *Abstract State Machines* oder kurz: ASM):

- Zustände sind Σ-Algebren über einer Signatur Σ
- Zustandsübergänge ändern die Interpretation von Σ-Termen

$\Rightarrow$ Abstrakte Zustandsmaschinen spezifizieren ein Zustandsübergangssystem

$\Rightarrow$ Begriffe der Korrektheit aus Kapitel 1 können direkt eingesetzt werden

$\Rightarrow$ Es genügt die Definition beobachtbaren Verhaltens und der Nachweis von n-m-Simulationen

2.2 Signaturen, Terme, Algebren

Definition 2.1 (Signatur)

Paar $\Sigma = (S, F)$ wobei S Menge von **Sorten** ist und $F = (F_{w,s})_{w \in S^*, s \in S}$ Menge von **Operationssymbolen** ist.
Notation: $f : s_1 \times \cdots \times s_n \to s$ ist Operationssymbol der Stelligkeit $s_1 \cdots s_n, s$

Definition 2.2 (Terme)

Sei $\Sigma \triangleq (S, F)$ eine Signatur. Ein Σ-**Term der Sorte** s **mit Variablen** $(X_s)_{s \in S}$ ist induktiv definiert (X_s sind **Variablen der Sorte** s, $X_s \cap F_{\varepsilon,s} = \emptyset$):
i. Jede Variable $x \in X_s$ ist Term der Sorte s.
ii. Wenn $f : s_1 \times \cdots \times s_n \to s \in F$ und $t_1, \ldots, t_n$ Terme der Sorte $s_1, \ldots, s_n$ sind, dann ist $f(t_1, \ldots, t_n)$ Term der Sorte s (Insbesondere auch für $n = 0$)
Σ-**Grundterm:** Σ-Term ohne Variablen

Notationen

- $\mathrm{T}_s(\Sigma, X)$ ist die Menge aller Σ-Terme der Sorte s mit Variablen X. $\mathrm{T}(\Sigma, X) \triangleq \bigcup_{s \in S} \mathrm{T}_s(\Sigma, X)$
- $\mathrm{T}_s(\Sigma)$ ist die Menge aller Σ-Grundterme der Sorte s $\mathrm{T}(\Sigma) \triangleq \bigcup_{s \in S} \mathrm{T}_s(\Sigma)$

Beispiel 2.1: Signatur für die Abstrakte Syntax einer einfachen While-Sprache

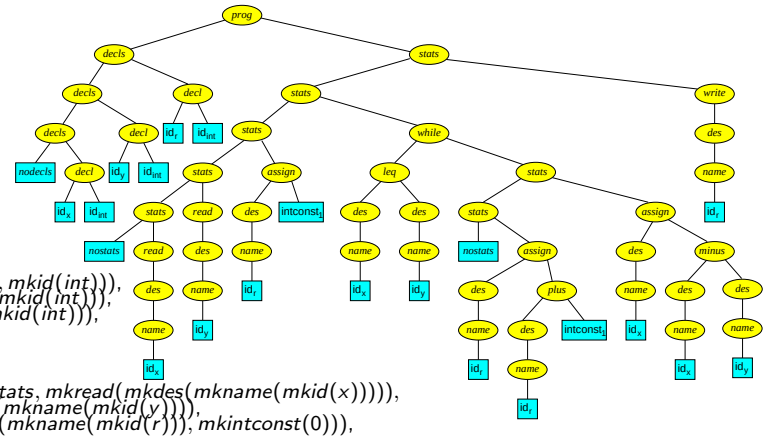
sig WHILE
sorts ID, INTCONST, BOOLCONST, OP, PROG, STATS, DECLS, DECL, TYPE, STAT, EXPR
operations

id :		→ ID
z :		→ INTCONST
true :		→ BOOLCONST
false :		→ BOOLCONST
plus, *minus*, . . . :		→ OP
type : ID		→ TYPE
mkprog :	DECLS × STATS	→ PROG
mknodecls :		→ DECLS
mkdecls :	DECLS × DECL	→ DECLS
mkdecl :	ID × TYPE	→ DECL
mknostats :		→ STATS
mkstats :	STATS × STAT	→ STATS
mkassign :	DES × EXPR	→ STAT
mkif :	EXPR × STATS × STATS	→ STAT
mkwhile :	EXPR × STATS	→ STAT
mkread :	DES	→ STAT
mkwrite :	EXPR	→ STAT
mkdes :	NAME	→ DES
mkname :	*ID*	→ NAME
mkintconst :	INTCONST	→ EXPR
mkboolconst :	INTCONST	→ EXPR
mkgeq :	EXPR × OP × EXPR	→ EXPR
mkplus :	EXPR × OP × EXPR	→ EXPR
mkminus :	EXPR × OP × EXPR	→ EXPR

wobei *id* beliebige Folge aus Buchstaben und Ziffern (beginnend mit Buchstabe) und *z* beliebige ganze Zahl in Dezimaldarstellung ist.

Beispiel 2.1: Programme sind Terme der abstrakten Syntax

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```




```
mkprog(
 mkdecls(
  mkdecls(
   mkdecls(
    mknodecl,
    mkdecl(mkid(x), mkid(int))),
   mkdecl(mkid(y), mkid(int))),
  mkdecl(mkid(r), mkid(int))),
 mkstats(
  mkstats(
   mkstats(
    mkstats(
     mkstats(mknostats, mkread(mkdes(mkname(mkid(x))))),
     mkread(mkdes(mkname(mkid(y))))),
    mkassign(mkdes(mkname(mkid(r))), mkintconst(0))),
   mkwhile(
    mkgeq(mkdes(mkdes(mkname(mkid(x))), mkdes(mkdes(mkname(mkid(y)))))),
    mkstats(
     mkstats(
      mknostats,
      mkassign(
       mkdes(mkname(mkid(r))), mkplus(mkdes(mkdes(mkname(mkid(r)))),
       mkintconst(1)))),
     mkassign(
      mkdes(mkname(mkid(x))),
      mkminus(mkdes(mkname(mkid(x))), mkdes(mkdes(mkname(mkid(y))))))))),
 mkwrite(mkdes(r)))))
```


Diskussion

Beobachtungen

- Terme sind geeignet zur Darstellung abstrakter Syntaxbäume
- Nichtterminale *des*, *plus* etc. werden nur implizit über Konstruktoren als Ausdruck behandelt.
- Keine Behandlung von Alternativen
- Keine Navigation in abstrakten Syntaxbäumen

⇒ Statische Semantik ist nicht darstellbar

- Keine Interpretation von Termen

⇒ Auswertung von Ausdrücken, Zustandsraum nicht darstellbar

⇒ Zustandstransformation nicht darstellbar

Lösungen

- Interpretation von Termen durch Σ-Algebren
- Navigation durch Adressierung der AST-Knoten (*occurrences*)
- Einführung von Untersorten (erlaubt z.B. einheitliche Behandlung der unterschiedlichen Arten von Ausdrücken)
- Statische Semantik durch Bedingungen (bedingte Gleichungen, Formeln)
- Zustandstransformationen verändern Interpretation von Operationssymbolen (abstrakte Zustandsmaschinen)

Algebren

Definition 2.3 (Σ-Algebra)

Sei $\Sigma = (S, F)$ eine mehrsortige Signatur mit Variablen $(X_s)_{s \in S}$. Eine Σ-Algebra ist ein Paar $\mathcal{A} = (A, \Phi)$ wobei

- $A = (A_s)_{s \in S}$ wobei A_s eine Menge mit $X_s \subseteq A_s$ ist (**Trägermenge der Sorte** s)
- $\Phi = \{f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s | f : s_1 \times \cdots \times s_n \to s \in \Sigma\}$, d.h., jedem Operationssymbol f der Stelligkeit $s_1 \cdots s_n, s$ wird eine Funktion $f_A : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ zugeordnet

Interpretation von Termen

Definition 2.4 (Variablenbelegung, Interpretation)

Sei $\Sigma \triangleq (S, F)$ eine mehrsortige Signatur mit Variablen $(X_s)_{s \in S}$ und $\mathcal{A} = (A, \Phi)$ eine Σ-Algebra. Eine **Variablenbelegung** ist eine Familie von Abbildungen $(\beta_s : X_s \to A_s)_{s \in S}$.
Die **Interpretation von Termen** β **und** $\mathcal{A}$ ist eine Familie von Abbildungen $([\![\cdot]\!]^\beta_s : \mathsf{T}_s(\Sigma, X) \to A_s)_{s \in S}$, die wie folgt definiert ist:

i. $[\![x]\!]^\beta_s \triangleq \beta_s(x)$ für $x \in X_s$
ii. $[\![f]\!]^\beta_s \triangleq f_A$ für $f :\to s \in F$
iii. $[\![f(t_1, \ldots, t_n)]\!]^\beta_s \triangleq f_A([\![t_1]\!]^\beta_s, \ldots, [\![t_n]\!]^\beta_s)$ für $f : s_1 \times \cdots \times s_n \to s \in F$ und Terme $t_1 \in \mathsf{T}_{s_1}(\Sigma, X), \ldots, t_n \in \mathsf{T}_{s_n}(\Sigma, X)$

Beobachtung

Abstrakte Syntaxbäume können als Term interpretiert werden

- Algebra $\mathcal{T} = (\mathsf{T}_s(\Sigma), \iota)$ (**Termalgebra**)

Beispiel 2.2: Algebra der ganzen Zahlen

Signatur

sig INT
sorts INT
operations

$\ldots, -2, -1, 0, 1, 2, \ldots :$		$\to$ INT
$plus, mal, minus$	INT $\times$ INT	$\to$ INT

INT-**Algebra**

$\mathfrak{Z} \triangleq \langle Z, \Phi_Z \rangle$

- Trägermenge $\mathsf{INT}_Z \triangleq \mathbb{Z}$
- Intepretationen Φ:

$\ldots, -2_Z \triangleq -2, -1_Z \triangleq -1, 0_Z \triangleq 0, 1_Z \triangleq 1, 2_Z \triangleq 2, \ldots$
$plus_Z \triangleq +$
$mal_Z \triangleq *$
$minus_Z \triangleq -$

Interpretation eines Terms

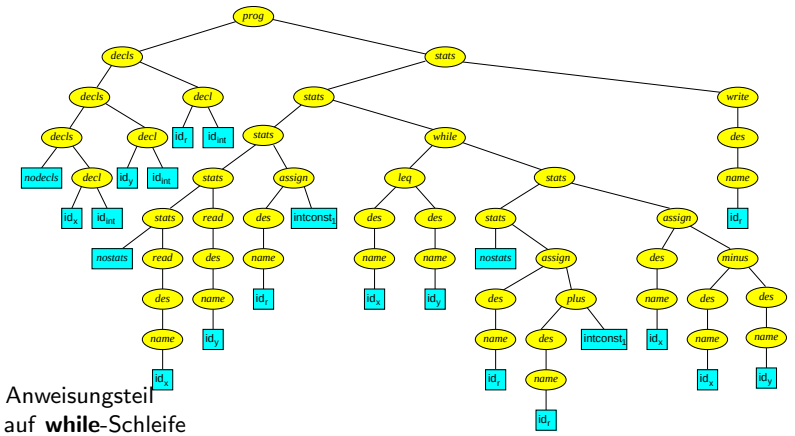
$[\![plus(mkintconst(3), mkintconst(5))]\!] = plus_Z([\![mkintconst(3)]\!], [\![mkintconst(5)]\!]) = [\![mkintconst(3)]\!] + [\![mkintconst(5)]\!] = 3 + 5 = 8$

Beispiel 2.3: Navigation in abstrakten Syntaxbäumen

Idee

Verwende Listen ganzer Zahlen zum Navigieren

- Leere Liste entspricht Wurzel
- Falls Liste l auf Knoten k verweist, verweist Liste $l.i$ auf i-tes Kind von K



1 verweist auf Anweisungsteil
1.0.1 verweist auf **while**-Schleife
1.0.1.0 verweist auf Bedingung der **while**-Schleife
2.0.5 verweist auf keinen AST-Knoten

Diskussion

Beobachtungen

- Signatur WHILE muss erweitert werden
- Aus Sicht der Navigation müssen alle Strukturbaumknoten einheitlich behandelt werden
- Es gibt Navigationslisten (engl *occurrences*), die auf keinen AST-Knoten verweisen.

Lösungen

- Signaturerweiterungen mit entsprechender Erweiterung der Algebren
- Einführung von Untersorten
- Einführung von partiellen Operationen

Partielle Signaturen mit Untersorten

Definition 2.5 (Partielle Signaturen mit Untersorten)

Tupel $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ wobei

i. S endliche Menge von **Sortensymbolen**

ii. $\sqsubseteq \subseteq S \times S$ ist eine Halbordnung auf S (**Untersortenbeziehung**)

iii. $F = \langle F_{w,s} \rangle_{w \in S^*, s \in S}$ ist eine Familie von Operationssymbolen (**totale Operationssymbole**) mit folgender Eigenschaft
Wenn $f : s_1 \times \cdots \times s_n \to s \in F$, $s_1' \sqsubseteq s_1, \ldots, s_n' \sqsubseteq s_n$ und $s \sqsubseteq s'$, dann ist $f : s_1' \times \cdots \times s_n' \to s' \in F$

iv. $F' = \langle F'_{w,s} \rangle_{w \in S^*, s \in S}$ ist eine von F disjunkte Familie von Operationssymbolen, d.h. $F_{w,s} \cap F'_{w,s} = \emptyset$, (**partielle Operationssymbole**) mit folgender Eigenschaft:
Wenn $f : s_1 \times \cdots \times s_n \to ?s \in F'$ $s_1' \sqsubseteq s_1, \ldots, s_n' \sqsubseteq s_n$ unnd $s \sqsubseteq s'$, dann ist $f : s_1' \times \cdots \times s_n' \to ?s' \in F'$

Notationen

- $f : s_1 \times \cdots \times s_n \to s \in F$ statt $f \in F_{s_1 \cdots s_n, s}$
- $f : s_1 \times \cdots \times s_n \to ?s \in F'$ statt $f \in F'_{s_1 \cdots s_n, s}$

Definition 2.6 (Σ-Terme)

Definition analog den klassischen Signaturen
$\Rightarrow$ Sei $s \sqsubseteq s'$. Dann ist $\mathsf{T}_s(\Sigma, X) \subseteq \mathsf{T}_{s'}(\Sigma, X)$

Σ-Algebren für partielle Signaturen mit Untersorten

Definition 2.7 (Partielle ordnungssortierte Algebra)

Sei $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ eine partielle Signatur mit Untersorten. Eine **partielle ordnungssortierte Σ-Algebra** ist ein Paar $\mathfrak{A} = \langle A, \Phi \rangle$ mit

i. $A = \langle A_s \rangle_{s \in S}$ ist eine Familie von **Trägermengen** ist, so dass $A_s \subseteq A_{s'}$ falls $s \sqsubseteq s'$ ist.

ii. Für alle Funktionssymbole $f : s_1^{(1)} \times \cdots \times s_n^{(1)} \to s^{(1)}, \ldots,$
$f : s_1^{(m)} \times \cdots \times s_n^{(m)} \to s^{(m)} \in F$ ist $f_A : \bigcup_{j=1}^{m} A_{s_1}^{(j)} \times \cdots \times \bigcup_{j=1}^{m} A_{s_n}^{(j)} \to \bigcup_{j=1}^{m} A_s^{(j)}$
eine totale Funktion (**Interpretation des Operationssymbols** f), die folgende Bedingung erfüllt: Falls $f : s_1 \times \cdots \times s_n \to s \in F$ und $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ dann ist auch $f_A(a_1, \ldots, a_n) \in A_s$.

iii. Für alle Funktionssymbole $f : s_1^{(1)} \times \cdots \times s_n^{(1)} \to ?s^{(1)}, \ldots,$
$f : s_1^{(m)} \times \cdots \times s_n^{(m)} \to ?s^{(m)} \in F'$ ist $f_A : \bigcup_{j=1}^{m} A_{s_1}^{(j)} \times \cdots \times \bigcup_{j=1}^{m} A_{s_n}^{(j)} \to \bigcup_{j=1}^{m} A_s^{(j)}$
eine partielle Funktion (**Interpretation des Operationssymbols** f), die folgende Bedingung erfüllt: Falls $f : s_1 \times \cdots \times s_n \to ?s \in F'$ und $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ dann ist auch $f_A(a_1, \ldots, a_n) \in A_s$.

Erweiterungen von Signaturen

Definition 2.8 (Signaturerweiterung)

Sei $\Sigma = \langle S, \sqsubseteq, F, F' \rangle$ eine Signatur. Eine Signatur $\overline{\Sigma} = \langle \overline{S}, \overline{\sqsubseteq}, \overline{F}, \overline{F'} \rangle$ mit $S \subseteq \overline{S}$, $\sqsubseteq \subseteq \overline{\sqsubseteq}$, $F \subseteq \overline{F}$ und $F' \subseteq \overline{F'}$ heißt **Erweiterung der Signatur** Σ.

Notationen

- $\Sigma \subseteq \Sigma'$ notiert die Erweiterung Σ' von Σ
- **sig** Σ **extends** $\Sigma_1, \ldots, \Sigma_n$ **by sorts** S **subsorts** $\sqsubseteq$ **operations** $F \uplus F'$ $\triangleq \Sigma = \langle S \cup S_1 \cup \cdots \cup S_n, \sqsubseteq \cup \sqsubseteq_1 \cup \cdots \cup \sqsubseteq_n, F \cup F_1 \cup \cdots \cup F_n, F' \cup F_1' \cup \cdots \cup F_n' \rangle$ wobei $\Sigma_i = \langle S_i, \sqsubseteq_i, F_i, F_i' \rangle$.

Definition 2.9 (Restriktion einer Σ-Algebra)

Sei $\mathfrak{A} = \langle A, \Phi \rangle$ eine Σ_1-Algebra zu einer partiellen Signatur Σ_1 und $\Sigma_0 \triangleq \langle (S_0, \sqsubseteq_0, F_0, F_0' \rangle \subseteq \Sigma_1$. Die **Restriktion von $\mathfrak{A}$ bzgl. Σ** ist definiert durch: $\mathfrak{A}|_{\Sigma_0} \triangleq \langle A', \Phi' \rangle$ wobei $A_s' \triangleq A_s$ für alle $s \in S_0$ und $f_{A'} \triangleq f_A$ für alle $f \in F_0 \cup F_0'$.

Beobachtung

$\mathfrak{A}|_{\Sigma_0}$ ist eine Σ_0-Algebra, die für die Sorten und für die Operationssymbole aus $\Sigma_0 \subseteq \Sigma_1$ dieselbe Interpretation hat wie die Σ_1-Algebra $\mathfrak{A}$.

Beispiel 2.4: Navigation in abstrakten Syntaxbäumen

Idee

- Führe Signatur und Algebra für Listen ganzer Zahlen ein
- Erweitere Signatur für abstrakte Syntaxbäume um Navigationsleisten
- Definiere entsprechende Algebra

Listen ganzer Zahlen

sig OCC **extends** INT

sorts	OCC			
operations	$()$:		$\to$ OCC	leere Liste
	$(.)$:	OCC $\times$ INT	$\to$ OCC	Infixnotation

OCC-Algebra: $\mathfrak{O} \triangleq \langle O, \Phi \rangle$ wobei

- $\mathfrak{O}|_{\text{INT}} = \mathfrak{Z}$
- O_{OCC} sind alle endlichen Listen ganzer Zahlen, d.h. $O_{\text{OCC}} \triangleq \{[x_1, \ldots, x_n] : n \geq 0, x_i \in \mathbb{Z}\}$
- $()_O \triangleq []$
  $(.)_O([x_1, \ldots, x_n], x) \triangleq [x_1, \ldots, x_n, x]$

Beispiel 2.4: Navigation in abstrakten Syntaxbäumen (Forts.)

Navigation in abstrakten Syntaxbäumen

sig PROG **extends** WHILE, OCC
sorts ASSIGN, READ, WRITE, WHILE, IF, PLUS, GEQ, MINUS, NODE
subsorts ASSIGN $\sqsubset$ STAT, READ $\sqsubset$ STAT, WRITE $\sqsubset$ STAT, WHILE $\sqsubset$ STAT, IF $\sqsubseteq$ STAT,
DES $\sqsubset$ EXPR, PLUS $\sqsubset$ EXPR, GEQ $\sqsubset$ EXPR, MINUS $\sqsubset$ EXPR,
ID $\sqsubset$ NODE, INTCONST $\sqsubset$ NODE, BOOLCONST $\sqsubset$ NODE,
OP $\sqsubset$ NODE, PROG $\sqsubset$ NODE, STATS $\sqsubset$ NODE, DECLS $\sqsubset$ NODE,
DECL $\sqsubset$ NODE, TYPE $\sqsubset$ NODE, STAT $\sqsubset$ NODE, EXPR $\sqsubset$ NODE
operations

mkassign :	DES × EXPR	→ ASSIGN
mkif :	EXPR × STATS × STATS	→ IF
mkwhile :	EXPR × STATS	→ WHILE
mkread :	DES	→ READ
mkwrite :	EXPR	→ WRITE
mkgeq :	EXPR × OP × EXPR	→ GEQ
mkplus :	EXPR × OP × EXPR	→ PLUS
mkminus :	EXPR × OP × EXPR	→ MINUS
mkdes :	NAME	→ EXPR
occ :	OCC × NODE	→?NODE
parent :	OCC	→?OCC

PROG-Algebra $\mathfrak{P}$ ist definiert durch: $\mathfrak{P}|_{\text{WHILE}} \triangleq \mathcal{T}(\text{WHILE})$, $\mathfrak{P}|_{\text{OCC}} \triangleq \mathfrak{O}$
occ ist in Beispiel 2.3 informell definiert

- $P_{\text{NODE}} \triangleq P_{\text{ID}} \cup \cdots \cup P_{\text{EXPR}}$
- Falls $occ_P([], t) = t$ für alle $t \in P_{\text{NODE}}$
- Falls $occ_P(o, t) = mkprog(d, s)$ ist, dann ist $occ_P(o{+}{+}[0], t) = d$, $occ(o{+}{+}[1], t) = s$ und $occ_P(o{+}{+}[i], t) = \bot_{\text{NODE}}$ für $i \geq 2$
- Falls $occ_P(o, t) = mkif(e, s_1, s_2)$ ist, dann ist $occ_P(o{+}{+}[0], t) = e$, $occ(o{+}{+}[1], t) = s_1$, $occ(o{+}{+}[2], t) = s_2$ und $occ_P(o{+}{+}[i], t) = \bot_{\text{NODE}}$ für $i \geq 3$ usw.
- $parent_P(o{+}{+}[i]) = o$

Diskussion

Problem

Semantik wird an der konkreten Definition der Algebra festgelegt

- Eigenschaften nicht transparent auf Spezifikatiationsebene
- Umständliche Definition

Idee

Spezifikation der Eigenschaften durch Gleichungen, Formeln etc.

- Nachweis von Eigenschaften über Beweiskalküle
- Hier genügen zunächst bedingte und unbedingte Gleichungen

Gleichungen und bedingte Gleichungen

Definition 2.10 ((Bedingte) Σ-Gleichung, Σ-Definitionsbereichsprädikat, Σ-Sortenprüfung)

Sei $\Sigma \triangleq (S, \sqsubseteq, F, F')$ eine Signatur. Eine Σ-**Gleichung** ist ein Paar von Σ-Termen $t_1 \doteq t_2$, $t_1, t_2 \in \mathsf{T}_s(\Sigma, X)$ für eine Sorte s und Variablen X. Für $s \in S$, $t \in \mathsf{T}_s(\Sigma, X)$ heißen $D_s(t)$ und $\neg D_s(t)$ Σ-**Definitionsbereichsprädikat** sowie t **is** s und $\neg(t$ **is** $s)$ Σ-**Sortenprüfung**. Eine **bedingte** Σ-**Gleichung** ist ein $n+1$-Tupel $eq_1 \wedge \cdots \wedge eq_n \Rightarrow eq$, $n \geq 0$, wobei $eq_1, \ldots, eq_n, eq \in \mathsf{E}(\Sigma, X) \cup \mathsf{D}(\Sigma, X) \cup \mathsf{S}(\Sigma, X)$.

Notationen:

- $\mathsf{E}(\Sigma, X)$ notiert die Menge der Σ-Gleichungen mit Variablen X.
- $\mathsf{D}(\Sigma, X)$ notiert die Menge der Σ-Definitionsbereichsprädikate mit Variablen X.
- $\mathsf{S}(\Sigma, X)$ notiert die Menge der Σ-Sortenprüfungen mit Variablen X.
- $\mathsf{C}(\Sigma, X)$ notiert die Menge der bedingten Σ-Gleichungen mit Variablen X

Erfüllung von Gleichungen und bedingten Gleichungen

Definition 2.11 (Erfüllung)

Sei $\mathfrak{A} = \langle A, \Phi \rangle$ eine Σ-Algebra und X eine Menge von Variablen.

i. $\mathfrak{A}$ **erfüllt** Σ-**Gleichung** $t_1 \doteq t_2$ gdw. $[\![t_1]\!]_A^\beta = [\![t_2]\!]_a^\beta$ für alle Belegungen $\beta : X \to A$ (Notation: $\mathfrak{A} \models t_1 \doteq t_2$)

ii. $\mathfrak{A}$ **erfüllt Definitionsprädikat** $D_s(t)$ gdw. $[\![t]\!]_A^\beta \neq \bot$ für alle Belegungen β mit $\beta_s(x) \neq \bot_s$ (Notation: $\mathfrak{A} \models D_s(t)$) und $\mathfrak{A} \models \neg D_s(t)$ gdw. $\mathfrak{A} \not\models D_s(t)$

iii. $\mathfrak{A}$ **erfüllt Typprüfung** t **is** s gdw. $[\![t]\!]_A^\beta \in A_s$ für alle Belegungen β. (Notation: $\mathfrak{A} \models t$ **is** s) und $\mathfrak{A} \models \neg(t$ **is** $s)$ gdw. $\mathfrak{A} \not\models t$ **is** s

iv. $\mathfrak{A}$ **erfüllt bedingte** Σ-**Gleichung** $eq_1 \wedge \cdots \wedge eq_n \Rightarrow eq$ gdw. $\mathfrak{A} \models eq_1, \ldots, \mathfrak{A} \models eq_n$ impliziert $\mathfrak{A} \models eq$. (Notation: $\mathfrak{A} \models eq_1 \wedge \cdots \wedge eq_n \Rightarrow eq$)

Beobachtungen

- Kann erweitert werden auf Mengen von Algebren und Mengen von Gleichungen
- Umkehrung: Menge bedingter Gleichungen definiert eine Menge von Algebren, die diese Gleichungen erfüllen
- Falls $\Sigma \subseteq \Sigma'$, $c \in \mathsf{C}(\Sigma, X)$ und $\mathfrak{A} \models c$ für eine Σ′-Algebra $\mathfrak{A}$, dann gilt auch $\mathfrak{A}|_\Sigma \models c$

$\Rightarrow$ Verwende bedingte Gleichungen zur Spezifikation gewünschter Eigenschaften

Abstrakte Datentypen

Definition 2.12 (Abstrakter Datentyp und Interpretation)

Ein **abstrakter Datentyp** ist ein Tripel $\mathcal{A} \triangleq \langle \Sigma, X, C \rangle$ wobei Σ Signatur, X Menge von Variablen und C Menge bedingter Σ-Gleichungen mit Variablen X (**Axiome**). Die **Interpretation** von $\mathcal{A}$ ist die Menge $I_{\mathcal{A}} \triangleq \{\mathfrak{A} \in \mathrm{Alg}(\Sigma) : \mathfrak{A} \models C\}$

Beispiel 2.5: Navigationslisten

spec PROG **extends** WHILE, OCC
sorts ASSIGN, READ, WRITE, WHILE, IF, PLUS, GEQ, MINUS, NODE
subsorts ASSIGN $\sqsubset$ STAT, READ $\sqsubset$ STAT, WRITE $\sqsubset$ STAT, WHILE $\sqsubset$ STAT, IF $\sqsubset$ STAT,
DES $\sqsubset$ EXPR, PLUS $\sqsubset$ EXPR, GEQ $\sqsubset$ EXPR, MINUS $\sqsubset$ EXPR,
ID $\sqsubset$ NODE, INTCONST $\sqsubset$ NODE, BOOLCONST $\sqsubset$ NODE,
OP $\sqsubset$ NODE, PROG $\sqsubset$ NODE, STATS $\sqsubset$ NODE, DECLS $\sqsubset$ NODE,
DECL $\sqsubset$ NODE, TYPE $\sqsubset$ NODE, STAT $\sqsubset$ NODE, EXPR $\sqsubset$ NODE

operations

mkassign :	DES × EXPR	→ ASSIGN
mkif :	EXPR × STATS × STATS	→ IF
mkwhile :	EXPR × STATS	→ WHILE
mkread :	DES	→ READ
mkwrite :	EXPR	→ WRITE
mkgeq :	EXPR × OP × EXPR	→ GEQ
mkplus :	EXPR × OP × EXPR	→ PLUS
mkminus :	EXPR × OP × EXPR	→ MINUS
mkdes :	NAME	→ EXPR
occ :	OCC × NODE	→?NODE
parent :	OCC	→?OCC

vars x : NODE; o : OCC; d : DECLS; s, s_1, s_2 : STAT, i : INT
axioms

$$
\begin{array}{lcl}
occ(o,x) \doteq mkprog(d,s) & \Rightarrow & occ(o.0,x) \doteq d \\
occ(o,x) \doteq mkprog(d,s) & \Rightarrow & occ(o.1,x) \doteq s \\
occ(o,x) \doteq mkprog(d,s) \wedge D_{\mathrm{OCC}}(occ(o.i),x) & \Rightarrow & i \geq 0 \doteq true \\
occ(o,x) \doteq mkprog(d,s) \wedge D_{\mathrm{OCC}}(occ(o.i),x) & \Rightarrow & i \leq 1 \doteq true \\
 & & \cdots \\
 & & \neg D_{\mathrm{OCC}}(parent(())) \\
 & & parent(o.i) \doteq o
\end{array}
$$

2.3 Abstrakte Zustandsmaschinen

Vorgehen zur Definition einer Programmiersprachsemantik

- Definition der abstrakten Syntax samt Navigation
- Definition des nächsten auszuführenden Befehls (Bestandteil der statischen Semantik)
- Definition des Zustandsraums
- Definition des Anfangszustands
- Definition der Zustandsübergänge

Beobachtungen

- Abstrakte Syntax und Navigation wurde bereits durch die Spezifikation PROG definiert.
- Zustandsraum und Zustandsübergangssystem muss noch definiert werden

Fazit

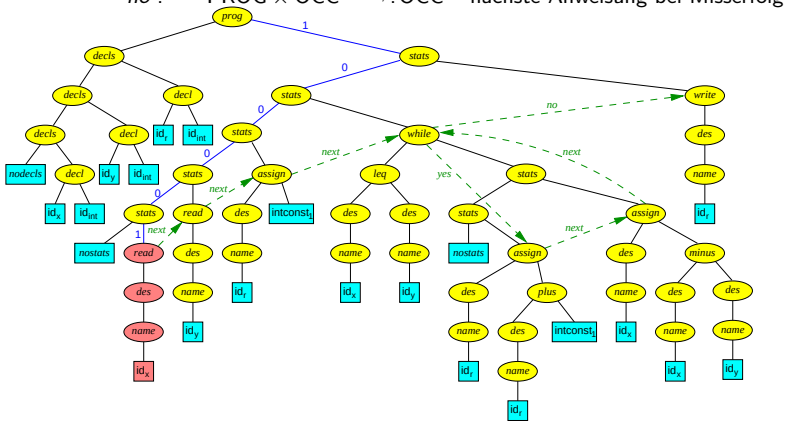
Zusammenfassung

- Abstrakte Datentypen sind geeignet statische Strukturen einer Programmiersprache zu modellieren
- Die abstrakte Syntax kann durch eine Termalgebra dargestellt werden
- Eigenschaften können durch Algebren oder bedingte Gleichungen dargestellt werden
- Um in abstrakten Syntaxbäumen zu navigieren benötigt man partielle Operationssymbole und Untersorten

Beispiel 2.6: Nächste auszuführende Befehle

spec PROGSTAT **extends** PROG
operations

first :	PROG	→ OCC	erste Anweisung
next :	PROG × OCC	→?OCC	nächste Anweisung
yes :	PROG × OCC	→?OCC	nächste Anweisung bei Erfolg
no :	PROG × OCC	→?OCC	nächste Anweisung bei Misserfolg



Übung

Spezifizieren Sie die Eigenschaften von *next*, *yes* und *no* als abstrakter Datentyp und als attributierte Grammatik. Welcher Zusammenhang besteht zwischen den Attributierungsregeln und den bedingten Gleichungen?

Beispiel 2.7: Zustandsraum der While-Sprache

Komponenten des Zustandsraums

- Programm, das ausgeführt wird
- Speicher, d.h. die Menge der Variablen, die Wert enthalten können (Werte sind ganze Zahlen oder Wahrheitswerte)
- Befehlzeiger, d.h. die Stelle im Programm, die als nächstes ausgeführt wird
- Ein- und Ausgabeströme (Listen ganzer Zahlen)

$\Rightarrow$ Das definiert eine Signatur

Spezifikation des Zustandsraums

spec STATE **extends** PROG
sorts VALUE
subsorts $\mathrm{INT} \sqsubseteq \mathrm{VALUE}, \mathrm{BOOL} \sqsubseteq \mathrm{VALUE}$
operations

$prog:$		$\rightarrow$	PROG
$mem:$	ID	$\rightarrow$	?VALUE
$pc:$		$\rightarrow$	OCC
$inp, out:$		$\rightarrow$	OCC
$eval:$	EXPR	$\rightarrow$	?VALUE

vars x : ID, n : INT, b : BOOL, e_1 : EXPR, e_2 : EXPR
axioms

$$
\begin{aligned}
eval(mkdes(mkname(x))) &\doteq mem(x)\\
eval(mkintconst(n)) &\doteq n\\
eval(mkboolconst(b)) &\doteq b\\
eval(mkgeq(e_1, e_2)) &\doteq intgeq(eval(e_1), eval(e_2))\\
eval(mkplus(e_1, e_2)) &\doteq intplus(eval(e_1), eval(e_2))\\
eval(mkminus(e_1, e_2)) &\doteq intminus(eval(e_1), eval(e_2))
\end{aligned}
$$

Beispiel 2.8: Anfangszustände der While-Sprache

Anfangszustand

- Der Speicher ist leer
- Der Ausgabestrom ist leer
- Ausführung beginnt mit Ausführung der Anweisungsfolge

Spezifikation des Anfangszustands

spec INITIALSTATE **extends** STATE
vars x : ID
axioms

$$
\begin{aligned}
&\neg D_{\mathrm{OCC}}(mem(x))\\
out &\doteq ()\\
pc &\doteq first(prog)
\end{aligned}
$$

Diskussion

Beobachtung

- Der Zustandsraum ist eine mehrsortige Signatur Σ

$\Rightarrow$ Die Menge der Zustände ist eine Menge von Σ-Algebren

Ziel

Definition der Zustandsübergänge durch Änderung der Interpretation

Definition 2.13 (Σ-Aktualisierung, Konsistenz, Nachfolgezustand)

Sei $\Sigma \triangleq \langle S, \sqsubseteq, F, F' \rangle$ eine Signatur und $X \triangleq (X_s)_{s \in S}$ eine Menge von Variablen.

i. Eine **Σ-Aktualisierung** ist ein Paar $t_1 := t_2$ von Σ-Termen $t_1, t_2 \in \mathcal{T}_s(\Sigma, X)$ derselben Sorte $s \in S$.

ii. Eine Menge $\mathbb{U}$ von Σ-Aktualisierungen heißt **konsistent bzgl. einer Σ-Algebra** $\mathfrak{A}$ gdw. für alle $f(t_1, \ldots, t_n) := t_0, f(u_1, \ldots, u_n) := u_0 \in \mathbb{U}$ gilt: $[\![t_1]\!]_{\mathfrak{A}} = [\![u_1]\!]_{\mathfrak{A}}, \ldots [\![t_n]\!]_{\mathfrak{A}} = [\![u_n]\!]_{\mathfrak{A}}$ impliziert $[\![t_0]\!]_{\mathfrak{A}} = [\![u_0]\!]_{\mathfrak{A}}$..

iii. Sei $\mathfrak{A}$ eine Σ-Algebra und $\mathbb{U} \neq \emptyset$ eine konsistente Menge von Σ-Aktualisierungen. Der **Nachfolgezustand von $\mathfrak{A}$ bzgl.** $\mathbb{U}$ ist die Σ-Algebra $next_{\mathbb{U}}(\mathfrak{A}) \triangleq (N, [\![\cdot]\!]_{next(\mathfrak{A})})$ wobei $N = (A_s)_{s \in S}$ die Trägermengen von $\mathfrak{A}$ und $[\![\cdot]\!]_{next(\mathfrak{A})}$ durch

$$
[\![f]\!]_{next(\mathfrak{A})}(a_1, \ldots, a_n) \triangleq \begin{cases} a & \text{falls } f(t_1, \ldots, t_n) := t \in \mathbb{U} \text{ und} \\ & [\![t_1]\!]_{\mathfrak{A}} = a_1, \ldots, [\![t_n]\!]_{\mathfrak{A}} = a_n, [\![t]\!]_{\mathfrak{A}} = a \\ [\![f]\!]_A(a_1, \ldots, a_n) & \text{sonst} \end{cases}
$$

definiert ist.

Beispiel 2.9: Aktualisierungen

Spezifikation

spec INIT **extends** INITIALSTATE
axioms

$$
\begin{aligned}
prog &\doteq \text{Term aus Beispiel 1.1}\\
inp &\doteq ().12.5
\end{aligned}
$$

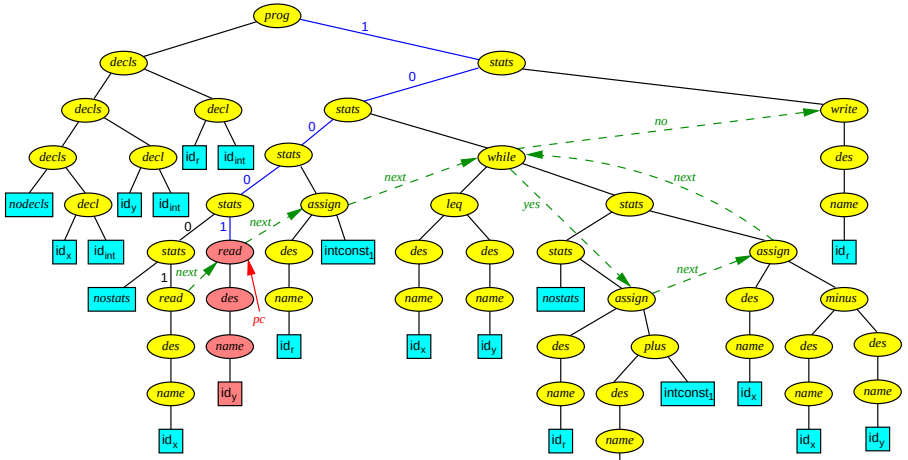
Init-Algebra

$\mathfrak{I} \triangleq \langle I, [\![\cdot]\!]_I \rangle$ mit $I_{\mathrm{VALUE}} \triangleq \mathbb{Z} \cup \mathbb{B}$

$$
\begin{aligned}
[\![mem]\!]_I(x) &\triangleq \bot_{\mathrm{VALUE}}\\
[\![inp]\!]_I &\triangleq [5, 12]\\
[\![out]\!]_I &\triangleq []\\
[\![pc]\!]_I &\triangleq [\![().1.0.0.0.0.1]\!]_I
\end{aligned}
$$

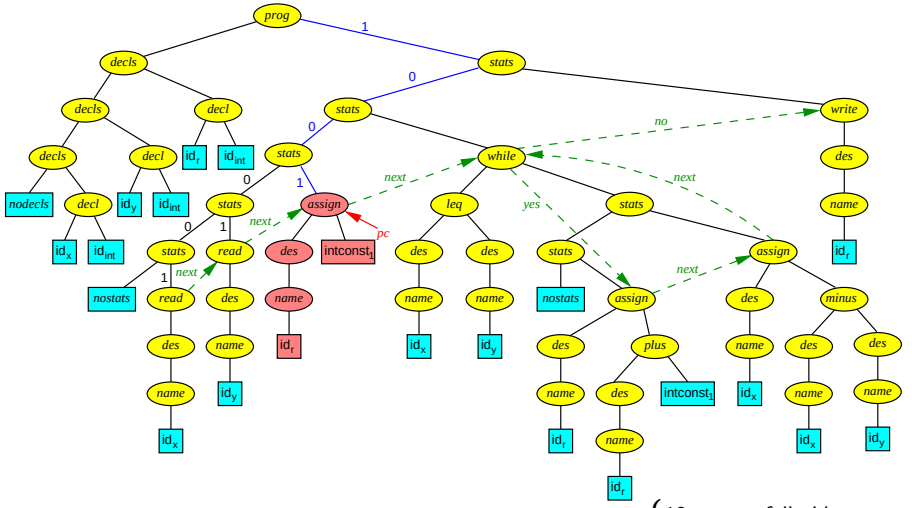
Beispiel 2.9: Erster Zustandsübergang

$\mathfrak{S}_1 \triangleq next_{\mathbb{U}_1}(\mathfrak{I})$ mit $\mathbb{U}_1 \triangleq \{mem(x) := head(inp),\ inp := tail(inp),\ pc := next(prog, pc)\}$



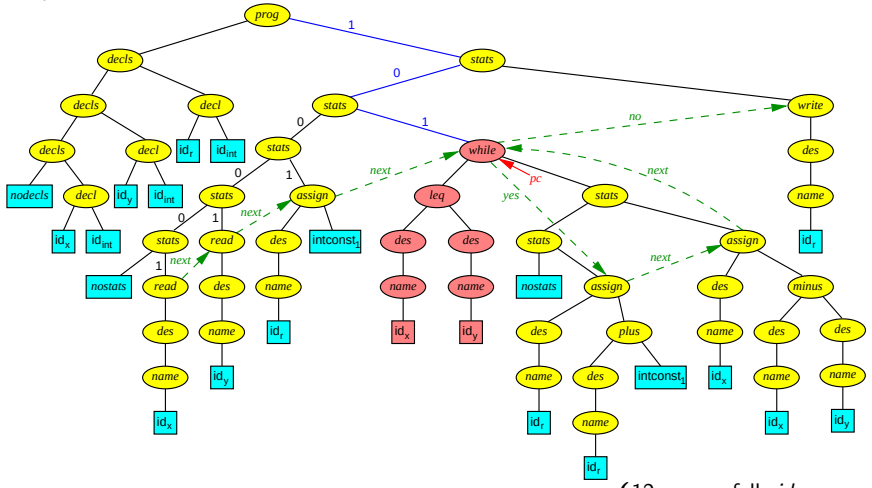
$$\begin{array}{rcl} [\![mem]\!]_I(id) & \triangleq & \bot_{\text{VALUE}} \\ [\![inp]\!]_I & \triangleq & [12, 5] \\ [\![out]\!]_I & \triangleq & [] \\ [\![pc]\!] & \triangleq & [\![().1.0.0.0.0.1]\!]_I \end{array} \rightarrow \begin{array}{rcl} [\![mem]\!]_{S_1}(id) & = & \begin{cases} 12 & \text{falls } id = x \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\ [\![inp]\!]_{S_1} & = & [5] \\ [\![out]\!]_{S_1} & = & [] \\ [\![pc]\!] & = & [\![().1.0.0.0.1]\!]_{S_1} \end{array}$$

$\mathfrak{S}_2 \triangleq next_{\mathbb{U}_2}(\mathfrak{S}_1)$ bzgl. $\mathbb{U}_2 \triangleq \{mem(y) := head(inp),\ inp := tail(inp),\ pc := next(pc, prog)\}$



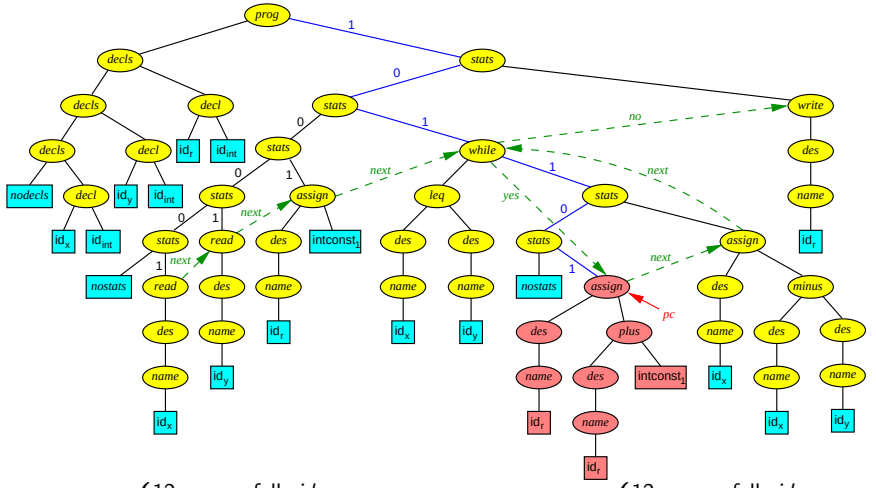
$$\begin{array}{rcl} [\![mem]\!]_{S_1}(id) & = & \begin{cases} 12 & \text{falls } id = x \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\ [\![inp]\!]_{S_1} & = & [5] \\ [\![out]\!]_{S_1} & = & [] \\ [\![pc]\!] & = & [\![().1.0.0.0.1]\!]_{S_1} \end{array} \rightarrow \begin{array}{rcl} [\![mem]\!]_{S_2}(id) & = & \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\ [\![inp]\!]_{S_2} & = & [] \\ [\![out]\!]_{S_2} & = & [] \\ [\![pc]\!] & = & [\![().1.0.0.1]\!]_{S_2} \end{array}$$

$\mathfrak{S}_3 \triangleq next_{\mathbb{U}_3}(\mathfrak{S}_2)$ bzgl. $\mathbb{U}_3 \triangleq \{mem(r) := eval(mkintconst(1)),\ pc := next(prog, pc)\}$



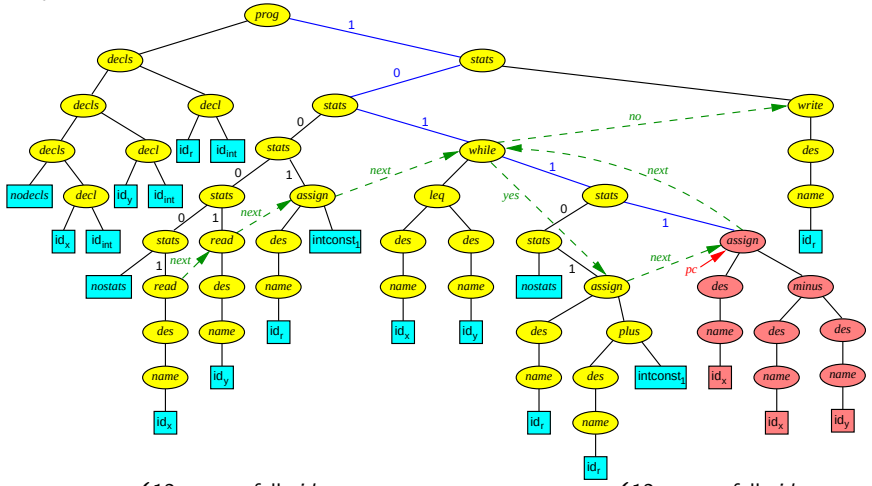
$$\begin{array}{rcl} [\![mem]\!]_{S_2}(id) & = & \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\ [\![inp]\!]_{S_2} & = & [] \\ [\![out]\!]_{S_2} & = & [] \\ [\![pc]\!] & = & [\![().1.0.0.1]\!]_{S_2} \end{array} \rightarrow \begin{array}{rcl} [\![mem]\!]_{S_3}(id) & = & \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\ [\![inp]\!]_{S_3} & = & [] \\ [\![out]\!]_{S_3} & = & [] \\ [\![pc]\!] & = & [\![().1.0.1]\!]_{S_3} \end{array}$$

$\mathfrak{S}_4 \triangleq next_{\mathbb{U}_4}(\mathfrak{S}_3)$ bzgl. $\mathbb{U}_4 \triangleq \{pc := yes(prog, pc)\}$



$$\begin{array}{rcl} [\![mem]\!]_{S_3}(id) & = & \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\ [\![inp]\!]_{S_3} & = & [] \\ [\![out]\!]_{S_3} & = & [] \\ [\![pc]\!] & = & [\![().1.0.1]\!]_{S_3} \end{array} \rightarrow \begin{array}{rcl} [\![mem]\!]_{S_4}(id) & = & \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases} \\ [\![inp]\!]_{S_4} & = & [] \\ [\![out]\!]_{S_4} & = & [] \\ [\![pc]\!] & = & [\![().1.0.1.1.0.1]\!]_{S_4} \end{array}$$

$\mathfrak{S}_5 \triangleq next_{\mathbb{U}_5}(\mathfrak{S}_4)$, $\mathbb{U}_5 \triangleq \{mem(r) := eval(mkplus(des(r), mkintconst(1))), pc := next(prog, pc)\}$



$$[\![mem]\!]_{S_4}(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \perp_{\text{VALUE}} & \text{sonst} \end{cases} \quad \rightarrow \quad [\![mem]\!]_{S_5}(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 2 & \text{falls } id = r \\ \perp_{\text{VALUE}} & \text{sonst} \end{cases}$$

$[\![inp]\!]_{S_4} = [\,]$ $[\![out]\!]_{S_4} = [\,]$ $[\![pc]\!] = [\![().1.0.1.1.0.1]\!]_{S_4}$

$[\![inp]\!]_{S_5} = [\,]$ $[\![out]\!]_{S_5} = [\,]$ $[\![pc]\!] = [\![().1.0.1.1.1]\!]_{S_5}$

Übung

Bestimmen Sie die folgenden STATE-Algebren:

- $\mathfrak{S}_6 \triangleq next_{\mathbb{U}_6}(\mathfrak{S}_5)$ bzgl. $\mathbb{U}_6 \triangleq \{\ mem(x) := eval(mkminus(des(x), des(y))), pc := next(prog, pc)\}$
- $\mathfrak{S}_7 \triangleq next_{\mathbb{U}_7}(\mathfrak{S}_6)$ bzgl. $\mathbb{U}_7 \triangleq \{\ pc := yes(prog, pc)\}$
- $\mathfrak{S}_8 \triangleq next_{\mathbb{U}_8}(\mathfrak{S}_7)$ bzgl. $\mathbb{U}_8 \triangleq$ $\{\ mem(r) := eval(binexpr(des(r), plus, mkintconst(1))), pc := next(prog, pc)\}$
- $\mathfrak{S}_9 \triangleq next_{\mathbb{U}_9}(\mathfrak{S}_8)$ bzgl. $\mathbb{U}_9 \triangleq \{\ mem(x) := eval(mkminus(des(x), des(y))), pc := next(prog, pc)\}$
- $\mathfrak{S}_{10} \triangleq next_{\mathbb{U}_{10}}(\mathfrak{S}_9)$ bzgl. $\mathbb{U}_9 \triangleq \{\ pc := yes(prog, pc)\}$
- $\mathfrak{S}_{11} \triangleq next_{\mathbb{U}_{11}}(\mathfrak{S}_{10})$ bzgl. $\mathbb{U}_{11} \triangleq \{\ out := out.eval(des(r)), pc := next(prog, pc)\}$

Diskussion

Beobachtungen

- Die Aktualisierungsmengen $\mathbb{U}_1$–$\mathbb{U}_5$ führen die Aktualisierungen der ersten 5 Anweisungen durch
- Die Trägermengen bleiben erhalten
- Die Interpretation vieler Operationssymbole bleiben erhalten

Weiteres Vorgehen

- Spezifikation der Zustandsübergangsregeln
- Semantik eines Programms ist Zustandsübergangsfolge

Abstrakte Zustandsmaschinen

Definition 2.14 (Abstrakte Zustandsmaschine (engl. Abstract State Machine, kurz ASM))

Sei Σ Signatur. Eine Σ-ASM ist ein Tupel $\mathcal{A} \triangleq \langle \Sigma, \mathbb{I}, \mathbb{A}, R \rangle$ wobei

i. $\mathbb{A}$ eine Menge von Σ-Algebren ist (**Zustände**)

ii. $\mathbb{I} \subseteq \mathbb{A}$ (**Anfangszustände**)

iii. und R endliche Menge von Regeln der Form **if** φ **then** $\mathbb{U}$, φ eine prädikatenlogische Formel über Σ und $\mathbb{U}$ eine endliche Menge von Σ-Aktualisierungen ist

Die Menge $Update_{\mathcal{A}}(\mathfrak{A}) \triangleq \{u \in \mathbb{U} : \exists \textbf{if } \varphi \textbf{ then } \mathbb{U} \in R \bullet \mathfrak{A} \models \varphi\}$ heißt **Aktualisierungsmenge im Zustand** $\mathfrak{A}$.
Notation: ASM(Σ) ist die Menge aller Σ-ASMs

Beispiel 2.10: ASM $\mathcal{W}$ für einfache While-Sprache

if $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog))$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog)) \doteq true$ **then**
  $pc := yes(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

lhs : OCC × PROG →?NODE $\quad D(lhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
rhs : OCC × PROG →?NODE $\quad lhs(o,p) \doteq occ(o.0,p)$
$dest$: OCC × PROG →?NODE $\quad D(rhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
arg : OCC × PROG →?NODE $\quad rhs(o,p) \doteq occ(o.1,p)$
$cond$: OCC × PROG →?NODE $\quad D(dest(o,p)) \Leftrightarrow occ(o,p)$ **is** READ
$dest(o,p) \doteq occ(o.0,p)$

$D(arg(o,p)) \Leftrightarrow occ(o,p)$ **is** WRITE
$arg(o,p) \doteq occ(o.0,p)$
$D(cond(o,p)) \Leftrightarrow occ(o,p)$ **is** WHILE $\vee$ $occ(o,p)$ **is** IF
$cond(o,p) \doteq occ(o.0,p)$

Beispiel 2.11: Aktualisierungsmengen

if $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq true$ **then**
  $pc := yes(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

lhs : OCC × PROG →?NODE
rhs : OCC × PROG →?NODE
$dest$: OCC × PROG →?NODE
arg : OCC × PROG →?NODE
$cond$: OCC × PROG →?NODE

$D(lhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$lhs(o,p) \doteq occ(o.0,p)$
$D(rhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$rhs(o,p) \doteq occ(o.1,p)$
$D(dest(o,p)) \Leftrightarrow occ(o,p)$ **is** READ
$dest(o,p) \doteq occ(o.0,p)$
$D(arg(o,p)) \Leftrightarrow occ(o,p)$ **is** WRITE
$arg(o,p) \doteq occ(o.0,p)$
$D(cond(o,p)) \Leftrightarrow occ(o,p)$ **is** WHILE $\vee$ $occ(o,p)$ **is** IF
$cond(o,p) \doteq occ(o.0,p)$

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

Zustand $\mathfrak{I}$: $[\![mem]\!]_I(x) \triangleq \perp_{\text{VALUE}}$
$[\![inp]\!]_I \triangleq [5, 12]$
$[\![out]\!]_I \triangleq []$
$[\![pc]\!]_I \triangleq [\![().1.0.0.0.0.1]\!]_I$

- Es gilt nur $\mathfrak{I} \models occ(prog, pc) \in$ READ.

$\Rightarrow Update_{\mathcal{W}}(\mathfrak{I}) = \{ mem(x) := head(inp),$
$inp := tail(inp),$
$pc := next(prog, pc)\}$

Beispiel 2.11: Aktualisierungsmengen

if $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq true$ **then**
  $pc := yes(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

lhs : OCC × PROG →?NODE
rhs : OCC × PROG →?NODE
$dest$: OCC × PROG →?NODE
arg : OCC × PROG →?NODE
$cond$: OCC × PROG →?NODE

$D(lhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$lhs(o,p) \doteq occ(o.0,p)$
$D(rhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$rhs(o,p) \doteq occ(o.1,p)$
$D(dest(o,p)) \Leftrightarrow occ(o,p)$ **is** READ
$dest(o,p) \doteq occ(o.0,p)$
$D(arg(o,p)) \Leftrightarrow occ(o,p)$ **is** WRITE
$arg(o,p) \doteq occ(o.0,p)$
$D(cond(o,p)) \Leftrightarrow occ(o,p)$ **is** WHILE $\vee$ $occ(o,p)$ **is** IF
$cond(o,p) \doteq occ(o.0,p)$

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

Zustand $\mathfrak{S}_1 = next_{Update_{\mathcal{W}}(\mathfrak{I})}(\mathfrak{I})$:

$$[\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ \perp_{\text{VALUE}} & \text{sonst} \end{cases}$$
$[\![inp]\!]_I = [5]$
$[\![out]\!]_I = []$
$[\![pc]\!] = [\![().1.0.0.0.1]\!]_I$

- Es gilt nur $\mathfrak{S}_1 \models occ(prog, pc) \in$ READ.

$\Rightarrow Update_{\mathcal{W}}(\mathfrak{S}_1) = \{ mem(y) := head(inp),$
$inp := tail(inp),$
$pc := next(prog, pc)\}$

Beispiel 2.11: Aktualisierungsmengen

if $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog))$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog)) := head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq true$ **then**
  $pc := yes(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog) \doteq false$ **then**
  $pc := no(prog, pc)$

lhs : OCC × PROG →?NODE
rhs : OCC × PROG →?NODE
$dest$: OCC × PROG →?NODE
arg : OCC × PROG →?NODE
$cond$: OCC × PROG →?NODE

$D(lhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$lhs(o,p) \doteq occ(o.0,p)$
$D(rhs(o,p)) \Leftrightarrow occ(o,p)$ **is** ASSIGN
$rhs(o,p) \doteq occ(o.1,p)$
$D(dest(o,p)) \Leftrightarrow occ(o,p)$ **is** READ
$dest(o,p) \doteq occ(o.0,p)$
$D(arg(o,p)) \Leftrightarrow occ(o,p)$ **is** WRITE
$arg(o,p) \doteq occ(o.0,p)$
$D(cond(o,p)) \Leftrightarrow occ(o,p)$ **is** WHILE $\vee$ $occ(o,p)$ **is** IF
$cond(o,p) \doteq occ(o.0,p)$

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

Zustand $\mathfrak{S}_2 = next_{Update_{\mathcal{W}}(\mathfrak{S}_1)}(\mathfrak{S}_1)$:

$$[\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ \perp_{\text{VALUE}} & \text{sonst} \end{cases}$$
$[\![inp]\!]_I = []$
$[\![out]\!]_I = []$
$[\![pc]\!] = [\![().1.0.0.1]\!]_I$

- Es gilt nur $\mathfrak{S}_2 \models occ(prog, pc) \in$ ASSIGN.

$\Rightarrow Update_{\mathcal{W}}(\mathfrak{S}_2) = \{ mem(r) := 1,$
$pc := next(prog, pc)\}$

Beispiel 2.11: Aktualisierungsmengen

if $occ(pc, prog)$ **is** ASSIGN **then**
  $mem(lhs(pc, prog)) := eval(rhs(pc, prog)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** READ **then**
  $mem(dest(pc, prog))$ $:= head(inp)$
  $inp := tail(inp)$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WRITE **then**
  $out := out.eval(arg(pc, prog))$
  $pc := next(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog)) \doteq true$ **then**
  $pc := yes(prog, pc)$
if $occ(pc, prog)$ **is** WHILE $\wedge$ $eval(cond(pc, prog)) \doteq false$ **then**
  $pc := no(prog, pc)$

lhs :	OCC $\times$ PROG	$\to$	?NODE
rhs :	OCC $\times$ PROG	$\to$	?NODE
$dest$:	OCC $\times$ PROG	$\to$	?NODE
arg :	OCC $\times$ PROG	$\to$	?NODE
$cond$:	OCC $\times$ PROG	$\to$	?NODE

$D(lhs(o,p))$	$\Leftrightarrow$	$occ(o,p)$	**is** ASSIGN
$lhs(o,p)$	$\doteq$	$occ(o.0,p)$	
$D(rhs(o,p))$	$\Leftrightarrow$	$occ(o,p)$	**is** ASSIGN
$rhs(o,p)$	$\doteq$	$occ(o.1,p)$	
$D(dest(o,p))$	$\Leftrightarrow$	$occ(o,p)$	**is** READ
$dest(o,p)$	$\doteq$	$occ(o.0,p)$	
$D(arg(o,p))$	$\Leftrightarrow$	$occ(o,p)$	**is** WRITE
$arg(o,p)$	$\doteq$	$occ(o.0,p)$	
$D(cond(o,p))$	$\Leftrightarrow$	$occ(o,p)$	**is** WHILE $\vee$ $occ(o,p)$ **is** IF
$cond(o,p)$	$\doteq$	$occ(o.0,p)$	

```
int x;
int y;
int r;
read(x);
read(y);
r=1;
while (x>=y) {
 r=r+1;
 x=x-y;
}
print(r);
```

Zustand $\mathfrak{S}_3 = next_{Update_{\mathcal{W}}(\mathfrak{S}_2)}(\mathfrak{S}_2)$:

$$[\![mem]\!]_I(id) = \begin{cases} 12 & \text{falls } id = x \\ 5 & \text{falls } id = y \\ 1 & \text{falls } id = r \\ \bot_{\text{VALUE}} & \text{sonst} \end{cases}$$

$[\![inp]\!]_I = []$
$[\![out]\!]_I = []$
$[\![pc]\!] = [\![().1.0.1]\!]_I$

- Es gilt $\mathfrak{S}_3 \models occ(prog, pc) \in$ WHILE und $\mathfrak{S}_3 \models eval(\texttt{x>=y}) \doteq true$

$\Rightarrow$ $Update_{\mathcal{W}}(\mathfrak{S}_3) = \{\ pc := yes(prog, pc)\}$

Übung Bestimmen Sie die Aktualisierungsmengen der Zustände $\mathfrak{S}_{i+1} = next_{Update_{\mathcal{W}}(\mathfrak{S}_i)}(\mathfrak{S}_i)$, $i = 4, \ldots$.

Makros und Notationen zur besseren Lesbarkeit

Beobachtungen

- Einige Aktualisierungen und Terme werden häufig benutzt
- Einige Terme sind unübersichtlich

$\Rightarrow$ Einführung von Abkürzungen

- Bei der Semantik einer while-Schleife würde eine "lokale" Regel einsichtiger sein

Beispiel 2.12: Abkürzungen

if CT **is** ASSIGN **then**
  $mem(Dest) := eval(Src)$
  $Proceed$
if CT **is** READ **then**
  $mem(Des) := head(inp)$
  $inp := tail(inp)$
  $Proceed$
if CT **is** WRITE **then**
  $out := out.eval(Expr)$
  $Proceed$
if CT **is** WHILE **then**
  if $eval(Cond) \doteq true$ **then**
    $pc := yes(prog, pc)$
  else $pc := no(prog, pc)$

$Proceed \triangleq pc := next(prog, pc)$
$CT \triangleq occ(pc, prog)$
$Dest \triangleq lhs(pc, prog)$
$Src \triangleq rhs(pc, prog)$
$Des \triangleq dest(pc, prog)$
$Expr \triangleq arg(pc, prog)$
$Cond \triangleq cond(pc, prog)$

if φ **then**
  if ψ **then** U_1 $\triangleq$ **if** $\varphi \wedge \psi$ **then** U_1
else U_2 **if** $\varphi \wedge \neg\psi$ **then** U_2

Abstrakte Zustandsmaschinen: Interpretation als Zustandsübergangssystem

Satz 2.1 (ASM und Zustandsübergangssystem)

Sei Σ eine Signatur, $\mathcal{A} \triangleq \langle \Sigma, \mathbb{I}, \mathbb{A}, R \rangle$ eine Σ-ASM. Dann ist $\mathcal{A}$ ein Zustandsübergangssystem $\mathcal{S} \triangleq (Q, I, \to_R)$ mit $Q \triangleq \mathbb{A}$, $I \triangleq \mathbb{I}$ und $\to_R \triangleq \{(\mathfrak{A}_1, \mathfrak{A}_2) : \mathfrak{A}_1 \in \mathbb{A} \wedge \mathfrak{A}_2 = next_{Update_{\mathcal{A}}(\mathfrak{A}_1)}(\mathfrak{A})\}$. Eine Σ-Algebra $\mathfrak{F} \in \mathbb{A}$ ist genau dann final, wenn $Update_{\mathcal{A}}(\mathfrak{F})$ inkonsistent oder $Update_{\mathcal{A}}(\mathfrak{F}) = \emptyset$ ist.

Beweis

Folgt direkt aus Definition 1.1, Definition 2.13, $\to_R \sqsubseteq \mathbb{A} \times \mathbb{A}$ und Definition 2.14.

Diskussion

Beobachtung

Alle Begriffe aus Kapitel 1 sind für ASMs definiert:

- (Vollständiger) Lauf einer ASM
- Abstraktion einer ASM
- Beobachtbares Verhalten
- ASM $\mathcal{A}_1$ 1-1-simuliert (eingeschränkt) ASM $\mathcal{A}_2$
- Wenn $[\![\cdot]\!] : \mathcal{T}_{\text{PROG}} \to \langle \Sigma, \text{ASM}(\Sigma) \rangle$ den Programmen einer Programmiersprache eine Semantik als ASM zuordnet, ist auch der Begriff der korrekten Übersetzung bzw. des korrekten Übersetzers definiert.
- (Eingeschränkte) n-m-Simulation

Folgerung

Auch die Sätze für Zustandsübergangssystem gelten für ASMs

- Horizontale Dekomposition
- Vertikale Dekomposition
- Satz 1.3 (Voraussetzung, dass n-m-Simulation eine eingeschränkte 1-1-Simulation auf beobachtbarem Verhalten impliziert)

Statische und dynamische Funktionen

Zusammenfassung des Beispiels (außer abstrakter Syntax, Navigation und Basistypen)

if CT **is** ASSIGN **then**
  $mem(Dest) := eval(Src)$
  $Proceed$
if CT **is** READ **then**
  $mem(Des) := head(inp)$
  $inp := tl(inp)$
  $Proceed$

if CT **is** WRITE **then**
  $out := out.eval(Expr)$
  $Proceed$
if CT **is** WHILE **then**
  if $eval(Cond) \doteq true$ **then**
    $pc := yes(pc)$
  else $pc := no(pc)$

$$
\begin{array}{rcl}
Proceed & \triangleq & pc := next(prog, pc) \\
CT & \triangleq & occ(pc, prog) \\
Dest & \triangleq & lhs(pc, prog) \\
Src & \triangleq & rhs(pc, prog) \\
Des & \triangleq & dest(pc, prog) \\
Expr & \triangleq & arg(pc, prog) \\
Cond & \triangleq & cond(pc, prog)
\end{array}
$$

$$
\begin{array}{rcl}
lhs(o,p) & \doteq & occ(o.0,p) \\
rhs(o,p) & \doteq & occ(o.1,p) \\
dest(o,p) & \doteq & occ(o.0,p) \\
arg(o,p) & \doteq & occ(o.0,p) \\
cond(o,p) & \doteq & occ(o.0,p)
\end{array}
$$

$$
\begin{array}{rcl}
eval(mkdes(mkname(x))) & \doteq & mem(x) \\
eval(mkintconst(n)) & \doteq & n \\
eval(mkboolconst(b)) & \doteq & b \\
eval(mkgeq(e_1,e_2)) & \doteq & intgeq(eval(e_1), eval(e_2)) \\
eval(mkplus(e_1,e_2)) & \doteq & intplus(eval(e_1), eval(e_2)) \\
eval(mkminus(e_1,e_2)) & \doteq & intminus(eval(e_1), eval(e_2))
\end{array}
$$

- Die Funktionen *mem*, *pc*, *inp*, *out*, *eval* und *prog* sind dynamisch.
  alle anderen Funktionen sind statisch.
- *prog* ist konstant; die übrigen dynamischen Funktionen sind nicht konstant.
- Die Funktionen *mem*, *pc*, *inp*, *out* sind direkt; *eval* ist abgeleitet.

Definition 2.15 (Statische und dynamische Funktionen)

Sei $\Sigma \triangleq \langle S, \sqsubseteq, F, F' \rangle$ eine Signatur und $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R \rangle$, so dass $\mathbb{A}$ die Menge der von $\mathbb{I}$ erreichbaren Zustände ist. Eine Funktion $f \in F \cup F'$ heißt **dynamisch** gdw. wenn es $\mathfrak{A}, \mathfrak{A}'$ gibt, so dass $[\![f]\!]_{\mathfrak{A}} \neq [\![f]\!]_{\mathfrak{A}'}$ ist. Ansonsten heißt $f \in F \cup F'$ heißt **statisch**. Eine dynamische Funktion f heißt **konstant** gdw. für jeden Initialzustand $\mathfrak{I} \in \mathbb{I}$ die Eigenschaft $[\![f]\!]_{\mathfrak{A}} = [\![f]\!]_{\mathfrak{I}}$ für alle von $\mathfrak{I}$ aus erreichbaren Zustände $\mathfrak{A}$ gilt. Eine nicht-konstante dynamische Funktion f heißt **direkt**, wenn es eine Regel **if** φ **then** U und Terme $t_1, \ldots, t_n, t$ gibt, so dass $f(t_1, \ldots, t_n) := t \in U$ ist. Andernfalls heißt f **abgeleitet**.

Diskussion

Konvention

Ab sofort betrachten wir noch ASMs $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R \rangle$, bei denen jedes $\mathfrak{A} \in \mathbb{A}$ von einem Initialzustand $\mathfrak{I} \in \mathbb{I}$ erreichbar ist.

Korollar 2.2 (Eigenschaften von Zuständen einer ASM)

Sei $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R \rangle$ eine ASM, $\Delta \subseteq \Sigma$ die Signatur der statischen Funktionen und Γ die Signatur der konstanten Funktionen von $\mathcal{A}$. Dann gilt:

i. $\mathfrak{A}|_\Delta = \mathfrak{A}'|_\Delta$ für alle $\mathfrak{A}, \mathfrak{A}' \in \mathbb{A}$ (**statische Algebra**)

ii. Für jedes $\mathfrak{I} \in \mathbb{I}$ gilt: $\mathfrak{A}|_{\Delta \cup \Gamma} = \mathfrak{I}|_{\Delta \cup \Gamma}$ für alle von $\mathfrak{I}$ aus erreichbaren Zustände $\mathfrak{A}$.

Beobachtungen

- Bei Erstellung einer ASM sollten direkte und abgeleitete Funktionen strikt voneinander getrennt werden.

⇒ Keine Axiome für direkte dynamische Funktionen

⇒ Aus den Axiomen für abgeleitete dynamische Funktionen dürfen keine Aussagen über direkte dynamische Funktionen gefolgert werden können

☞ Gefahr von inkonsistenten Spezifikationen

Zusammenfassung

- Abstrakte Zustandsmaschinen sind Zustandsübergangssysteme
- Zustände sind Σ-Algebren über einer Signatur Σ
- Zustandsübergangsregeln definieren die Änderung der Interpretation von Funktionssymbolen
- Initialzustände und die statischen Algebren werden über abstrakte Datentypen spezifiziert.
- Abgeleitete dynamische Funktionen werden ebenfalls durch Gleichungen definiert.

2.4 Modulare Konstruktion von Semantiken mit ASMs

Ziele

- Formale Definition einer Semantik von ASMs ausgehend von einer Kernsprache, die dann sukzessive erweitert wird
- Erweiterungen sollen keine Änderungen der bisherigen Zustandsübergangsregeln und Axiome erfordern

⇒ Auch Korrektheitsbeweise können entsprechend modular aufgebaut werden.

1. Allgemeines Vorgehen
2. Die Sprache C--
3. Steuerstrukturen (C0)
4. Verbunde und Verbundzeiger (C1)
5. Prozeduren und Funktionen

2.4.1 Allgemeines Vorgehen

Schritte

- Definition der abstrakten Syntax
- Definition der Navigation
- Definition des Zustandsraums
- Definition der Zustandsübergangsregeln

Modularisierung

- Schrittweise Erweiterung der abstrakten Syntax
- Schrittweise Erweiterung der Navigation
- Schrittweise Erweiterung der Definition des Zustandsraums
- Schrittweise Erweiterung der Zustandsübergangsregeln

Diskussion

Probleme

- Bei der Definition des Zustandsraums müssen manchmal auch zukünftige Erweiterungen berücksichtigt werden.
- Wenn neue Sprachkonzepte eingeführt werden, müssen manchmal auch für alte Sprachkonstrukte neue Zustandsübergänge eingeführt werden.

Beispiel 2.13: Probleme beim Erweitern der einfachen While-Sprache

- Ausdrucksauswertung mit *eval* kann so nicht mehr formuliert werden, wenn Funktionsaufrufe eingeführt werden
- ⇒ Jeder Ausdruck hat einen Wert und diese Zuordnung ist zu definieren.
- Bei Einführung von Prozeduren und zusammengesetzten Datentypen können mehrere Objekte demselben Namen zugeordnet sein
- ⇒ Speicher $mem : \text{ID} \rightarrow \text{VALUE}$ funktioniert so nicht
- ⇒ Bindung von Variablen an Adressen und Adressen an Werte: $bind : \text{ID} \rightarrow \text{LOC}$ und $mem : \text{LOC} \rightarrow \text{VALUE}$
- Bei Einführung von Ausnahmen muss bei Zuweisung etc. die Fortsetzung zur Ausnahmebehandlung definiert werden.
- ☞ Ohne Ausnahmebehandlung endet man in finalem Zustand bei Auftreten einer Ausnahme (z.B. Division durch 0)

Lösungen

- Allgemeinere Konzepte zur Definition des Zustandsraums
- Neue Zustandsübergänge für alte Befehle durch neue Regeln
- ☞ Bei alten Befehlen werden die alten und neuen Aktualisierungen vereinigt

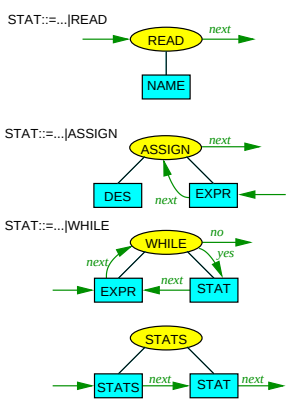
Ausblick

Weiteres Vorgehen in diesem Abschnitt

- Informelle Vorstellung eines reduzierten C-Dialekts
- Definition des Zustandsraums
- Bei der Definition der Semantik wird die informelle Definition, die abstrakte Syntax, die Navigation und die Zustandsübergangsregel(n) eingeführt

Notationen

- Die abstrakte Syntax wird als Baum notiert. Die Knoten werden mit den entsprechenden Sorten gekennzeichnet. Rechtecke stehen für komplette Unterbäume. Alternativen werden als Produktion angegeben.
- Navigationen sind von der Sorte OCC und werden als Pfeile notiert.
- Jeder Unterbaum hat einen Anfang und mögliche Ausgänge, die durch ein– bzw. ausgehende Pfeile notiert werden.
- Eine Navigation auf einen kompletten Unterbaum hat als Ziel dessen Eingang
- Eine Navigation aus einem kompletten Unterbaum hat als Quellen alle Ausgänge des Unterbaums



2.4.2 Die Sprache C--

Programme

prog ::= *decl** *block*

- Die in den Deklarationen vereinbarten Namen sind im Hauptprogram *block* sichtbar (**Gültigkeitsbereich**)
- Die in den Deklarationen vereinbarten Namen müssen eindeutig sein.
- Die in den Deklarationen vereinbarten Variablen, Prozeduren und Typen heißen **global**.
- Die Ausführung eines Programms führt den Block aus.

Basistypen und Anpassung

type ::= **id**

- Basisdatentypen sind ganze Zahlen (Typ `int`), Gleitkommazahlen (Typ `float`) und Wahrheitswerte (Typ `bool`) mit den üblichen Operationen und Konstanten
- Die Darstellung ganzer Zahlen entspricht der Darstellung ganzer Zahlen auf der Zielmaschine. Die Arithmetik ist Ringarithmetik (z.B ist $\texttt{maxint} + 1 = \texttt{minint}$)
- Die Gleitkommazahlen werden gemäß IEEE 754-1985 dargestellt. Die Genauigkeit entspricht der Wortlänge der Zielmaschine.
- Ganze Zahlen können automatisch in Gleitkommazahlen konvertiert werden.
- Konstanten werden in der selben Syntax wie C notiert

Deklarationen

Zusammengesetzte Typen

decl ::= *vardecl*|*typedecl*|*procdecl*
typedecl ::= *struct*|*class*
struct ::= `struct` **id** {*vardecl**}
class ::= `class` **id** {*vardecl**}

- Ein Verbundtyp (`struct`) definiert ein kartesisches Produkt. Verbundobjekte (`struct`) bestehen aus Teilobjekten, die durch die Variablendeklarationen des Verbundtyps gegeben sind (**Verbundfelder**)
- Ein Klassentyp (`class`) definiert einen Zeigertyp auf Verbundtypen. Ein Klassenobjekt ist ein Zeiger auf ein Verbundobjekt, dessen Teilobjekte durch die Verbundfelder des Klassentyps definiert sind. Zeiger können auch leer sein (`NULL`)
- **id** ist jeweils der Name des Verbund– bzw. Klassentyps.
- Die Verbundfeldnamen müssen paarweise verschieden sein.

Variablendeklarationen

vardecl ::= *type* **id**;

- Eine Variablendeklaration vereinbart ein Objekt, in dem Werte des Typs *type* gespeichert werden können
- Der Bezeichner **id** ist der Name der Variablen. Mit diesem Namen kann die Variable in ihrem Gültigkeitsbereich angesprochen werden.

Zugriffspfade

des ::= **id**|*fieldaccess*
fieldaccess ::= *des*.**id**

- Zugriffspfade verweisen auf Objekte
- Falls ein Zugriffspfad aus einem Bezeichner **id** besteht, muss dieses Objekt im Kontext durch eine Variablendeklaration vereinbart sein. In diesem Fall verweist der Zugriffspfad auf dieses Objekt.
- Bei einem Feldzugriff muss der Typ des Zugriffspfads *des* ein Verbund– oder Klassentyp sein, der ein Feld mit dem Namen **id** enthält.
- Falls *des* Verbundtyp ist, bezeichnet der Zugriffspfad das durch **id** definierte Teilobjekt
- Falls *des* Klassentyp ist, muss der Wert von *des* $\neq$ `NULL` sein, ansonsten bricht das Programm mit einer `NullPointerDereferenceException` ab.
- Wenn der Wert von *des* $\neq$ `NULL` ist, wird zunächst das durch den Zeiger verwiesene Objekt ermittelt (**Dereferenzieren**). Der Klassenfeldzugriff verweist dann auf das durch **id** definierte Teilobjekt.

Ausdrücke (Syntax und statische Semantik)

expr ::= **null**|**intconst**|**fltconst**|**boolconst**|*des*|*unexpr*|*binexpr*|*call*|*new*|(*expr*)
unexpr ::= *unop expr*
unop ::= `!`|`-`|`+`
binexpr ::= *expr binop expr*
unop ::= *eqop*|*relop*|*addop*|*mulop*
eqop ::= `==`|`!=`
relop ::= `<`|`>`|`<=`|`>=`
addop ::= `+`|`-`
mulop ::= `*`|`/`|`%`
boolop ::= `||`|`&&`
new ::= `new` *type*

- Die Prioritäten der Operatoren sind in der folgenden Reihenfolge definiert: unäres + und -, *mulop*, *addop*, *relop*, *eqop*, `&&`, `||`, `!`.
- Die binären Operatoren (außer `relop`) sind linksassoziativ geklammert.
- Der Typ eines Zugriffspfads ist der Typ des durch ihn bezeichneten Objekts.
- Boolesche Operatoren erwarten von ihren Operanden den Typ `bool`. Boolesche Ausdrücke haben den Typ `bool`.
- Gleichheitsausdrücke erwarten entweder von beiden Operanden denselben Typ oder einer der Operanden muss vom Typ `int` und der andere vom Typ `bool` sein. Der Typ eines Gleichheitsausdrucks ist `bool`.
- Der Operator % erwarten von beiden Operanden den Typ `int`. Der Ausdruck ist vom Typ `int`.
- Relationale Ausdrücke (*relop*) und arithmetische Ausdrücke (außer %) erwarten von ihren Operanden den Typ `int` oder `float`. Der Typ eines relationalen Ausdrucks ist `bool`. Der Typ eines arithmetischen Ausdrucks ist `int`, wenn alle seine Operanden vom Typ `int` sind, ansonsten ist der Typ `float`.
- Der Typ einer `new`-Operation muss eine Klasse sein. Dieser Typ ist der Typ des Ausdrucks

Ausdrücke (dynamische Semantik)

- Ausdrücke werden ausgewertet. Die Auswertereihenfolge arithmetischer und relationaler Ausdrücke ist implementierungsabhängig.
- Boolesche Ausdrücke werden durch Kurzauswertung ausgewertet.
- Der Wert einer Konstanten entspricht dem im Programm bezeichneten Wert
- Der Wert eines Zugriffspfad *des* ist der Inhalt des Objekts, auf das *des* verweist.
- Der Wert eines unärer Ausdrucks wendet den Operator auf den Wert des Operanden an (`!` ist die logische Negation)
- Die Werte binärer Ausdrücke verknüpfen die beiden Werte mit dem Operator des Ausdrucks (`||` ist die logische Disjunktion, `&&` die logische Konjunktion und % der Rest bzgl. der Division)
- Hat der rechte Operand von / oder % den Wert 0, dann bricht das Programm mit der Ausnahme `DivByZeroException` ab.
- Eine `new`-Operation erzeugt ein Objekt der Klasse und liefert einen Verweis auf dieses Objekt

Anweisungen

Zuweisung

stat ::= *assign*|*read*|*write*|*if*|*while*|*break*|*continue*|*call* ;|*return*|*block*
assign ::= *des* = *expr* ;

- Der Typ von *expr* muss an den Typ *des* anpassbar sein.
- Der Zugriffspfad wird zu einem Verweis auf ein Objekt ausgewertet und der Ausdruck wird ausgewertet. Der Wert des Ausdrucks wird in das durch den Zugriffspfad verwiesene Objekt geschrieben. Die Auswertereihenfolge ist implementierungsabhängig.
- Danach wird mit der nächsten Anweisung fortgefahren.
- Wird an ein Verbundobjekt zugewiesen, dann werden die Werte der Verbundfelder in das Teilobjekt, auf das entsprechende Verbundfeld verweist, geschrieben. Wird an ein Verbundzeigerobjekt zugewiesen wird der Zeiger in das Objekt, auf das die linke Seite verweist, geschrieben.

Anweisung(en)

Lese-Anweisung

read ::= `read` `(` *des* `)` `;`

- Der Zugriffspfad darf kein Verbund– oder Klassentyp haben.
- Die Lese-Anwweisung bestimmt das Objekt, auf das der Zugriffspfad verweist und schreibt dort den aktuellen Wert des Eingabestroms. Die aktuelle Position des Eingabestroms auf den nächsten Wert fortgeschalten.
- Danach wird mit der nächsten Anweisung fortgefahren.

Schreibe-Anweisung

write ::= `write` `(` *expr* `)` `;`

- Der Ausdruck darf kein Verbund– oder Klassentyp haben.
- Die Schreibe-Anwweisung wertet den Ausdruck aus und schreibt dessen Wert an das Ende des Ausgabestroms.
- Danach wird mit der nächsten Anweisung fortgefahren.

Anweisung(en)

Verzweigung

if ::= `if` `(` *expr* `)` *stat* [`else` *stat*]

- Der Ausdruck muss vom Typ `bool` sein.
- Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die erste Anweisung ausgeführt. Danach wird die Anweisung nach der Verzweigung ausgeführt. Ist das Ergebnis `false` und keine `else`-Anweisung vorhanden, dann wird die Anweisung nach der Verzweigung ausgeführt. Ansonsten wird zuerst die `else`-Anweisung und dann die Anweisung nach der Verzweigung ausgeführt.

Schleifen

if ::= `while` `(` *expr* `)` *stat*

- Der Ausdruck muss vom Typ `bool` sein.
- Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die Anweisung ausgeführt. Danach wird wieder die Schleife ausgeführt.Ist das Ergebnis `false`, dann wird die Anweisung nach der Schleife ausgeführt.

Anweisung(en)

Schleifenabbruch und –fortsetzung

break ::= `break` `;`
continue ::= `continue` `;`

- Eine `break`– oder `continue`-Anweisung darf nur innerhalb des Blocks einer Schleifenanweisung stehen.
- Bei Ausführen einer `break`-Anweisung wird die Schleife abgebrochen und danach die Anweisung nach der Schleife ausgeführt.
- Bei Ausführen einer `continue`-Anweisung wird die aktuelle Ausführung des Schleifenrumpfs abgebrochen und danach die Schleifenanweisung ausgeführt.

Blockanweisung

block ::= { *vardecl** *stat** }

- Die durch die Variablendeklarationen eingeführten Namen eines Blocks müssen paarweise verschieden sein.
- Die Namen der deklarierten Variablen sind nur im Block sichtbar. Sie verdecken Bezeichner desselben Namens, die außerhalb des Blocks sichtbar sind.
- Die Ausführung eines Blocks führt alle seine Anweisungen nacheinander aus. Danach wird die Anweisung nach dem Block ausgeführt.

Prozeduren und Funktionen (Syntax und statische Semantik)

procdecl ::= *type* **id** (*pars*) *block*
pars ::= [*type* **id** (, *type* **id**)*]
call ::= **id** (*args*)
args ::= [*expr* (, `expr`)*]
return ::= `return` [*expr*]

- Die Parameter sind im Prozedurrumpf (*block*) sichtbar.
- Die Namen der Parameter müssen paarweise verschieden sein.
- Der Typ in einer Prozedurdeklaration heißt **Rückgabetyp**. Ist der Rückgabetyp `void` so redet man von **Prozeduren**, sonst von **Funktionen**.
- Prozeduraufrufe sind Anweisungen, Funktionsaufrufe sind Ausdrücke. Der Typ eines Funktionsaufrufs ist der Rückgabetyp der aufgerufenen Funktion.
- Bei einem Aufruf einer Prozedur/Funktion muss die Anzahl der Parameter der aufgerufenen Prozedur **id** mit der Anzahl der Argumente übereinstimmen. Vom *i*-ten Argument wird der Typ des *i*-ten Parameters erwartet.
- Eine `return`-Anweisung darf nur innerhalb eines Prozedur– oder Funktionsrumpfs vorkommen. Bei einer Prozedur darf nur die `return`-Anweisung ohne Ausdruck verwendet werden. Bei einer Prozedur muss die `return`-Anweisung mit Ausdruck verwendet werden. Von diesem Ausdruck wird erwartet, dass er Typ gleich dem Rückgabetyp der Funktion ist.

Prozeduren und Funktionen (dynamische Semantik)

- Mit dem Prozedur–/Funktionsaufruf werden den Parametern Objekte zugeordnet.
- Bei einem Prozedur– oder Funktionsaufruf werden die Argumente ausgewertet und an die Parameter positional übergehen, d.h. der Wert eines Arguments wird in das Objekt geschrieben, auf das der Parameter verweist. Danach wird der Prozedurrumpf ausgeführt. Die Reihenfolge der Auswertung der Argumente ist implementierungsabhängig.
- Ein Prozedur– oder Funktionsaufruf **kehrt zurück**, wenn die letzte Anweisung des Prozedurrumpfs oder eine `return`-Anweisung ausgeführt wurde.
- Nach Rückkehr von einem Prozeduraufruf wird mit die Anweisung nach dem Aufruf ausgeführt.
- Nach Rückkehr von einem Funktionsaufruf wird mit der Ausführung der Anweisung, in der der Aufruf vorkam, fortgefahren. Der Wert eines Funktionsaufrufs ist der Wert des Ausdrucks der ausgeführten `return`-Anweisung.

2.4.3 Steuerstrukturen (C0)

Ausgangspunkt

- Spezifikationen INT für ganze Zahlen (64-Bit, 2-Komplement, Ringarithmetik
- FLOAT für Gleitkommazahlen (64-Bit 754-1985)
- Spezifikation BOOL für Wahrheitswerte
- **spec** VALUE **extends** INT, FLOAT, BOOL
  sorts VALUE
  subsorts INT $\sqsubseteq$ VALUE, FLOAT $\sqsubseteq$ VALUE, BOOL $\sqsubseteq$ VALUE
- Spezifikation OCC für Listen ganzer Zahlen
- Spezifikation LISTOFVAL für Listen von Werten
- Spezifikation PROG **extends** OCC, LISTOFVAL $\cdots$, die die abstrakte Syntax, Attributierung und Navigation definiert

Vorgehen

- Definition des Zustandsraums und des Anfangszustands
- Graphische Angabe der abstrakten Syntax und der Navigationen
- Pro Konzept eine Zustandsübergangsregel

Zustandsraum

Umgebung und Speicher

- **Umgebungen** binden Variablenbezeichner an Adressen:
  spec ENV **extends** PROG
  sorts ENV, LOC, IDLIST
  operations

mkenv :		$\to$ ENV	erzeugt leere Umgebung
newbind :	ENV $\times$ ID $\times$ LOC	$\to$?ENV	neue Zuordnung
isUsed	ENV $\times$ LOC	$\to$ BOOL	wahr gdw. Adresse gebunden
isBound	ENV $\times$ ID	$\to$ BOOL	prüft Bindung
bind :	ENV $\times$ ID	$\to$?LOC	ermittelt Bindung
delbind :	ENV $\times$ ID	$\to$ ENV	beseitigt Bindung
$(++)$:	ENV $\times$ ENV	$\to$ ENV	Anhängen zweier Umgebungen
$(\setminus)$:	ENV $\times$ IDLIST	$\to$ ENV	

  axioms

$$D(newbind(e,x,l)) \Leftrightarrow isUsed(e,l) \doteq false$$
$$D(bind(e,x)) \Leftrightarrow isBound(e,x) \doteq true$$
$$bind(newbind(e,x,l),x) \doteq l$$
$$\neg x \doteq y \Rightarrow bind(newbind(e,y,l),x) \doteq bind(e,x)$$
$$delbind(mkenv,x) \doteq mkenv$$
$$delbind(newbind(e,x,l),x) \doteq e$$
$$\neg x \doteq y \Rightarrow delbind(newbind(e,y,l),x) \doteq newbind(delbind(e,x),y,l)$$

Übrige Axiome sind zur Übung überlassen

- $\text{ENV} \models D(bind(e,x)) \wedge D(bind(e,y)) \wedge bind(e,x) \doteq bind(e,y) \Rightarrow x \doteq y$
  $\text{ENV} \models isBound(e',x) \doteq false \Rightarrow bind(e'{+}{+}e,x) \doteq bind(e,x)$
  $\text{ENV} \models isBound(e',x) \doteq true \Rightarrow bind(e'{+}{+}e,x) \doteq bind(e',x)$
- **Speicher** binden Werte an Adressen: *mem* : LOC $\to$?VALUE
- $e \setminus i$ beseitigt die Bindungen der Bezeichner in Liste i aus e_1

Zustandsraum und Anfangszustand

Zustand

Ein Zustand besteht aus einer Umgebung, einem Speicher, einem Eingabe– und Ausgabestrom, dem Programm sowie einem Befehlszeiger:

spec STATE **extends** ENV
operations

env :		→	ENV
mem :	LOC	→?	VALUE
inp :		→	LISTOFVAL
out :		→	LISTOFVAL
prog :		→	PROG
pc :		→	OCC

Anfangszustand

Am Anfang ist der Speicher leer, der Ausgabestrom leer, die Umgebung leer und der Befehlszeiger verweist auf die erste auszuführende Anweisung

spec INITSTATE **extends** STATE
axioms

$$\neg D(mem(x))$$
$$out \doteq []$$
$$env \doteq mkenv$$
$$pc \doteq first(prog)$$

⇒ Bezug auf Attribute des attributierten Strukturbaums

- Die globalen Variablen des Hauptprogramms sind bereits zur Übersetzungszeit an Objekte gebunden

⇒ Statische Funktion *globenv* : PROG → ENV

Ausdrucksauswertung

Problem

Definition mit *eval* ist wegen Funktionsaufrufen nicht erweiterbar

⇒ Jeder Operator ist auszuwertender Befehl
⇒ Ausdruckswert wird Knoten zugeordnet
⇒ Direkte dynamische Funktion *val* : OCC →?VALUE

- Zugriffspfade bestimmen zusätzlich den Ort des Objekts, auf das sie verweisen: dynamische Funktion *loc* : OCC →?LOC

Beobachtungen

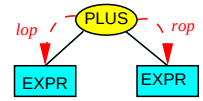
- Ausdrücke benötigen die Werte ihrer Operanden
- Schleifen und Verzweigungen benötigen den Wert ihrer Bedingung
- Die Schreibanweisung benötigt den Wert ihres Argument

⇒ Einführung von Datenflusskanten analog der Steuerflusskanten:

lop :	PROG × OCC	→?OCC	linker Operand eines Ausdrucks
rop :	PROG × OCC	→?OCC	rechter Operand eines Ausdrucks
opd :	PROG × OCC	→?OCC	Operand eines unären Ausdrucks
cond :	PROG × OCC	→?OCC	Bedingung einer Schleife oder Verzweigung
arg :	PROG × OCC	→?OCC	Auszudruckender Ausdruck

☞ Use-Def-Beziehungen

- Notation durch gestrichelte Pfeile:
- Eine Datenflusskante auf einen kompletten Unterbaum hat als Ziel dessen Wurzel
- Die Quelle einer Datenflusskante ist immen ein AST-Knoten



Ausdrucksauswertereihenfolgen

Problem

Spezifikation der implementierungsabhängigen Reihenfolge

⇒ Aus Sicht der Sprachsemantik sind alle Reihenfolgen zulässig, nur die linken Operanden von AND bzw. OR müssen vor den rechten ausgewertet werden

⇒ Nur der Datenfluss muss eingehalten werden

Beobachtung

Eine Programmiersprache definiert eine Halbordnung, welche Operatoren vor anderen Operatoren berechnet werden müssen.

⇒ Jede topologische Sortierung dieser Halbordnung ist eine gültige Reihenfolge

☞ Bei strenger Links-Rechts-Auswertereihenfolge (wie z.B. in Java oder C#) ist die Halbordnung sogar eine Totalordnung.

Ausdrucksauswertereihenfolgen

Formalisierung

- Wenn ein Ausdruck ausgewertet werden soll, wird irgendein Knoten k gewählt, der minimal bzgl. der Halbordnung ist.
- Nichtdeterministische Auswahl einer topologischen Sortiertung, die k als erstes Element enthält
- Abarbeitung dieser topologischen Sortierung

⇒ Weitere Form von Zustandsübergangsregeln erforderlich: **choose** x : S • φ **from** UJede Menge $\mathbb{U} \triangleq U[t/x]$ von Aktualisierungen mit $\mathfrak{A} \models \varphi[t/x]$ ist eine Aktualisierungsmenge im Zustand $\mathfrak{A}$

ASM-Zustandsübergangsregel

- Dynamische Funktion *toBeEval* :→ LISTOFOCC
- Falls ein Ausdruck das erste Mal betreten wird, wird nichtdeterministisch eine topologische Sortierung gewählt:

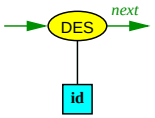
if $next(pc, prog)$ **is** EXPR **then**
  choose o : OCC • $isMin(o, next(pc, prog))$ **from** $pc := o$
  choose l : LISTOFOCC • $isTopOrder(l, next(pc, prog))$ **from** $toBeEval := l$

Bemerkung

Wir betrachten hier der Einfachheit halber eine Links-Rechts-Auswertereihenfolge

Zugriffspfade

- Falls ein Zugriffspfad aus einem Bezeichner **id** besteht, muss dieses Objekt im Kontext durch eine Variablendeklaration vereinbart sein. In diesem Fall verweist der Zugriffspfad auf dieses Objekt.
- Der Wert eines Zugriffspfad *des* ist der Inhalt des Objekts, auf das *des* verweist.



if CT **is** DES **then**
  $loc(pc) := Loc$
  $Coerce(val(pc), mem(Loc), pc)$
  $Proceed$

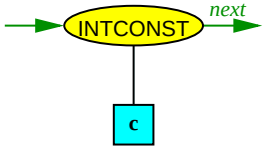
$$
\begin{array}{ll}
CT & \triangleq occ(prog, pc) \\
Env & \triangleq env{+}{+}globenv(prog) \\
Loc & \triangleq bind(Env, id(pc, prog)) \\
Pri(p) & \triangleq pri(prog, p) \\
Post(p) & \triangleq post(prog, p) \\
Coerce(src, dest, p) & \triangleq dest := coerce(src, Pri(p), Post(p)) \\
Proceed & \triangleq pc := next(prog, pc)
\end{array}
$$

Statische Funktionen:

- id : OCC $\times$ PROG $\rightarrow$?ID zur Bestimmung des Bezeichners eines Zugriffspfads
- pri, $post$: PROG $\times$ OCC $\rightarrow$?TYPE ist der Typ des Ausdrucks bzw. der vom Kontext erwartete Typ
- $coerce$: VALUE $\times$ TYPE $\times$ TYPE $\rightarrow$?VALUE konvertiert einen Wert des ersten Typs in einen äquivalenten Wert des zweiten Typs, wobei $coerce(x, t, t) \doteq x$

Konstanten

Der Wert einer Konstanten entspricht dem im Programm bezeichneten Wert



if CT **is** CONST **then**
  $Coerce(val(pc), const(pc, prog), pc)$
  $Proceed$

Untersorten: INTCONST $\sqsubseteq$ CONST, FLTCONST $\sqsubseteq$ CONST, BOOLCONST $\sqsubseteq$ CONST

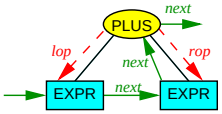
Statische Funktion: $const$: OCC $\times$ PROG $\rightarrow$?VALUE zur Bestimmung der Konstanten

Binärer Operationen

Binäre Ausdrucke (ohne Divisionsoperatoren)

- Die Werte binärer Ausdrücke verknüpfen die beiden Werte mit dem Operator des Ausdrucks
- Ganze Zahlen werden – falls erforderlich – automatisch in Gleitkommazahlen konvertiert werden.

if CT **is** PLUS **then**
  $Coerce(val(pc), Sum, pc)$
  $Proceed$

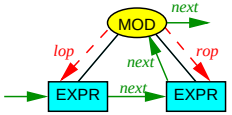


$$
\begin{array}{ll}
Lop & \triangleq val(lop(pc, prog)) \\
Rop & \triangleq val(rop(pc, prog)) \\
Sum & \triangleq intplus(Lop, Rop)
\end{array}
$$

Multiplikation und Subtraktion werden analog behandelt.

Division

- Die Werte binärer Ausdrücke verknüpfen die beiden Werte mit dem Operator des Ausdrucks
- Hat der rechte Operand von / oder % den Wert 0, dann bricht das Programm mit der Ausnahme `DivByZeroException` ab.



if CT **is** MOD **then**
  if $Rop \doteq 0$ **then**
    $exc := divByZero$
    $pc := handle(divByZero, pc)$
  else
    $Coerce(val(pc), Rem, pc)$
    $Proceed$

$Rem \triangleq mod(Lop, Rop)$

- $handle$: EXCEPTION $\rightarrow$ OCC spezifiziert die Anweisung, mit der bei Ausnahme fortgefahren werden muss
- Division wird analog behandelt.

Unäre Ausdrücke

Der Wert eines unärer Ausdrucks wendet den Operator auf den Wert des Operanden an.

if CT **is** NOT **then**
  $val(pc) := not(Opd)$
  $Proceed$
$Opd \triangleq val(opd(pc, prog))$

Unäres Minus und Plus wird analog behandelt unter Berücksichtigung von Anpassungen

Anweisungsfolgen

Leere Anweisungsfolge

Eine leere Anweisungsfolge hat keinen Effekt.

Stats::=NoStats

if *CT* **is** NOSTATS **then** *Proceed*

Nichtleere Anweisungsfolge

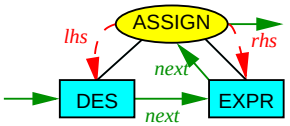
Die Ausführung eines Blocks führt alle seine Anweisungen nacheinander aus.

Keine Zustandsübergangsregel erforderlich, da das Fortschalten des Befehlszeigers bei den einzelnen Anweisungen erfolgt

Anweisungen: Zuweisung

- Der Zugriffspfad wird zu einem Verweis auf ein Objekt ausgewertet und der Ausdruck wird ausgewertet. Der Wert des Ausdrucks wird in das durch den Zugriffspfad verwiesene Objekt geschrieben.
- Danach wird mit der nächsten Anweisung fortgefahren.

STAT ::= ...|ASSIGN|...



if *CT* **is** ASSIGN $\wedge$ *isAtomic*(*Pri*(*Lhs*)) **then**
  mem(*loc*(*Lhs*)) := *val*(*Rhs*)
  Proceed

$Lhs \triangleq lhs(prog, pc)$
$Pri(p) \triangleq pri(prog, p)$
$Rhs \triangleq rhs(prog, pc)$

- Das geht nur für atomare Werte

$\Rightarrow$ Statische Funktion *isAtomic* : TYPE $\rightarrow$ BOOL

- $isAtomic(x) \doteq true$ für die Typen $x =$ INT, BOOL und FLOAT

Anweisungen: Lesen und Schreiben

Leseanweisung

- Die Lese-Anweisung bestimmt das Objekt, auf das der Zugriffspfad verweist und schreibt dort den aktuellen Wert des Eingabestroms. Die aktuelle Position des Eingabestroms auf den nächsten Wert fortgeschalten.
- Danach wird mit der nächsten Anweisung fortgefahren.

STAT ::= ...|READ|...

if *CT* **is** READ **then**
  mem(*loc*(*dest*(*prog*, *pc*))) := *head*(*inp*)
  inp := *tail*(*inp*)
  Proceed

Der Zugriffspfad darf kein Verbund– oder Klassentyp haben.

Schreibeanweisung

- Die Schreibe-Anwweisung wertet den Ausdruck aus und schreibt dessen Wert an das Ende des Ausgabestroms.
- Danach wird mit der nächsten Anweisung fortgefahren.

STAT ::= ...|WRITE|...

if *CT* **is** WRITE **then**
  out := *out*.*val*(*Arg*)
  Proceed

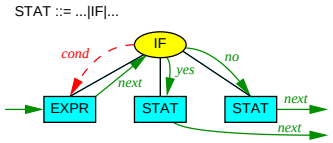
$Arg \triangleq arg(pc, prog)$

Anweisungen: Verzweigungen und Schleifen

Verzweigung

Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die erste Anweisung ausgeführt. Danach wird die Anweisung nach der Verzweigung ausgeführt. Ist das Ergebnis `false` und keine `else`-Anweisung vorhanden, dann wird die Anweisung nach der Verzweigung ausgeführt. Ansonsten wird zuerst die `else`-Anweisung und dann die Anweisung nach der Verzweigung ausgeführt.

STAT ::= ...|IF|...



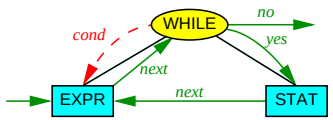
if *CT* **is** IF **then**
  if *val*(*Cond*) $\doteq$ *true* **then**
    pc := *yes*(*pc*, *prog*)
  else *pc* := *no*(*pc*, *prog*)

$Cond \triangleq cond(prog, pc)$

Schleifen

Der Ausdruck wird ausgewertet. Ist das Ergebnis *true* so wird die Anweisung ausgeführt. Danach wird wieder die Schleife ausgeführt. Ist das Ergebnis `false`, dann wird die Anweisung nach der Schleife ausgeführt.

STAT ::= ...|WHILE|...



if *CT* **is** WHILE **then**
  if *val*(*Cond*) $\doteq$ *true* **then**
    pc := *yes*(*pc*, *prog*)
  else *pc* := *no*(*pc*, *prog*)

Anweisungen: Schleifenabbruch– und Fortsetzung

Grundprinzip

- Jede Schleife kennt über *no* die Anweisung nach der Schleife und den ersten Befehl der Schleifenbedingung
- Diese werden über *break* : PROG × OCC →? bzw. *continue* : PROG × OCC →? auf die Anweisungen des Schleifenrumpfs fortgeschrieben

⇒ Teil der statischen Semantik

Break-Anweisung

Bei Ausführen einer `break`-Anweisung wird die Schleife abgebrochen und danach die Anweisung nach der Schleife ausgeführt

STAT ::= ...|BREAK|...

if *CT* **is** BREAK **then**
pc := *break*(*prog*, *pc*)

Continue-Anweisung

Bei Ausführen einer `continue`-Anweisung wird die Schleife abgebrochen und danach und danach die Schleifenanweisung ausgeführt.

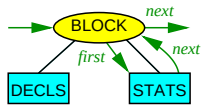
STAT ::= ...|CONTINUE|...

if *CT* **is** CONTINUE **then**
pc := *continue*(*prog*, *pc*)

Blockanweisung

Die Ausführung eines Blocks führt alle seine Anweisungen nacheinander aus. Danach wird die Anweisung nach dem Block ausgeführt.

STAT ::= ...|BLOCK|...



Beobachtungen

- Den lokalen Variablen des Blocks müssen bei Betreten **unterschiedliche** Objekte zugeordnet werden und diese Zuordnung bei Verlassen wieder aufgehoben werden.

⇒ Der Blockknoten wird zwei Mal betreten

⇒ Zwei Zustandsübergangsregeln

⇒ Dynamische Funktion *entered* : OCC →?BOOL

Zustandsübergangsregeln

if *CT* **is** BLOCK ∧ *entered*(*pc*) ≐ *false* **then**
AllocateVars(*locVars*(*prog*, *pc*))
entered(*pc*) := *true*
pc := *first*(*prog*, *pc*)

if *CT* **is** BLOCK ∧ *entered*(*pc*) ≐ *true* **then**
DeAllocateVars(*locVars*(*prog*, *pc*))
entered(*pc*) := *false*
Proceed

wobei *locVars* : PROG × OCC →?IDLIST die Liste der lokalen Variablen eines Blocks ergibt.

Allokation und Deallokation von Objekten

Allokation

Idee: Bestimmung einer Umgebung der lokalen Variablen mit Bindungen an paarweise unterschiedliche Objekte, die alle verschieden von den bisher gebundenen Objekten sind.

Eigenschaften: *Good*(*e*, *ids*) ≙
∀*x* : ID • *isBound*(*e*, *x*) ≐ *isElem*(*x*, *ids*)∧
∀*x*, *y* : ID • *D*(*bind*(*e*, *x*)) ∧ *D*(*bind*(*e*, *y*)) ∧ *bind*(*e*, *x*) ≐ *bind*(*e*, *y*) ⇒ *x* ≐ *y*∧
∀*x* : ID • *isBound*(*e*, *x*) ≐ *true* ⇒ *IsUsed*(*bind*(*e*, *x*)) ≐ *false*
wobei das Makro *IsUsed*(*l*) angibt, ob ein Verweis *l* schon benutzt wurde (wird später definiert).

Makro: *AllocateVars*(*ids*) ≙ **choose** *e* : ENV • *Good*(*e*, *ids*) **do**
env := *e*++*env*

Deallokation

Bindungen werden wieder bei Verlassen des Blocks zerstört:
DeAllocateVars(*ids*) ≙ *env* := *env* \ *ids*

Zusammenfassung

- Der Zustandsraum ist nun:
  spec STATE **extends** ENV
  sorts EXCEPTION
  operations

env :		→ ENV
mem :	LOC	→?VALUE
inp :		→ LISTOFVAL
out :		→ LISTOFVAL
prog :		→ PROG
pc :		→ OCC
exc :		→ EXCEPTION
val :	OCC	→?VALUE
loc :	OCC	→?LOC
entered :	OCC	→?BOOL

- Makro *IsUsed*, das angibt, ob ein Verweis *l* schon benutzt wurde, muss noch definiert werden.

☞ Es muss aber auf jeden Fall gelten:
isUsed(*l*) ≐ *true* ⇒ *IsUsed*(*env*, *l*) ≐ *true*

2.4.4 Verbunde und Klassen (C1)

Verbundobjekte

- Werte sind Tupel. Der Zugriff auf die Komponenten erfolgt über die Namen der Felder

⇒ Unter jedem der Namen ist ein Wert gespeichert

⇒ Für jeden der Namen ist ein Verweis auf ein Objekt zugeordnet

⇒ Zugriff auf Verbundfelder ergeben den dem Namen zugeordneten Verweis

⇒ Verbunde sind ebenfalls Umgebungen

- Aber bei Zuweisung müssen explizit die einzelnen Verbundfelder kopiert werden

Klassen

- Klassen sind Zeiger auf Verbunde

⇒ Es werden Verweise auf Verbunde gespeichert

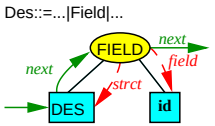
- Bei Zugriff auf eine Komponente muss erst diesem Verweis gefolgt werden.

Zustandsraum

spec $\textsc{Env}_1$ **extends** $\textsc{Env}$
subsorts $\mathrm{LOC} \sqsubseteq \mathrm{VALUE}$
operations $\mathit{floc} : \mathrm{LOC} \times \mathrm{ID} \to \mathrm{LOC}$
$\mathit{null} : \to \mathrm{LOC}$
spec $\textsc{State}_1$ **extends** $\textsc{State}, \textsc{Env}_1$
spec $\textsc{InitState}_1$ **extends** $\textsc{State}_1, \textsc{InitState}$

Verbundfeld– und Klassenfeldzugriffe

- Falls *des* Verbundtyp ist, bezeichnet der Zugriffspfad das durch **id** definierte Teilobjekt
- Falls *des* Klassentyp ist, muss der Wert von $\mathit{des} \neq$ `NULL` sein, ansonsten bricht das Programm mit einer `NullPointerDereferenceException` ab.
- Wenn der Wert von $\mathit{des} \neq$ `NULL` ist, wird zunächst das durch den Zeiger verwiesene Objekt ermittelt (**Dereferenzieren**). Der Klassenfeldzugriff verweist dann auf das durch **id** definierte Teilobjekt.



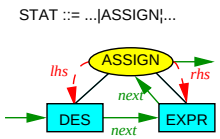
if CT **is** FIELD **then**
  if $\mathit{isClass}(\mathit{Pri}(\mathit{pc}))$ **then**
    if $\mathit{loc}(\mathit{pc}) = \mathit{null}$ **then**
      $\mathit{exc} := \mathit{nullPointerDeref}$
      $\mathit{pc} := \mathit{handle}(\mathit{nullPointerDeref}, \mathit{pc})$
    else $\mathit{loc}(\mathit{pc}) := \mathit{FldLoc}(\mathit{ClassLoc})$
      $\mathit{Coerce}(\mathit{val}(\mathit{pc}), \mathit{mem}(\mathit{FldLoc}(\mathit{ClassLoc})), \mathit{pc})$
      $\mathit{Proceed}$
  else $\mathit{loc}(\mathit{pc}) := \mathit{FldLoc}(\mathit{StrctLoc})$
    $\mathit{Coerce}(\mathit{val}(\mathit{pc}), \mathit{mem}(\mathit{FldLoc}(\mathit{StrctLoc})), \mathit{pc})$
    $\mathit{Proceed}$

$\mathit{StrctLoc} \triangleq \mathit{loc}(\mathit{strct}(\mathit{prog}, \mathit{pc}))$
$\mathit{ClassLoc} \triangleq \mathit{mem}(\mathit{StrctLoc})$
$\mathit{FldLoc}(l) \triangleq \mathit{floc}(l, \mathit{field}(\mathit{prog}, \mathit{pc}))$

Zuweisung

Wird an ein Verbundobjekt zugewiesen, dann werden die Werte der Verbundfelder in das Teilobjekt, auf das entsprechende Verbundfeld verweist, geschrieben. Wird an ein Verbundzeigerobjekt zugewiesen wird der Zeiger in das Objekt, auf das die linke Seite verweist, geschrieben.

⇒ Nur für Verbundtypen A gilt $\mathit{isAtomic}(\mathrm{A}) \doteq \mathit{false}$



if CT **is** ASSIGN $\wedge$ $\mathit{isAtomic}(\mathit{Pri}(\mathit{lhs})) \doteq \mathit{false}$ **then**
  $\mathit{StructCopy}(\mathit{lhs}, \mathit{rhs})$
  $\mathit{Proceed}$

$\mathit{StructCopy}(\mathit{dest}, \mathit{src}) \triangleq$ **forall** $x : \mathrm{ID} \bullet \mathit{isField}(\mathit{Pri}(\mathit{dest}), x)$ **do**
  if $\mathit{isAtomic}(\mathit{gettype}(\mathit{fields}(\mathit{Pri}(\mathit{dest}), x))$ **then**
    $\mathit{mem}(\mathit{FldLoc}(\mathit{loc}(\mathit{dest}))) := \mathit{mem}(\mathit{FldLoc}(\mathit{loc}(\mathit{src})))$
  else $\mathit{StructCopy}(\mathit{FldLoc}(\mathit{loc}(\mathit{dest})), \mathit{FldLoc}(\mathit{loc}(\mathit{src})))$

Bemerkung

StructCopy ist definiert, da jedes Verbundobjekt endlich ist.

forall und choose

Definition 2.16 (forall-Aktualisierungen)

Sei Σ eine Signatur und $\mathcal{A} \triangleq \langle \Sigma, \mathbb{A}, \mathbb{I}, R \rangle$ eine ASM, wobei in einer Regel **if** φ **then** U auch Aktualisierungen der Form $\mathit{FORALL} \triangleq$ **forall** $x : \mathrm{A} \bullet \psi$ **do** U' und $\mathit{CHOOSE} \triangleq$ **choose** $x : \mathrm{A} \bullet \chi$ **do** U'' vorkommen dürfen.

i. Durch **forall** wird eine Aktualisierungsmenge $\mathit{Update}_{\mathfrak{A}}(\mathit{FORALL}) \triangleq \{u[t/x] : t \in \mathcal{T}_{\mathrm{A}}(\Sigma) \wedge u \in U' \wedge \mathfrak{A} \models \psi[t/x]\}$. Die **forall**-Aktualisierungsmenge ist **beschränkt** gdw. $|\{\langle [\![t_1]\!]_{\mathfrak{A}}, [\![t_2]\!]_{\mathfrak{A}} \rangle : t_1 := t_2 \in \mathit{Update}_{\mathfrak{A}}(\mathit{FORALL})\}| < \infty$ ist.

ii. Jede Aktualisierungsmenge $\{u[x/t] : u \in U''\}$ für ein $t \in \mathcal{T}_{\mathrm{A}}(\Sigma)$ mit $\mathfrak{A} \models \chi[t/x]$ ist eine *CHOOSE*-**Aktualisierungsmenge**

iii. Sei $U = U_1 \uplus U_2 \uplus U_3$ wobei U_1 eine Menge von Aktualisierungen der Form $t_1 := t_2$, U_2 eine Menge von **forall**-Aktualisierungen und U_3 eine Menge von **choose**-Aktualisierungen ist. Weiter sei $\mathfrak{A} \in \mathbb{A}$ ein Zustand. Eine Menge $\bar{U} \triangleq U_1 \cup \bigcup_{f \in U_2} \mathit{Update}_{\mathfrak{A}}(f) \cup \bigcup_{c \in U_3} \{u : u \text{ ist } c\text{-Aktualisierung}\}$ heißt durch $\mathfrak{A}$ und U definierte Aktualisierungsmenge.

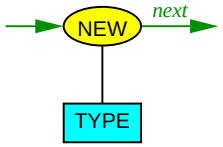
iv. Sei $\mathbb{U} \triangleq \{\bar{U} : \exists$ **if** φ **then** $U \in R \bullet \mathfrak{A} \models \varphi \wedge \bar{U}$ ist durch $\mathfrak{A}$ und U definierte Aktualisierungsmenge$\}$ eine Menge von Aktualisierungsmengen. Eine Menge $\mathit{Update}_{\mathcal{A}}(\mathfrak{A}) = \bigcup_{\bar{U} \in \mathbb{U}} \bar{U}$ heißt **Aktualisierungsmenge** im Zustand $\mathfrak{A}$. Ein Zustand $\mathfrak{A}'$ heißt **Nachfolgezustand** von $\mathfrak{A}$ gdw. $\mathfrak{A}' = \mathit{next}_U(\mathfrak{A})$ für eine Aktualisierungsmenge U im Zustand $\mathfrak{A}$.

Erzeugen von Objekten

Eine `new`-Operation erzeugt ein Objekt der Klasse und liefert einen Verweis auf dieses Objekt.

⇒ Dieser Verweis darf nicht benutzt worden sein

EXPR ::=...|NEW|...



if *CT* **is** NEW **then**
  choose *l* : LOC • *IsUsed*(*l*) ≐ *false* **do**
    loc(*pc*) := *l*
  Proceed

Benutzte Verweise

- Verweise auf Verbundfelder werden immer benutzt

⇒ *isUsed*(*e*, *floc*(*l*, *x*)) ≐ *false*

2.4.5 Prozeduren und Funktionen

Beobachtung

- Prozeduraufruf sichert Zustand (Bindungen, Ausdruckswerte) und legt neue Objekte an
- Dabei werden Parameter übergeben
- Nach Rückkehr aus der Prozedur wird der alte Zustand wieder hergestellt
- Dabei werden Funktionsresultate übergeben

⇒ Klassisches Verhalten eines Kellers (engl. *stack*)

Laufzeitkeller

spec ENV_2 **extends** ENV_1
  sorts FRAME, FRAMESTACK, EXPRVALS
  operations

mkframe :	ENV × OCC × EXPRVALS	→ FRAME
mkstack :		→ FRAMESTACK
push :	FRAMESTACK × FRAME	→ FRAMESTACK
pop :	FRAMESTACK	→?FRAMESTACK
getenv :	FRAMESTACK	→?ENV
getpc :	FRAMESTACK	→?OCC
getval :	FRAMESTACK	→?EXPRVALS
novals :		→ EXPRVALS
addval :	OCC × VALUE × EXPRVALS	→ EXPRVALS
findval :	OCC × EXPRVALS	→?VALUE

- *mkframe* verwaltet den Zustand
- Mit *getenv*, *getpc* und *getval* ist die Wiederherstellung des Zustands möglich

Erweiterung des Zustandsraums und des Anfangszustands

Erweiterung des Zustandsraums

Für die Verwaltung der Prozeduraufrufe muss noch zusätzlich gegenbüber C0 ein Prozedurkeller eingeführt werden.

spec STATE_2 **extends** $\text{STATE}_1, \text{ENV}_2$
operations *procstack* :⟶ FRAMESTACK

Anfangszustand

Zusätzlich zu den Bedingungen in C0 muss noch der Prozedurkeller leer sein:

spec INITSTATE_1 **extends** $\text{STATE}_1, \text{INITSTATE}$
axioms *procstack* ≐ *mkstack*

Anlegen der Parameter und Parameterübergabe

Vorgehen

- Anlegen der Objekte für die Parameter

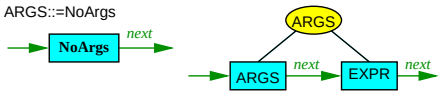
⇒ Parameter können übergeben werden

- Vorbereiten für andere Übergabemodi

⇒ Statische Funktion *mode* : OCC →?MODE

Auswerten der Argumente

ARGS::=NoArgs



if *CT* **is** NOPARS **then** *Proceed*

Parameterübergabe

PASS(*ids*) ≙ **choose** *e* : ENV • *Good*(*e*, *ids*) **do**
  env := *e*
  forall *i* : INT • 0 ≤ *i* < *length*(*ids*) **do**
    if *isAtomic*(*Post*(*Arg*(*i*))) ≐ *true* **then**
      mem(*bind*(*e*, *get*(*ids*, *i*))) := *val*(*Arg*(*i*))
    else *StructCopy*(*bind*(*e*, *get*(*ids*, *i*)), *Arg*(*i*))

Arg(*i*) ≙ *arg*(*prog*, *pc*, *i*)

Zustandssicherung

Was ist zu tun?

- Sichern des Befehlszeigers
- Sichern der Bindungen lokaler Variable an Objekte
- Bei Funktionen müssen noch Zwischenergebnisse des Ausdrucks der gerade berechnet wird, gesichert werden.
  - Statische Funktion *exproccs* : PROG × OCC → OCCLIST; *exproccs*(p, o) ist die eine Liste der Navilagitionslisten, die zum selben Ausdruck wie o gehören. Die Liste ist leer, falls $\neg occ(p, o)$ **is** EXPR.
  - Sorte SAVELIST mit Funktionen

mksavelist :		→ SAVELIST
save :	OCC × VALUE × SAVELIST	→ SAVELIST
findval :	OCC × SAVELIST	→ VALUE

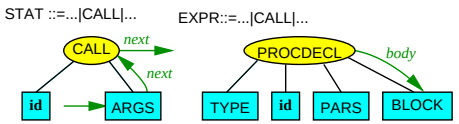
  - Abgeleitete dynamische Funktion *save* : OCCLIST →?SAVELIST:

    $save(nil) \doteq mksavelist$

    $save(cons(o, occs) \doteq save(o, val(o), save(occs)$

Prozeduraufruf

- Mit dem Prozedur–/Funktionsaufruf werden den Parametern Objekte zugeordnet.
- Bei einem Prozedur– oder Funktionsaufruf werden die Argumente ausgewertet und an die Parameter positional übergehen, d.h. der Wert eines Arguments wird in das Objekt geschrieben, auf das der Parameter verweist. Danach wird der Prozedurrumpf ausgeführt.



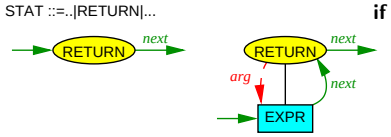
if *CT* **is** CALL **then**
  Pass(*pars*(*CurProc*))
  procstack := *push*(*procsstack*, *Frame*)
  pc := *body*(*CurProc*)

Frame ≙ *mkframe*(*env*, *pc*, *save*(*exproccs*(*pc*)))
CurProc ≙ *identifyDef*(*deftab*(*prog*, *pc*), *occ*(*prog*, *pc*.0))

- *deftab* ist eine Definitionstabelle, *identifyDef* identifiziert eine Deklaration
- Eine Prozedurdefinition speichert u.A. die Parameter(namen) und den Rumpf

Prozedurrückkehr

- Ein Prozedur– oder Funktionsaufruf **kehrt zurück**, wenn die letzte Anweisung des Prozedurrumpfs oder eine `return`-Anweisung ausgeführt wurde.
- Nach Rückkehr von einem Prozeduraufruf wird die Anweisung nach dem Aufruf ausgeführt.
- Nach Rückkehr von einem Funktionsaufruf wird mit der Ausführung der Anweisung, in der der Aufruf vorkam, fortgefahren. Der Wert eines Funktionsaufrufs ist der Wert des Ausdrucks der ausgeführten `return`-Anweisung.



if *CT* **is** RETURN ∨ ¬*D*(*CT*) **then**
  pc := *next*(*prog*, *OldPc*)
  env := *OldEnv*
  procstack := *pop*(*procstack*)
  forall *o* : OCC • *isElem*(*o*, *exproccs*(*prog*, *OldPc*)) **do**
    val(*o*) := *findval*(*o*, *OldVals*)
  if *isFunction*(*CurProc*) **then**
    if *isAtomic*(*rettype*(*CurProc*)) ≐ *true* **then**
      Coerce(*val*(*OldPc*), *val*(*Arg*), *OldPc*)
    else choose *l* : LOC • *IsUsed*(*l*) ≐ *false* **do**
      StructCopy(*l*, *Arg*)
      loc(*OldPc*) := *l*

OldPc ≙ *getpc*(*procstack*)
OldEnv ≙ *getenv*(*procstack*)
OldVals ≙ *getval*(*procstack*)
Arg ≙ *arg*(*prog*, *pc*)

Statische Funktion *rettype* : PROCDEF → TYPE ergibt Rückgabetyp der Funktion

Diskussion

- Der Zustandsraum ist nun:

  spec STATE **extends** ENV_1
  sorts EXCEPTION
  operations

env :		→ ENV
mem :	LOC	→?VALUE
inp :		→ LISTOFVAL
out :		→ LISTOFVAL
prog :		→ PROG
pc :		→ OCC
exc :		→ EXCEPTION
val :	OCC	→?VALUE
loc :	OCC	→?LOC
entered :	OCC	→?BOOL
procstack :		→?FRAMESTACK

- Makro *IsUsed*, das angibt, ob ein Verweis *l* schon benutzt wurde, muss noch definiert werden.
- Auch Verweise im Prozedurkeller werden benutzt: Funktion *isUsed* : FRAMESTACK × LOC → BOOL

⇒ $IsUsed(l) \triangleq isUsed(env, l) \doteq true \vee isUsed(procstack, l) \doteq true \vee isUsed(mem(x), l) \doteq true$

Übungen

- Erweitern Sie die Semantik um Ausnahmebehandlung:
  exception ::=`try` *block*
  (`when` *excp stat*)$^+$
  [`otherwise` *stat*]
  Wird in *block* eine Ausnahme ausgelöst, die in einem der `when`-Fälle aufgeführt ist, dann wird die entsprechend Anweisung ausgeführt. Danach wird die Anweisung nach dem Block ausgeführt. Mit `otherwise` werden alle nicht-aufgeführt Ausnahmen behandelt. Ein Prozedur– oder Funktionsaufruf endet mit einer Ausnahme, wenn sie im Rumpf nicht behandelt wird.
- Einführung von Benutzerdefinierten Ausnahmen:
  exceptdecl ::= `exception` **id**
  raise ::= `raise` **id**
  - Die Ausnahmen müssen global deklariert werden. Die Bezeichner in allen Deklarationen müssen paarweise verschieden sein.
  - Mit der `raise`-Anweisung wird explizit eine Ausnahme ausgelöst. Die ausgelöste Ausnahme muss deklariert sein.
- Einführung von Referenzaufruf (Modus `ref`) und Wert-/Ergebnisaufruf (Modus `inout`)

Übung: Einführen von Reihungen

arrtype ::= *type* `[ ]`
arracc ::= *des* `[` *expr* `[]`
arrcreate ::= `new` *type* `[` *expr* `[]`

- Reihungen ordnen Indices Werten zu
- Reihungszugriffe sind Zugriffspfade
- Der Typ eines Reihungszugriffs ist der Elementtyp der Reihung
- Der Typ des Indexausdrucks muss `int` sein.
- Der Typ einer Reihungserzeugung ist der Elementtyp.
- Der Typ des Ausdrucks einer Reihungserzeugung muss `int` sein.
- Ein Reihungszugriff werden den Indexausdruck zu einer Zahl n aus. Ist $n < 0$ oder n größer gleich des Reihungsumfangs, so wird die Ausnahme `IndexOutOfBoundException` ausgelöst. Ansonsten bezeichnet der Reihungszugriff das $n + 1$-te Element der Reihung bzw. dessen Wert
- Eine Reihungserzeugung wertet den Ausdruck zu einer Zahl n aus. Ist $n \leq 0$ so wird die Ausnahme `IndexOutOfBoundException` ausgelöst. Ansonsten wird eine Reihung mit n Elementen erzeugt. Das Ergebnis ist ein Verweis auf die erzeugte Reihung