Semantik

Nachweisbar korrekte Implementierung

Autor: Walter Kammergruber

erstellt im Rahmen des Informatikproseminars:

"Grundlagen höherer Programmiersprachen"
Wintersemester 2002/03
(Kröger, Rauschmayer)

Bezogen auf Kapitel 3 aus:

Semantics with applications, a formal introduction
von Hanne Riis Nielson und Flemming Nielson
http://www.daimi.au.dk/~hrn/

Inhaltsverzeichnis

Nachweisbar korrekte Implementierung

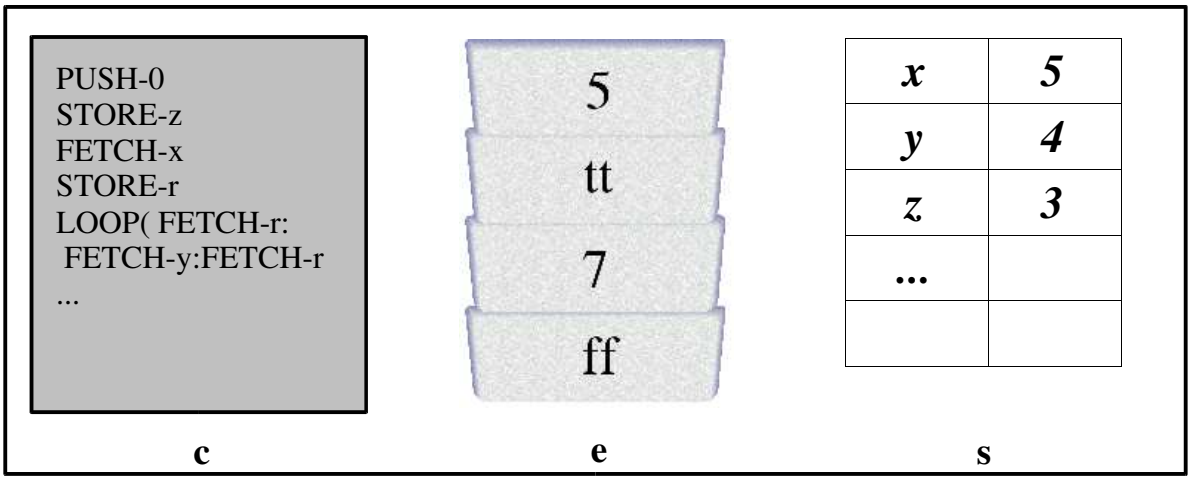
1. Einleitung

Eine formale Spezifizierung der Semantik einer Programmiersprache ist zumeist notwendig, um deren Implementierung auf ihre Richtigkeit überprüfen zu können. Hierbei wird dies am Beispiel der Sprache While gezeigt. Zunächst wird der Programmcode in einen Assemblercode für eine abstrakte Maschine übersetzt und später die Korrektheit der Übersetzung bewiesen. Der zugrundeliegende Gedanke dabei ist, dass die Bedeutung der Befehle aus der abstrakten Maschine mittels einer operationalen Semantik definiert wird. Danach wird eine Übersetzungsfunktion definiert, die die Ausdrücke und Befehle aus While in sequenzielle Instruktionen der abstrakten Maschine abbildet. Wenn nun ein in While geschriebenes Programm in Code der abstrakten Maschine umgewandelt wird und auf dieser abstrakten Maschine ausgeführt wird, so muss das Resultat dasselbe sein, wie wenn es in der Semantikfunktion S_{ns} oder S_{sos} spezifiziert worden wäre.

2 Die abstrakte Maschine (AM)

2.1. Semantik

Bei der Spezifizierung der abstrakten Maschine (AM) werden zunächst ihre Konfigurationen dargestellt, danach erst ihre Instruktionen und deren Bedeutung.
AM hat die Konfigurationen der Form <c, e, s>:



	c	e	s
aus Menge	Code	Stack= (Z ∪ T)*	State= Var → Z
ist konkret	Programmcode	Abarbeitungsstapel	Speicher

Der Abarbeitungsstapel wird zu Ausführung von arithmetischen und bool´schen Ausdrücken verwendet. Formal gesehen ist er eine Liste bestehend aus Einzelwerten.
Also ist e ∈ Stack, wobei Stack= (Z ∪ T)*. Vereinfacht wird der Speicher mit dem Zustand gleichgesetzt, somit ist s ∈ State, wobei State= Var → Z und zum Sichern der Variablen benutzt wird.

Die Befehle von AM werden nun über die abstrakte Syntax festgelegt:

inst ::= PUSH-n | ADD | MULT | SUB
| TRUE | FALSE | EQ | LE | AND | NEG
| FETCH-x | STORE-x
| NOOP | BRANCH(c, c) | LOOP(c, c)

c ::= □ | *inst*: c

hierbei ist □ die leere Sequenz. Es wird c als Meta-Variable über den Code betrachtet, wobei dieser eine syntaktische Kategorie von Befehlssequenzen ist.
Nun ergibt sich folgende Beziehung:

<c, e, s> □ Code × Stack × State

Eine Endkonfiguration hat die leere Sequenz als Code-Komponente.

Sie ist der Gestalt: <□ , e, s>.

Die Transitionsrelation der AM wird durch <c, e, s> ✇<c´, e´, s´> ausgedrückt.
✇gibt dabei an, wie der Befehl ausgeführt wird. Der Übergang von dem einen Zustand in den anderen erfolgt in einem Schritt. Die Ausführung des Programms wird definiert.

Folgende Relationsdefinition wird verwendet:

$$
\begin{array}{lcl}
\langle \text{PUSH-}n{:}c,\ e,\ s\rangle & \triangleright & \langle c,\ \mathcal{N}[\![n]\!]{:}e,\ s\rangle \\
\langle \text{ADD:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c,\ (z_1{+}z_2){:}e,\ s\rangle \quad \text{if } z_1,\ z_2{\in}\mathbf{Z} \\
\langle \text{MULT:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c,\ (z_1{\star}z_2){:}e,\ s\rangle \quad \text{if } z_1,\ z_2{\in}\mathbf{Z} \\
\langle \text{SUB:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c,\ (z_1{-}z_2){:}e,\ s\rangle \quad \text{if } z_1,\ z_2{\in}\mathbf{Z} \\
\langle \text{TRUE:}c,\ e,\ s\rangle & \triangleright & \langle c,\ \mathbf{tt}{:}e,\ s\rangle \\
\langle \text{FALSE:}c,\ e,\ s\rangle & \triangleright & \langle c,\ \mathbf{ff}{:}e,\ s\rangle \\
\langle \text{EQ:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c,\ (z_1{=}z_2){:}e,\ s\rangle \quad \text{if } z_1,\ z_2{\in}\mathbf{Z} \\
\langle \text{LE:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c,\ (z_1{\leq}z_2){:}e,\ s\rangle \quad \text{if } z_1,\ z_2{\in}\mathbf{Z} \\
\langle \text{AND:}c,\ t_1{:}t_2{:}e,\ s\rangle & \triangleright & \\
\multicolumn{3}{c}{\left\{\begin{array}{ll} \langle c, \mathbf{tt} : e, s\rangle & \text{if } t_1{=}\mathbf{tt} \text{ and } t_2{=}\mathbf{tt} \\ \langle c, \mathbf{ff} : e, s\rangle & \text{if } t_1{=}\mathbf{ff} \text{ or } t_2{=}\mathbf{ff},\ t_1,\ t_2{\in}\mathbf{T} \end{array}\right.} \\
\langle \text{NEG:}c,\ t{:}e,\ s\rangle & \triangleright & \left\{\begin{array}{ll} \langle c, \mathbf{ff} : e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c, \mathbf{tt} : e, s\rangle & \text{if } t{=}\mathbf{ff} \end{array}\right. \\
\langle \text{FETCH-}x{:}c,\ e,\ s\rangle & \triangleright & \langle c,\ (s\ x){:}e,\ s\rangle \\
\langle \text{STORE-}x{:}c,\ z{:}e,\ s\rangle & \triangleright & \langle c,\ e,\ s[x{\mapsto}z]\rangle \quad \text{if } z{\in}\mathbf{Z} \\
\langle \text{NOOP:}c,\ e,\ s\rangle & \triangleright & \langle c,\ e,\ s\rangle \\
\langle \text{BRANCH}(c_1,\ c_2){:}c,\ t{:}e,\ s\rangle & \triangleright & \left\{\begin{array}{ll} \langle c_1 : c, e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{if } t{=}\mathbf{ff} \end{array}\right. \\
\langle \text{LOOP}(c_1,\ c_2){:}c,\ e,\ s\rangle & \triangleright & \\
\multicolumn{3}{c}{\langle c_1{:}\text{BRANCH}(c_2{:}\text{LOOP}(c_1,\ c_2),\ \text{NOOP}){:}c,\ e,\ s\rangle}
\end{array}
$$

Tabelle 1: Operationale Semantik der AM

Zusätzlich zu den normalen arithmetischen und boolschen Operationen werden sechs auf den Ausarbeitungsstapel bezogene Befehle definiert. PUSH-n legt legt einen konstanten Wert n auf den Stapel, TRUE und FALSE geben tt, bzw. ff auf den Stapel. FETCH-x schreibt den an x gebundenen Wert auf den Stapel. Der Befehl BRANCH(c_1, c_2) lenkt obendrein den Kontrollfluß. Wenn auf dem Stapel tt liegt, wird c_1 ausgeführt anderenfalls c_2. BRANCH kann daher zur Implementierung einer Bedingung verwendet werden. LOOP wird deshalb durch BRANCH ausgedrückt (siehe Tabelle 1).
Nun wird eine Ausführungssequenz für AM definiert.Gegeben sei eine Befehlssequenz und ein Speicher s. Eine Ausführungssequenz für c und s ist dann

- eine endliche Sequenz
  $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$
  (für die gilt: $\gamma_0 = \langle c, \square, s \rangle$ und $\gamma_i \to \gamma_{i+1}$ mit $0 \square i < k$, $k \square 0$
  und es gibt kein γ, so dass $\gamma_k \to \gamma$)
- eine unendliche Sequenz
  $\gamma_0, \gamma_1, \gamma_2, \ldots$
  (für die gilt: $\gamma_0 = \langle c, \square, s \rangle$ und $\gamma_i \to \gamma_{i+1}$ mit $0 \square i$).

Der Initialisierungsstapel ist immer die leere Sequenz. Eine Ausführungssequenz terminiert nur dann, wenn sie finit ist, anderenfalls durchläuft sie sich endlos.
Eine terminierende Ausführungssequenz kann in eine Konfiguration mit leerem Code enden (<□, e, s>) oder in einer unbefriedigten Konfiguration (z.B. <SUB, □, s>).

Beispiele:
(1) Wir betrachten folgende Befehlssequenz:

PUSH-1:FETCH-x:ADD:STORE-x

Als Ausgangswert nehmen wir s x=3, wir bekommen also:

<PUSH-1:FETCH-x:ADD:STORE-x, □, s>
→<FETCH-x:ADD:STORE-x, 1, s>
→<ADD:STORE-x, 3:1, s>
→<STORE-x, 4, s>
→<□, □, s[x↦4]>

Die Abarbeitung endet hier mangels nächsten Ausführungsschrittes. Somit handelt es sich um ein terminierendes Beispiel.

(2) Gegeben sei dieser Code:

LOOP(TRUE, NOOP)

Abzuarbeiten:
<LOOP(TRUE, NOOP), □, s>
→<TRUE:BRANCH(NOOP:LOOP(TRUE, NOOP), NOOP), □, s>
→<BRANCH(NOOP:LOOP(TRUE, NOOP), NOOP), tt, s>
→<NOOP:LOOP(TRUE, NOOP), □, s>
→<LOOP(TRUE, NOOP), □, s>
→..
Die Schleife wird endlos durchlaufen und Ausführungsequenz terminiert nicht.

2.2. AM ist deterministisch

Definition von Determinismus bzgl. Eines Programms:

Zu jedem Zeitpunkt während der Ausführung eines deterministischen Programms existiert genau eine einzige mögliche Instruktion. Wählt man genau eine Instruktion I_K in einem deterministischen Programm, so kann eindeutig bestimmt werden, welche Anweisung als nächste nach I_K ausgeführt werden wird. Wird diese Regel verletzt, wird ein Programm nichtdeterministisch genannt. Daraus folgt, dass ein nichtdeterministisches Programm bei aufeinanderfolgenden Programmausführungen verschiedene Ergebnisse produzieren kann, auch wenn es die gleichen Eingaben erhält.

AM hat eine deterministische Programmausführung, da nach jeder Zustandsübersetzung die Eigenschaften der Vorzustände erhalten bleiben. Wollte man den Beweis dazu erbringen, würde dieser mittels vollständiger Induktion über die Länge der Berechnungssequenzen passieren. Der Beweis wird hier allerdings der Kürze wegen nicht erbracht.

3. Spezifizierung der Übersetzung

3.1. Arithmetische und boolsche Ausdrücke

Arithmetische und boolsche Ausdrücke werden auf dem Ausarbeitungsstapel der abstrakten Maschine ausgewertet. Damit der Code dieser Anforderung genügt werden folgende totale Funktionen definiert:

CA: Aexp → Code

und

CB: Bexp → Code

Spezifiziert werden sie durch Tabelle 2:

$\mathcal{CA}[\![n]\!]$	=	PUSH-n
$\mathcal{CA}[\![x]\!]$	=	FETCH-x
$\mathcal{CA}[\![a_1+a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:ADD
$\mathcal{CA}[\![a_1 \star a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:MULT
$\mathcal{CA}[\![a_1-a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:SUB
$\mathcal{CB}[\![\texttt{true}]\!]$	=	TRUE
$\mathcal{CB}[\![\texttt{false}]\!]$	=	FALSE
$\mathcal{CB}[\![a_1 = a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:EQ
$\mathcal{CB}[\![a_1 \leq a_2]\!]$	=	$\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:LE
$\mathcal{CB}[\![\neg b]\!]$	=	$\mathcal{CB}[\![b]\!]$:NEG
$\mathcal{CB}[\![b_1 \wedge b_2]\!]$	=	$\mathcal{CB}[\![b_2]\!]$:$\mathcal{CB}[\![b_1]\!]$:AND

Tabelle 2: Übersetzung von Ausdrücken

Bei binären Ausdrücken wird die Reihenfolge so geändert, dass zuerst das rechte gefolgt vom linken Argument und schließlich der Operator stehen. Auf diese Weise ist die Reihenfolge auf dem Evaluierungstapel richtig.

Beispiel :

Gegeben sei x+1, ein arithmetischer Ausdruck. Dieser wird folgendermaßen umgewandelt:

CA(x+1)= CA(1):CA(x):ADD=PUSH-1:FETCH-x:ADD

3.2 Statements

Die Übersetzung von Statements wird duch die totale Funktion

CS: Stm → Code

bestimmt, dargellegt in Tabelle 3:

$\mathcal{CS}[\![x := a]\!]$	=	$\mathcal{CA}[\![a]\!]$:STORE-x
$\mathcal{CS}[\![\texttt{skip}]\!]$	=	NOOP
$\mathcal{CS}[\![S_1;S_2]\!]$	=	$\mathcal{CS}[\![S_1]\!]$:$\mathcal{CS}[\![S_2]\!]$
$\mathcal{CS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]$	=	$\mathcal{CB}[\![b]\!]$:BRANCH($\mathcal{CS}[\![S_1]\!]$,$\mathcal{CS}[\![S_2]\!]$)
$\mathcal{CS}[\![\texttt{while } b \texttt{ do } S]\!]$	=	LOOP($\mathcal{CB}[\![b]\!]$,$\mathcal{CS}[\![S]\!]$)

Tabelle 3: Übersetzung von Statements

Der Code, generiert für arithmetische Operationen, stellt sicher, dass diese oberhalb von STORE-x auf dem Stapel liegen. Für das skip-Statement wird der NOOP-Befehl erzeugt. Bei der konditionalen Anweisung wird gewährleiste, dass der richtige boolsche Wert auf dem Stapel gelegt wird, um den BRANCH-Befehl richtig auszuführen. Der LOOP-Befehl ist ähnlich dem BRANCH-Befehl.

Beispiel:

CS(if 0=0 then 3 else 4) = CB[0=0]: BRANCH(CS(3),CS(4))=
CA[0]:CA[0]:EQ:BRANCH(CS(3),CS(4)) =CB (true):BRANCH(CS (3),CS(4))=
TRUE:BRANCH(CS (3);CS(4))= TRUE:BRANCH(CA [3]:STORE-x,CA [4]-STORE-x)=
CA [3]:STORE-x= 3

4. Korrektheit

Die Korrektheit der Implementierung ermöglicht es zu zeigen, dass wenn das stament zuerst in den Code für AM übersetzt wird, und dieser dann ausgeführt wird, man dasselbe Ergebnis erhält wie in der operationalen Semantik von While spezifiziert.

4.1. Ausdrücke

Die Richtigkeit der Implementierung von arithmetischen Ausdrücken wird durch das folgende Lemma ausgedrückt:

Lemma 1:

Für alle arithmetischen Ausdrücke a existiert

$< CA[a], \varepsilon, s> \overset{*}{\rightarrow} <\varepsilon, A[a]s, s>$.

Weiterhin haben alle dazwischenliegenden Konfigurationen dieser Ausführungssequenz keinen leeren Ausarbeitungsstapel.

Beweis:

Der Beweis erfolgt mittels struktureller Induktion über a. Dies soll hier nur an drei Beispielfällen aufgezeigt werden. Die übrigen Fälle sind analog zu vollziehen.

1. Fall: n

Gegeben sei CA[n]= PUSH-n. Mit Tabelle 1 ergibt sich:

$<$PUSH-n, ε, s$> \rightarrow <\varepsilon$, N[n], s$>$

Wegen A[n]s =N[n] laut Definition der Semantik von arithmetischen Ausdrücken (hier nicht aufgeführt) ist dieser Fall bewiesen.

2. Fall: x

Gegeben sei CA[x]= FETCH-n. Mit Tabelle 1 ergibt sich:

$<$FETCH-x, ε, s$> \rightarrow <\varepsilon$, (s x), s$>$

Wegen A[x]s = s x ergibt sich die Richtigkeit.

3. Fall: a_1+a_2:

Gegeben sei $CA[a_1+a_2]= CA[a_1]:CA[a_2]:$ADD

Die Induktionshypothesen auf a_1 und a_2 angewandt ergeben:

$$< CA[a_1], \varepsilon, s> \overset{*}{\rightarrow} <\varepsilon, A[a_1]s, s>$$

und

$$< CA[a_2], \varepsilon, s> \overset{*}{\rightarrow} <\varepsilon, A[a_2]s, s>$$

In beiden Fällen erhält man keine Zwischenkonfiguration mit einem leeren Evaluierungstapel.

Wir erhalten also:

$$< CA[a_2]:CA[a_1]:ADD, \varepsilon, s> \overset{*}{\rightarrow} <CA[a_1]:ADD, A[a_2]s, s>$$

weiterentwickelt:

$< CA[a_1]:ADD, A[a_2], s> \rightarrow^* < ADD, (A[a_1]s):(A[a_2]s), s>$

Benützt man die Übersetzungsrelation für ADD gegeben in Tabelle 1 erhält man:

$< ADD, (A[a_1]s):(A[a_2]s), s> \rightarrow <\square, A[a_1]s + A[a_2]s, s>$

Es ist einfach zu sehen, dass alle Zwischenkonfigurationen keinen leeren Evaluierungstapel haben. Da $A[a_1+a_2]s = A[a_1]s + A[a_2]s$ gilt ergibt sich das gewünschte Ergebnis.
Für die weiteren Fälle ist das Vorgehen ähnlich.

4.2. Statements

Wenn man die Richtigkeit des Ergebnisses von Statements beweisen will, so kann man entweder die natürliche oder die strukturell operationale Semantik verwenden. Hier würde die natürliche benützt. Allerdings muß der Beweis der Kürze wegen entfallen.

Doch der wichtige Satz wird dennoch nicht enthalten:

Satz:

Für jedes stament S in While gilt: $S_{ns}(S) = S_{am}(S)$.

Dieser Satz bindet das Verhalten eines Statements in natürlicher Semantik mit dem Verhalten des Codes für die Abstrakte Maschine mit ihrer strukturell operationalen Semantik.

Es gilt somit:

- Wenn bei der Ausführung von S in einem beliebigen Zustand dieses in einer Semantik terminierend ist, so ist S auch in der anderen Semantik terminierend und das reultierende Ergebnis ist dasselbe.
- Hingegen ist S in einem beliebigen Zustand in einer Semantik nicht terminierend, so ist es das auch nicht im anderen.