Universität Siegen
Fachbereich 12, Angewandte Informatik und Elektrotechnik
Veranstalter: PD. Dr. Kurt Sieber; Fachgruppe Programmiersprachen

Umgebungssemantik

Codegenerierung

Uhrhan, Christian

29. April 2009

Betreut durch Dipl. Inf. Benedikt Meurer

Inhaltsverzeichnis

1 Einführung

Ziel dieser Seminararbeit ist es, eine semantikerhaltende Übersetzung für eine funktionale Programmiersprache mit De Bruijn Indizes in eine maschinennahe Programmiersprache anzugeben. Diese Seminararbeit baut auf die beiden Arbeiten von [Reh09] und [Meu09] auf. Auf die dort verwendete Programmiersprache soll hier kurz eingegangen werden.

1.1 Syntax und Semantik der funktionalen Programmiersprache $\mathbf{L_{db}}$

Die verwendete Programmiersprache bezeichnen wir im Folgenden mit $\mathbf{L_{db}}$. Sie basiert im Wesentlichen auf der verwendeten Programmiersprache in [Meu09], wobei wir einige kleine Änderungen vornehmen werden, welche dazu führen, dass die Übersetzung vereinfacht wird.

Definition: 1 (Syntax von $\mathbf{L_{db}}$)**.** *Vorgegeben seien*

- *eine Menge* $Bool = \{true, false\}$ *von booleschen Konstanten* b*,*
- *eine Menge* $Int = \mathbb{Z}$ *von Integerkonstanten* z*,*
- *eine (unendliche) Menge* Id *von Namen* id *und*
- *eine (unendliche) Menge* $\{\underline{1}, \underline{2}, \dots\}$ *von De Bruijn-Indizes* ι

In [Meu09] treten in der Menge der Konstanten die Operatoren und der Fixpunktoperator auf. Diese entfernen wir und nehmen stattdessen einen Ausdruck der Form $e_1\ op\ e_2$ zur Kernsyntax hinzu. Wir definieren die Menge Op aller Operatoren op, $Const$ aller Konstanten c, $dbExp$ aller Ausdrücke e und $dbVal$ aller Werte v durch die folgende kontextfreie Grammatik:

$$
\begin{array}{rcl}
op & ::= & + \mid - \mid * \mid \leq \mid \geq \mid < \mid > \mid = \\
c & ::= & b \mid z \\
e & ::= & c \mid \iota \mid \lambda.\, e \mid e_1\ op\ e_2 \mid e_1\ e_2 \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \\
v & ::= & c \mid \lambda.\, e
\end{array}
$$

Durch Entfernung des Fixpunktoperators wird die (UNFOLD)-Regel unnötig. Wir entfernen also diese Regel und nehmen folgende Konstrukte als syntaktischen Zucker auf:

$$\begin{array}{rcl} (op) & =_{def} & \lambda.\,\lambda.\,\underline{2}\;op\;\underline{1} \\ \textbf{fix} & =_{def} & \lambda.\,(\lambda.\,\underline{2}\,(\underline{1}\,\underline{1}))\,(\lambda.\,\underline{2}\,(\underline{1}\,\underline{1})) \end{array}$$

Letztere Übersetzung entspricht dem vom Haskell Curry vorgestellten Fixpunktkombinator für call-by-name:

$$Y \;=_{def}\; \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x))$$

Definition: 2 (Closures und Umgebungen)**.** *Die Menge dbEnv aller De Bruijn-(Laufzeit)Umgebungen η und die Menge dbCl aller De Bruijn-Closures cl sind durch die folgende kontextfreie Grammatik definiert:*

$$\begin{array}{rcl} \eta & ::= & [\,] \mid cl;\eta \\ cl & ::= & (e,\eta) \end{array}$$

Definition: 3 (Big Step Regeln)**.** *Ein big step in der De Bruijn-Umgebungssemantik ist eine Formel der Gestalt $(e,\eta) \Downarrow (e',\eta')$, wobei $e, e' \in dbExp$ und $\eta, \eta' \in dbEnv$. Ein derartiger big step heißt gültig, wenn er sich mit den folgenden Regeln herleiten lässt:*

(VAL) $\quad (v,\eta) \Downarrow (v,\eta)$

(INDEX) $\quad \dfrac{\eta(\iota) \Downarrow cl}{(\iota,\eta) \Downarrow cl}$

(BETA) $\quad \dfrac{(e_1,\eta) \Downarrow (\lambda.\,e,\eta_1) \quad (e,(e_2,\eta);\eta_1) \Downarrow cl}{(e_1\,e_2,\eta) \Downarrow cl}$

(OP-1) $\quad \dfrac{(e_1,\eta) \Downarrow (op,\eta_1) \quad (e_2,\eta) \Downarrow (z,\eta_2)}{(e_1\,e_2,\eta) \Downarrow (op\,z,[])}$

(OP-2) $\quad \dfrac{(e_1,\eta) \Downarrow (op\,z_1,\eta_1) \quad (e_2,\eta) \Downarrow (z_2,\eta_2) \quad op^I(z_1,z_2) = z}{(e_1\,e_2,\eta) \Downarrow (z,[])}$

(COND-TRUE) $\quad \dfrac{(e_0,\eta) \Downarrow (true,\eta_0) \quad (e_1,\eta) \Downarrow cl}{(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2,\eta) \Downarrow cl}$

(COND-FALSE) $\quad \dfrac{(e_0,\eta) \Downarrow (false,\eta_0) \quad (e_2,\eta) \Downarrow cl}{(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2,\eta) \Downarrow cl}$

2 Lazy Categorical Abstract Machine

Für die im vorangegangenen Abschnitt beschriebene Programmiersprache soll Maschinencode erzeugt werden. Die Zielmaschine ist eine abgewandelte Version der *CAM (Categorical Abstract Machine)*, welche in [Hin99] beschrieben wird.

Wir definieren also eine abstrakte Maschine in die die Ausdrücke der funktionalen Programmiersprache $\mathbf{L_{db}}$ übersetzt werden. Wir nennen die hier beschriebene Maschine *LazyCAM*, da es sich um die Zielmaschine für eine Programmiersprache mit *lazy evaluation Semantik* handelt.

2.1 Architektur der LazyCAM

Bevor wir die Syntax, d. h. den Befehlssatz und die Umgebung, in der unsere Programme ablaufen, und die Semantik, d. h. die Bedeutung der Instruktionen, festlegen, wollen wir uns zunächst einmal der Architektur der LazyCAM zuwenden.

Die Architektur der LazyCAM besteht aus

1. einem Register,
2. einem Stack,
3. einem Heap und
4. einem Programmzähler.

Eine Besonderheit der LazyCAM ist, dass sie einen Heap benötigt. Wir wollen kurz darauf eingehen, warum dies der Fall ist und betrachten dazu das folgende Beispiel:

Gegeben sei eine Funktion *add* die zwei Parameter a und b entgegen nimmt. Wir wollen uns nun einmal den Hergang eines Funktionsaufrufes von *add* in einer herkömmlichen Programmiersprache, wie z. B. C, vor Augen führen:

Programmzeile	Anweisung
⋮	⋮
100	$add(5, 5)$;
⋮	⋮

Es stellt sich nun die Frage, was in unserer Maschine passiert, wenn dieser Aufruf stattfindet. Ein solcher Funktionsaufruf zieht folgende Schritte nach sich:

1. die Parameter werden auf den Stack gelegt
2. die Rücksprungadresse wird auf den Stack gelegt
3. der Funktionsaufruf wird durchgeführt und der entsprechede Code ausgeführt
4. nach dem Funktionsaufruf wird der betreffende Stack der Funktion *add* aufgegeben (steht nicht mehr zur Verfügung)
5. die Abarbeitung des Codes geht in der nächsten Programmzeile, nach *add*, weiter

Der hier entscheidende Punkt ist Nummer 4. Nach dem wir aus dem Funktionsaufruf zurückkehren, stehen die Parameter, welche im Stackbereich von *add* lagen, nicht mehr zur Verfügung, sie sind, wie man sagt, *out-of-scope* gegangen.

Betrachten wir nun folgendes, in einer funktionalen Sprache verfasstes, Beispiel der *add*-Funktion:

$$\textbf{let } add\ a\ b = a + b \textbf{ in } add\ 5$$

Die Funktion *add* wird hier mit nur *einem* Parameter aufgerufen. Das Resultat ist eine Funktion, welche

- noch einen Parameter b erwartet und
- zu diesem Parameter b den Wert 5 addiert

Doch wie wir eben gesehen haben, wird der zum Aufruf einer Funktion gehörige Stackbereich, nach Funktionsrückkehr ungültig und somit auch die Paramter, in diesem Falle die 5.

Wie wir an diesem Beispiel leicht sehen ist der Umstand, das Funktionen in funktionalen Sprachen als Ergebnis auftreten können, mit einer reinen Stackdisziplin nicht zu bewältigen. Um dieses Problem zu lösen, benötigt unsere Maschine einen *Heap*. Im Register stehen dann sogenannte Umgebungen (Zeiger auf eine Adresse im Heap), die die Bindungen freier Variablen beschreiben.

2.2 Syntax der LazyCAM

Definition: 4 (Syntax der Lazy Categorical Abstract Machine)**.** *Vorgegeben sei eine Menge* $Loc = \mathbb{N}$ *von Programmadressen* ℓ*. Die Menge Prog aller (LCAM-)Programme* P*, Instr aller (LCAM-)Instruktionen* I*, Reg aller (LCAM-)Registerinhalte* R *und Stack aller (LCAM-)Stackinhalte* S *sind wie folgt induktiv definiert:*

$$\begin{array}{rcl} P & ::= & I \mid P_1; P_2 \\ I & ::= & \textsc{Fst} \mid \textsc{Snd} \mid \textsc{Push} \mid \textsc{Swap} \mid \textsc{Cons} \mid \textsc{Quote}\, c \mid \textsc{Prim}\ op \\ & \mid & \textsc{Closure}\, \ell \mid \textsc{App} \mid \textsc{Eval} \mid \textsc{Return} \\ & \mid & \textsc{GotoFalse}\, \ell \mid \textsc{Goto}\, \ell \mid \textsc{Stop} \\ R & ::= & c \mid \ell \mid \underbrace{(R_1, R_2)}_{Umgebung} \mid \underbrace{[R : \ell]}_{Closure} \\ S & ::= & [\,] \mid R; S \end{array}$$

Die Länge $|P| \in \mathbb{N}$ *eines Programms* P *ist induktiv definiert durch:*

$$\begin{array}{rcl} |I| & = & 1 \\ |P_1; P_2| & = & |P_1| + |P_2| \end{array}$$

Jede Instruktion I eines Programmes P besitzt eine eindeutige Programmadresse ℓ, identifiziert durch die Position an der I in P steht, beginnend mit 0. Die Menge $dom(P) \subseteq Loc$ aller gültigen Adressen eines Programmes P ist definiert durch:

$$dom(P) = \{\ell \in Loc | 0 \leq \ell \wedge \ell < |P|\}$$

Wir schreiben $P[\ell]$ für die Instruktion I an der Programmadresse $\ell \in dom(P)$ im Programm P. Weiterhin schreiben wir $\ell : I$ für die Programmadresse $\ell \in dom(P)$ mit $P[\ell] = I$.

Definition: 5 (Relative Sprungziele und gültige Programme)**.** *Die Menge* $locns\,(I, \ell) \subseteq Loc$ *aller relativ zu* ℓ *in* I *vorkommenden Sprungziele (implizit oder explizit) ist wie folgt definiert:*

$$\begin{array}{rcl} locns\,(\textsc{Closure}\, \ell', \ell) & = & \{\ell', \ell + 1\} \\ locns\,(\textsc{Return}, \ell) & = & \emptyset \\ locns\,(\textsc{Goto}\, \ell', \ell) & = & \{\ell'\} \\ locns\,(\textsc{GotoFalse}\, \ell', \ell) & = & \{\ell', \ell + 1\} \\ locns\,(\textsc{Stop}, \ell) & = & \emptyset \\ locns\,(I, \ell) & = & \{\ell + 1\} \quad \textit{für alle übrigen } I \end{array}$$

Die Menge $locns\,(P, \ell) \subseteq Loc$ *aller relativ zu* ℓ *in* P *vorkommenden Sprungziele (implizit oder explizit) ist induktiv definiert durch:*

$$\begin{aligned} locns\,(I, \ell) &= locns\,(I, \ell) \\ locns\,(P_1; P_2, \ell) &= locns\,(P_1, \ell) \cup locns\,(P_2, \ell + |P_1|) \end{aligned}$$

Statt $locns\,(P, 0)$ *schreiben wir auch kurz* $locns\,(P)$*. Ein Programm* $P \in Prog$ *heißt gültig, wenn* $locns\,(P) \subseteq dom(P)$*.*

Intuitiv sollte sofort ersichtlich sein, dass ein gültiges Programm immer entweder mit STOP, RETURN oder GOTO enden muss, da in der Sprungmenge sonst die Nachfolgeadresse des letzten Befehls enthalten wäre, die aber nicht mehr zum Programm gehört. Obige Definition ist schon teilweise ein Vorgriff auf die Semantik der Maschine.

Definition: 6. *Die Menge* $locns\,(R) \subseteq Loc$ *aller im Register* R *vorkommenden Sprungziele ist wie folgt induktiv definiert:*

$$\begin{aligned} locns\,(c) &= \emptyset \\ locns\,(\ell) &= \{\ell\} \\ locns\,([R : \ell]) &= locns\,(R) \cup locns\,(\ell) \\ locns\,((R_1, R_2)) &= locns\,(R_1) \cup locns\,(R_2) \end{aligned}$$

Ein Registerinhalt R *heißt gültig für ein Programm* P*, wenn* $locns\,(R) \subseteq dom(P)$*. Die Menge aller gültigen Registerinhalte für* P *bezeichnen wir mit* $Reg(P)$*.*

Definition: 7. *Analog ist die Menge* $locns\,(S) \subseteq Loc$ *aller auf dem Stack* S *vorkommenden Sprungziele definiert:*

$$\begin{aligned} locns\,([\,]) &= \emptyset \\ locns\,(R; S) &= locns\,(R) \cup locns\,(S) \end{aligned}$$

Ein Stackinhalt S *heißt gültig für ein Programm* P*, wenn* $locns\,(S) \subseteq dom(P)$*. Die Menge aller gültigen Stackinhalte für* P *bezeichnen wir mit* $Stack(P)$*.*

2.3 Semantik der LazyCAM

Definition: 8 (Semantik der LazyCAM)**.** *Sei* $P \in Prog$ *ein gültiges Programm. Ein gültiger (LCAM-)Programmzustand für* P *ist ein Tripel der Form* $(R, S, \ell) \in Reg \times Stack \times Loc$*, mit*

$$locns\,(R) \cup locns\,(S) \cup \{\ell\} \subseteq dom(P)$$

Die Menge aller gültigen Programmzustände für P *bezeichnen wir mit* $Stack(P)$*. Die Übergangsrelation* $\rightarrow$ *der LazyCAM ist in Tabelle 1 beschrieben.*

Zustand		Nachfolger	Beschreibung
$((R_1, R_2), S, \ell : \textsc{Fst})$	$\rightarrow$	$(R_1, S, \ell + 1)$	Erste Projektion
$((R_1, R_2), S, \ell : \textsc{Snd})$	$\rightarrow$	$(R_2, S, \ell + 1)$	Zweite Projektion
$(R, S, \ell : \textsc{Push})$	$\rightarrow$	$(R, R; S, \ell + 1)$	Registerinhalt auf Stack
$(R_1, R_2; S, \ell : \textsc{Swap})$	$\rightarrow$	$(R_2, R_1; S, \ell + 1)$	Registerinhalt und oberstes Stackelement vertauschen
$(R_2, R_1; S, \ell : \textsc{Cons})$	$\rightarrow$	$((R_1, R_2), S, \ell + 1)$	Paarbildung
$(R, S, \ell : \textsc{Quote}\, c)$	$\rightarrow$	$(c, S, \ell + 1)$	c in Register laden
$(z_2, z_1; S, \ell : \textsc{Prim}\, op)$	$\rightarrow$	$(op^I(z_1, z_2), S, \ell + 1)$	Operation op ausführen
$(R, S, \ell : \textsc{Closure}\, \ell')$	$\rightarrow$	$([R : \ell'], S, \ell + 1)$	Abschluss bilden
$([R_1 : \ell'], R_2; S, \ell : \textsc{App})$	$\rightarrow$	$((R_1, R_2), \ell + 1; S, \ell')$	Funktionsaufruf
$([R : \ell'], S, \ell : \textsc{Eval})$	$\rightarrow$	$(R, \ell + 1; S, \ell')$	Verzögerte Auswertung einer Closure
$(R, \ell'; S, \ell : \textsc{Return})$	$\rightarrow$	(R, S, ℓ')	Rücksprung aus Closure
$(\mathit{true}, R; S, \ell : \textsc{GotoFalse}\, \ell')$	$\rightarrow$	$(R, S, \ell + 1)$	Sprung zu ℓ' wenn Register
$(\mathit{false}, R; S, \ell : \textsc{GotoFalse}\, \ell')$	$\rightarrow$	(R, S, ℓ')	den Wert *false* enthält
$(R, S, \ell : \textsc{Goto}\, \ell')$	$\rightarrow$	(R, S, ℓ')	Unbedingter Sprung zu ℓ'

Abbildung 2.1: Übergangsrelation

Lemma 1 (Wohldefiniertheit der Übergangsrelation)**.** *Sei P ein gültiges Programm. Wenn $(R, S, \ell) \in \mathit{State}(P)$ und $(R, S, \ell) \rightarrow (R', S', \ell')$, dann gilt auch $(R', S', \ell') \in \mathit{State}(P)$.*

Proof. Wir beweisen das Lemma durch Induktion über die Länge der Herleitung und Fallunterscheidung nach der an Adresse ℓ stehenden Instruktion:

> $\underline{\ell : \textsc{Closure}\, \ell''}$: Nach Voraussetzung ist (R, S, ℓ) gültig für P und es existiert $(R, S, \ell) \rightarrow (R', S', \ell')$.
>
> - Der Stack bleibt unverändert $\Rightarrow S'$ gültig für P.
> - Nach Definition 5 folgt noch mindestens ein Befehl $\Rightarrow \ell' = \ell + 1$ gültig für P.
> - P gültig $\Rightarrow \ell''$ gültig für P und da R gültig für $P \Rightarrow [R : \ell''] = R'$ gültig für P
>
> Also folgt: (R', S', ℓ') gültig für P.
>
> $\underline{\ell : \textsc{App}}$: Nach Voraussetzung ist (R, S, ℓ) gültig für P und es existiert $(R, S, \ell) \rightarrow (R', S', \ell')$. Insbesondere existieren R_1 und R_2, so dass $R = [R_1 : \ell'']$ und $S = R_2; S''$.
>
> - $R = [R_1 : \ell'']$ gültig für $P \Rightarrow \ell' = \ell''$ gültig für P

- $S = R_2; S''$ gültig für $P \Rightarrow R_2$ gültig für P und da R_1 gültig für $P \Rightarrow R' = (R_1, R_2)$ gültig für P
- Nach Definition 5 folgt noch mindestens ein Befehl $\Rightarrow \ell + 1$ gültig für P
- Da S'' gültig $\Rightarrow S' = \ell + 1; S''$ gültig für P

Also folgt: (R', S', ℓ') gültig für P.

ℓ : EVAL: Nach Voraussetzung ist (R, S, ℓ) gültig für P und es existiert $(R, S, \ell) \rightarrow (R', S', \ell')$. Insbesondere existieren R'' und ℓ'', so dass $R = [R'' : \ell'']$ und es gilt:

- Da $R = [R'' : \ell'']$ gültig für $P \Rightarrow \ell' = \ell''$ und $R' = R''$ gültig für P.
- Nach Definition 5 folgt noch mindestens ein Befehl $\Rightarrow \ell + 1$ gültig für P
- Da S und $\ell + 1$ gültig für $P \Rightarrow S' = \ell + 1; S$ gültig für P

Also folgt: (R', S', ℓ') gültig für P.

ℓ : RETURN: Nach Voraussetzung ist (R, S, ℓ) gültig für P und es existiert $(R, S, \ell) \rightarrow (R', S', \ell')$. Insbesondere existieren ℓ'' und S'', mit $S = \ell''; S''$ und es gilt:

- Da R gültig für $P \Rightarrow R' = R$ gültig für P
- und da $S = \ell''; S''$ gültig für $P \Rightarrow S' = S''$ und $\ell' = \ell''$ gültig für P

Also folgt: (R', S', ℓ') gültig für P.

ℓ : GOTO ℓ'': Nach Voraussetzung ist (R, S, ℓ) gültig für P und es existiert $(R, S, \ell) \rightarrow (R', S', \ell')$. Dann gilt:

- Da R gültig für $P \Rightarrow R' = R$ gültig für P
- Da S gültig für $P \Rightarrow S' = S$ gültig für P
- Da ℓ : GOTO ℓ'' gültige Instruktion für $P \Rightarrow \ell''$ und somit $\ell' = \ell''$ gültig für P

Also folgt: (R', S', ℓ') gültig für P.

Die restlichen Fälle folgen analog. □

Lemma 2 (Eindeutigkeit der Übergangsrelation)**.** *Sei P ein gültiges Programm. Für jeden Zustand $s \in State(P)$ existiert höchstens ein $s' \in State(P)$, so dass $s \rightarrow s'$.*

Proof. Offentsichtlich gilt: Wenn ein P gültiges Programm ist, dann steht an jeder Programmadresse ein Befehl und somit folgt aus Lemma 1, dass $s' \in State(P)$. □

Definition: 9 (Berechnung für die LazyCAM)**.** *Sei P ein gültiges Programm. Eine Berechnung für P ist eine (möglicherweise unendliche) Folge von Zuständen*

$$(R_0, S_0, 0) \to (R_1, S_1, \ell_1) \to (R_2, S_2, \ell_2) \to \ldots$$

Eine Berechnung heißt

1. terminierend*, wenn sie endlich ist und der letzte Zustand die Form* $(R_e, S_e, \ell_e : \textsc{Stop})$ *hat, oder*

2. divergierend*, wenn sie unendlich ist, oder*

3. stecken bleibend*, sonst.*

3 Erzeugung von Zielcode

Wie im Kapitel „Einführung“ auf Seite 1 ff. beschrieben, haben wir die Programmiersprache, wie sie in [Meu09] beschrieben wird, zugunsten einer einfacheren Übersetzung abgeändert, in dem wir die Operatoren und den Fixpunktoperator aus der Menge der Konstanten, sowie die (UNFOLD)-Regel entfernt und durch einfachere Konstrukte ersetzt haben. Es ist nun basierend auf der big step Semantik zu zeigen, dass diese Übersetzung, d. h. die Einführung des Infixoperators *op* und des Fixpunktoperators wie sie auf Seite 2 beschrieben, bezüglich der zuvor benutzten Semantik, korrekt ist.

3.0.1 Die Übersetzungsfunktionen $\mathcal{P}, \mathcal{C}$ und $\mathcal{V}$

Die *LazyCAM*-Codegenerierung für einen Ausdruck $e \in dbExp$ wird durch die folgenden drei Funktionen erledigt:

1. $\mathcal{P}[\![e]\!]$ erzeugt das Programm für e, d. h. den Code zur Berechnung des Wertes von e und den abschliessenden STOP-Befehl.

2. $\mathcal{C}[\![e]\!]\ell$ erzeugt Code, beginnend ab Programmadresse ℓ, welcher erwartet, dass R die Umgebung enthält, in der e ausgewertet werden soll und am Ende eine Closure (mit der zuvor in R befindlichen Umgebung) für e in R ablegt.

3. $\mathcal{V}[\![e]\!]\ell$ erzeugt Code, beginnend ab Programmadresse ℓ, welcher erwartet, dass R die Umgebung enthält, in der e ausgewertet werden soll und am Ende den Wert von e in R ablegt.

Diese Funktionen sind wie folgt induktiv definiert:

$$
\begin{array}{rcl}
\mathcal{P}[\![e]\!] & = & \mathcal{V}[\![e]\!]0;\ \textsc{Stop} \\
\\
\mathcal{C}[\![e]\!]\ell & = & \textsc{Goto}\,\ell';\ P;\ \textsc{Return};\ \textsc{Closure}\,(\ell+1) \\
 & & \text{mit } P = \mathcal{V}[\![e]\!](\ell+1), \ell' = \ell + |P| + 2 \\
\\
\mathcal{V}[\![c]\!]\ell & = & \textsc{Quote}\,c \\
\mathcal{V}[\![i]\!]\ell & = & \underbrace{\textsc{Fst};\ldots;\textsc{Fst}}_{(i-1)\text{-mal}};\ \textsc{Snd};\ \textsc{Eval} \\
\mathcal{V}[\![\lambda.\, e]\!]\ell & = & \mathcal{C}[\![e]\!]\ell \\
\mathcal{V}[\![e_1\, e_2]\!]\ell & = & \textsc{Push};\ P_1;\ \textsc{Swap};\ P_2;\ \textsc{Swap};\ \textsc{App} \\
 & & \text{mit } P_1 = \mathcal{V}[\![e_1]\!](\ell+1), P_2 = \mathcal{C}[\![e_2]\!](\ell + |P_1| + 2) \\
\mathcal{V}[\![e_1\ op\ e_2]\!]\ell & = & \textsc{Push};\ P_1;\ \textsc{Swap};\ P_2;\ \textsc{Prim}\ op \\
 & & \text{mit } P_1 = \mathcal{V}[\![e_1]\!](\ell+1), P_2 = \mathcal{V}[\![e_2]\!](\ell + |P_1| + 2) \\
\mathcal{V}[\![\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2]\!]\ell & = & \textsc{Push};\ P_0;\ \textsc{GotoFalse}\,\ell';\ P_1;\ \textsc{Goto}\,\ell'';\ P_2 \\
 & & \text{mit } P_0 = \mathcal{V}[\![e_0]\!](\ell+1), P_1 = \mathcal{V}[\![e_1]\!](\ell + |P_0| + 2), \\
 & & P_2 = \mathcal{V}[\![e_2]\!]\ell', \ell' = \ell + |P_0| + |P_1| + 3, \ell'' = \ell' + |P_2|
\end{array}
$$

Es stellt sich nun noch die Frage nach der Wohldefiniertheit der Übersetzungsfunktionen. Es gilt also, uns davon zu überzeugen, dass die Anwendung der Übersetzungsfunktionen auf ein $e \in \mathit{dbExp}$ zu einer Übersetzung in ein gültiges Programm P_e führt.

Bevor wir jedoch die Wohldefiniertheit der Übersetzungsfunktionen zeigen können, müssen wir zunächst einmal eine passende Formulierung für $\mathcal{C}$ und $\mathcal{V}$ finden, was wir mit dem folgenden Lemma tun.

Lemma 3. *Sei $e \in \mathit{dbExp}$. Dann gilt:*

$$\mathit{locns}\,(P) \subseteq \mathit{dom}(P) \cup \{|P| + 1\}$$

mit $P = \mathcal{V}[\![e]\!]\ell$ und $\ell \in \mathit{dom}(P)$.

Proof. Wir beweisen das Lemma durch Induktion über die Größe von e und Fallunterscheidung nach der Form von e.

$\underline{e = c}$: Dann gilt $P = \mathcal{V}[\![c]\!]\ell = \textsc{Quote}\,c$ und also

$$\mathit{locns}\,(P, \ell) = \{\ell + 1\} \subseteq \mathit{dom}(P) \cup \{|P| + 1\} = \{\ell, \ell + 1\}$$

$\underline{e = \iota}$: Dann gilt $P = \mathcal{V}[\![\iota]\!]\ell = \underbrace{\textsc{Fst};\ldots\textsc{Fst};}_{(i-1)\text{ mal}}\textsc{Snd};\textsc{Eval}$ und also

$$\begin{aligned} \mathit{locns}\,(P,\ell) &= \mathit{locns}\,(\textsc{Fst},\ell)\cup\cdots\cup\mathit{locns}\,(\textsc{Fst},\ell+(i-1)) \\ &\quad \cup\mathit{locns}\,(\textsc{Snd},\ell+i)\cup\mathit{locns}\,(\textsc{Eval},\ell+i+1) \\ &= \{\ell+1,\ldots,\ell+i+2\} \\ &\subseteq \mathit{dom}(P)\cup\{|P|+1\} \end{aligned}$$

$\underline{e = \lambda.\, e'}$:
Dann gilt $P = \mathcal{V}[\![\lambda.\, e]\!]\ell = \mathcal{C}[\![\lambda.\, e]\!]\ell = \textsc{Goto}\,\ell'; P_1; \textsc{Return}; \textsc{Closure}\,(\ell+1)$ und also

$$\begin{aligned} \mathit{locns}\,(P,\ell) &= \mathit{locns}\,(\textsc{Goto}\,\ell',\ell)\cup\mathit{locns}\,(P_1,\ell+1) \\ &\quad \cup\mathit{locns}\,(\textsc{Return},\ell+|P_1|+1) \\ &\quad \cup\mathit{locns}\,(\textsc{Closure}(\ell+1),\ell+|P_1|+2) \\ &= \{\ell+1,\ldots,\ell+|P_3|+3\} \\ &\subseteq \mathit{dom}(P)\cup\{|P|+1\} \end{aligned}$$

mit $P_1 = \mathcal{V}[\![e]\!](\ell+1)$ und $\ell' = \ell+|P_1|+2$

$\underline{e = e_1\, e_2}$:
Dann muss e_1 von der Form $e_1 = \lambda.\, e$ sein und es gilt
$P = \mathcal{V}[\![e_1\, e_2]\!]\ell = \textsc{Push}; P_1; \textsc{Swap}; P_2; \textsc{Swap}; \textsc{App}$ und also

$$\begin{aligned} \mathit{locns}\,(P,\ell) &= \mathit{locns}\,(\textsc{Push},\ell)\cup\mathit{locns}\,(P_1,\ell+1) \\ &\quad \cup\mathit{locns}\,(\textsc{Swap},\ell+|P_1|+1)\cup\mathit{locns}\,(P_2,\ell+|P_1|+2) \\ &\quad \cup\mathit{locns}\,(\textsc{Swap},\ell+|P_1|+|P_2|+3)\cup\mathit{locns}\,(\textsc{App},\ell+|P_1|+|P_2|+4) \\ &= \{\ell+1,\ldots,\ell+|P_1|+|P_2|+5\} \\ &\subseteq \mathit{dom}(P)\cup\{|P|+1\} \end{aligned}$$

mit $P_1 = \mathcal{V}[\![e_1]\!](\ell+1), P_2 = \mathcal{C}[\![e_2]\!](\ell+|P_1|+2)$

$\underline{e = e_1\ op\ e_2}$:
Dann gilt $P = \mathcal{V}[\![e_1\ op\ e_2]\!]\ell = \textsc{Push}; P_1; \textsc{Swap}; P_2; \textsc{Prim}\ op$ und also

$$\begin{aligned} locns\,(P,\ell) &= locns\,(\textsc{Push},\ell) \cup locns\,(P_1,\ell+1) \\ &\quad \cup locns\,(\textsc{Swap},\ell+|P_1|+1) \cup locns\,(P_2,\ell+|P_1|+2) \\ &\quad \cup locns\,(\textsc{Prim}\ op,\ell+|P_1|+|P_2|+3) \\ &= \{\ell+1,\ldots,\ell+|P_1|+|P_2|+4\} \\ &\subseteq dom(P) \cup \{|P|+1\} \end{aligned}$$

mit $P_1 = \mathcal{V}[\![e_1]\!](\ell+1), P_2 = \mathcal{V}[\![e_2]\!](\ell+|P_1|+2)$

$\underline{e = \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2}$: Dann gilt
$P = \mathcal{V}[\![\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]\ell = \textsc{Push}; P_0; \textsc{GotoFalse}\,\ell'; P_1; \textsc{Goto}\,\ell''; P_2$ und also:

$$\begin{aligned} locns\,(P,\ell) &= locns\,(\textsc{Push},\ell) \cup\ locns\,(P_0,\ell+1) \\ &\quad \cup\ locns\,(\textsc{GotoFalse}\,\ell',\ell+|P_0|+1) \\ &\quad \cup\ locns\,(P_1,\ell+|P_0|+2) \cup\ locns\,(\textsc{Goto}\,\ell'',\ell+|P_0|+|P_1|+2) \\ &\quad \cup\ locns\,(P_2,\ell+|P_0|+|P_1|+3) \\ &= \{\ell+1,\ldots,\ell+|P_0|+|P_1|+|P_2|+4\} \\ &\subseteq dom(P) \cup \{|P|+1\} \end{aligned}$$

mit $P_0 = \mathcal{V}[\![e_1]\!](\ell+|P_0|+2)$,$P_2 = \mathcal{V}[\![e_2]\!]\ell'$, $\ell' = \ell+|P_0|+|P_1|+3$,$\ell'' = \ell'+|P_2|$

□

Korollar 1 (Wohldefiniertheit der Übersetzungsfunktion)**.** *Sei $e \in dbExp$. Dann ist $\mathcal{P}[\![e]\!]$ ein gültiges Programm.*

Proof. Dies folgt unmittelbar aus Lemma 3 und der Tatsache, dass an der letzten Programmadresse die Instruktion Stop steht. □

Anmerkungen

Diese Übersetzung ist sehr ineffizient, denn z. B. werden für λ-Ausdrücke auf Parameterposition zwei Closures erzeugt oder für einfache Konstanten wird auf Parameter-position ebenfalls immer eine Closure erzeugt, obwohl dies nicht notwendig wäre. Noch sinnloser sind die De Bruijn-Indizes, hier wird dann eine Closure erzeugt, welche die Closure

aus der Umgebung holt und auswertet. Letzteres liesse sich durch geschickte Änderung der Übersetzungsfunktionen regeln. Das allgemeine Problem unnötiger Closures könnte man dadurch lösen, dass man zwischen Closure und Funktionswert unterscheidet und entsprechend die EVAL-Instruktion abändert, wie dies im [WM92] gemacht wird (allerdings dann ohne call-by-need semantik). Auf einige Verbesserungsmöglichkeiten werden wir an späterer Stelle noch einmal zurückkommen.

Zunächst wollen wir aber diese sehr naive Übersetzung nehmen und beweisen, dass diese semantikerhaltend ist. Dazu müssen wir zunächst einmal geeignete Invarianten bestimmen. Intuitiv muss folgendes gezeigt werden:

Wenn $(e, [\,]) \Downarrow (v, \eta)$, dann existiert für $P = \mathcal{P}[\![e]\!]$ eine terminierende Berechnung $(R_0, S_0, 0) \xrightarrow{*} (R, S, \ell : \textsc{Stop})$, so dass R zu $(v\,\eta)$ passt.

3.0.2 Korrektheit der Codegenerierung

Definition: 10. *Die Menge $Reg(\eta, P)$ der* gültigen Register-Repräsentationen *der Umgebung η im Programm P ist induktiv durch*

$$\begin{array}{rcl} Reg([\,], P) & = & Reg(P) \\ Reg((e,\eta);\eta', P) & = & \{(R', [R : \ell]) \mid R' \in Reg(\eta', P) \\ & & \wedge R \in Reg(\eta, P) \wedge P = I_0; \ldots; I_{\ell-1}; \mathcal{V}[\![e]\!]\ell; \textsc{Return}; \ldots\} \end{array}$$

und die Menge $Reg(v, \eta, P)$ der gültigen Register-Repräsentationen *des Wertes v im Programm P ist induktiv durch*

$$\begin{array}{rcl} Reg(c, \eta, P) & = & \{c\} \\ Reg(\lambda.\, e, \eta, P) & = & \{[R : \ell] \mid R \in Reg(\eta, P) \\ & & \wedge P = I_0; \ldots; I_{\ell-1}; \mathcal{V}[\![e]\!]\ell; \textsc{Return}; \ldots\} \end{array}$$

definiert.

Lemma 4. *Sei $(e, \eta) \in dbCl, P_e = \mathcal{V}[\![e]\!]\ell, P = I_0; \ldots; I_{\ell-1}; P_e; \ldots; \textsc{Stop}$ ein gültiges Programm, $R \in Reg(\eta, P)$ und $S \in Stack(P)$. Dann gilt:*

> *Wenn $(e, \eta) \Downarrow (v', \eta')$, dann existiert eine Berechnung*
> *$(R, S, \ell) \xrightarrow{*} (R', S, \ell + |P_e|)$ mit $R' \in Reg(v', \eta', P)$.*

- *Im Falle von* (VAL) *gilt $\eta = \eta'$ und entweder $e = c$ oder $e = \lambda.\, e'$.*

  Für $e = c$ gilt $P_e = \textsc{Quote}\, c$, also $(R, S, \ell) \to (c, S, \ell + 1)$. Weiterhin gilt $v' = c \in Reg(c, \eta', P)$.

Für $e = \lambda.\, e'$ gilt $P_e = \textsc{Goto}\,\ell'; P_{e'}; \textsc{Return}; \textsc{Closure}\,(\ell+1)$ mit $P_{e'} = \mathcal{V}[\![e']\!](\ell+1)$ und $\ell' = \ell + |P_{e'}| + 2$. Dafür ergibt sich:

$$(R, S, \ell) \;\to\; (R, S, \ell') \to ([R : (\ell+1)], S, \ell'+1)$$

Also gilt $P = I_0; \ldots; \mathcal{I}_{\ell-1}; \textsc{Goto}\,\ell'; P_{e'}; \textsc{Return}; \ldots$ und somit folgt unmittelbar, dass $[R : (\ell+1)] \in Reg(\lambda.\, e', \eta, P)$.

- (Index) *als letzte Regel impliziert $e = \iota$ mit $\eta(\iota) = (e_i, \eta_i) \Downarrow (v', \eta')$, und $\eta = [(e_1, \eta_1); \ldots; (e_i, \eta_i); \ldots]$. Weiter gilt*

$$P_e = \underbrace{\textsc{Fst}; \ldots; \textsc{Fst}}_{(i-1)\text{-}mal}; \textsc{Snd}; \textsc{Eval}$$

und

$$R = ((\ldots(R'', [R_i : \ell_i]), \ldots), [R_1 : \ell_1]).$$

Daraus ergibt sich:

$$\begin{array}{lll} (R, S, \ell) & \xrightarrow{(i-1)} & ((R'', [R_i : \ell_i]), S, \ell + i - 1) \\ & \to & ([R_i : \ell_i], S, \ell + i) \\ & \to & (R_i, (\ell + |P_e|); S, \ell_i) \end{array}$$

Nach Voraussetzung steht $P_i = \mathcal{V}[\![e_i]\!]\ell_i; \textsc{Return}$ an Adresse ℓ_i in P und es gilt $R_i \in Reg(\eta_i, P)$. Entsprechend folgt nach Induktionsvoraussetzung

$$\begin{array}{lll} (R_i, (\ell + |P_e|); S, \ell_i) & \xrightarrow{*} & (R', (\ell + |P_e|); S, \ell_i + |P_i|) \\ & \to & (R', S, \ell + |P_e|) \end{array}$$

wobei $R' \in Reg(v', \eta', P)$.

- *Für (Op) gilt $e = e_1\, op\, e_2$, $(e_1, \eta) \Downarrow (z_1, \eta_1)$, $(e_2, \eta) \Downarrow (z_2, \eta_2)$, $v' = op^I(z_1, z_2)$ und $\eta = [\,]$. Weiterhin gilt $P_e = \textsc{Push}; P_1; \textsc{Swap}; P_2; \textsc{Prim}\, op$ mit $P_1 = \mathcal{V}[\![e_1]\!](\ell+1)$ und $P_2 = \mathcal{V}[\![e_2]\!](\ell + |P_1| + 2)$. Daraus folgt unmittelbar*

$$\begin{array}{lll} (R, S, \ell) & \to & (R, R; S, \ell+1) \\ & \xrightarrow{*} & (z_1, R; S, \ell + |P_1| + 1) \quad \textit{wg. I.V.} \\ & \to & (R, z_1; S, \ell + |P_1| + 2) \\ & \xrightarrow{*} & (z_2, z_1; S, \ell + |P| - 1) \quad \textit{wg. I.V.} \\ & \to & (op^I(z_1, z_2), S, \ell + |P|) \end{array}$$

und trivialerweise gilt $op^I(z_1, z_2) \in Reg(op^I(z_1, z_2), [\,], P)$.

- *Für* (BETA) *gilt* $e = e_1 e_2$ *und*
  - $(e_1, \eta) \Downarrow (\lambda.\, e', \hat{\eta})$
  - $(e', (e_2, \eta); \hat{\eta}) \Downarrow (v', \eta')$

  Ferner gilt $P_e = \text{PUSH}; P_1; \text{SWAP}; P_2; \text{SWAP}; \text{APP}$ *mit*
  - $P_1 = \mathcal{V}[\![e_1]\!](\ell + 1)$
  - $P_2 = \mathcal{C}[\![e_2]\!](\ell + |P_1| + 2)$

  und somit:

$$
\begin{array}{lll}
 & (R, S, \ell) & \\
\to & (R, R; S, \ell + 1) & \textit{mit } \text{PUSH} \\
\xrightarrow{I.V.} & (\hat{R}, R; S, \ell + 1 + |P_1|) & \textit{mit } \hat{R} \in \textit{Reg}(\lambda.\, e', \hat{\eta}, P) \\
 & & \Rightarrow \exists \ell_0 \in \textit{dom}(P) \wedge\ R_0 \in \textit{Reg}(\hat{\eta}, P), \\
 & & \textit{so dass } \hat{R} = [R_0 : \ell_0] \\
\to & (R, \hat{R}; S, \ell + |P_1| + 2) & \textit{mit } \text{SWAP} \\
\to & (R, \hat{R}; S, \ell + |P_1| + |P_{e_2}| + 4) & \textit{mit } \text{GOTO} \\
\to & ([R : \ell + |P_1| + 3], \hat{R}; S, \ell + |P_1| + |P_{e_2}| + 5) & \textit{mit } \text{CLOSURE} \\
\to & (\hat{R}, [R : \ell + |P_1| + 3]; S, \ell + |P_1| + |P_{e_2}| + 6) & \textit{mit } \text{SWAP} \\
\xrightarrow{(1)} & ((R_0, [R : \ell + |P_1| + 3]), \ell + |P_1| + |P_{e_2}| + 7; S, \ell_0) & \textit{mit } \text{APP} \\
\xrightarrow[(2)]{I.V.} & (R', \ell + |P_1| + |P_{e_2}| + 8; S, \ell_0 + |P_{e_1}|) & \textit{mit } R' \in \textit{Reg}(v', \eta', P) \\
\to & (R', S, \ell + |P_1| + |P_{e_2}| + 8) & \textit{mit } \text{RETURN}
\end{array}
$$

  Es bleibt noch (1) zu zeigen, d. h.:
  $(R_0, [R : \ell + |P_1| + 3]) \in \textit{Reg}((e_2, \eta); \hat{\eta}, P)$ *gilt wegen*
  - $R_0 \in \textit{Reg}(\hat{\eta}, P)$
  - $R \in \textit{Reg}(\eta, P)$

- (COND-TRUE)*, dann gilt:* $e =$ **if** e_0 **then** e_1 **else** e_0*,* $(e_0, \eta) \Downarrow (\textit{true}, \eta_0)$*,*
  $(e_1, \eta) \Downarrow \textit{cl}(= (v', \eta'))$*,* $P_e = \text{PUSH}; P_0; \text{GOTOFALSE}\, \ell'; P_1; \text{GOTO}\, \ell''; P_2$ *Ferner gilt:*
  - $P_0 = \mathcal{V}[\![e_0]\!](\ell + 1)$
  - $P_1 = \mathcal{V}[\![e_1]\!](\ell + |P_0| + 2)$
  - $P_2 = \mathcal{V}[\![e_2]\!]\ell'$ *mit* $\ell' = \ell + |P_0| + |P_1| + 3$
  - $\ell'' = \ell' + |P_2|$

und somit:

$$\begin{array}{rcl}(R, S, \ell) & \to & (R_1, R_1; S, \ell + 1) \\ & \xrightarrow{*} & (\mathit{true}, R_1; S, \ell + |P_0| + 1) \quad \textit{wg. I.V.} \\ & \to & (R_1, S, \ell + |P_0| + 2) \\ & \xrightarrow{*} & ([R_1 : \ell + |P_0| + 4], S, \ell' - 1) \quad \textit{wg. I.V.} \\ & \to & ([R_1 : \ell + |P_0| + 4], S, \ell'')\end{array}$$

- (COND-FALSE) *analog.*

Satz 5 (Korrektheit der Codeerzeugung)**.** *Sei $e \in \mathit{dbExp}$ mit $\mathit{free}\,(e) = \emptyset$, $v \in \mathit{dbVal}$, $\eta \in \mathit{dbEnv}$, $P = \mathcal{P}[\![e]\!]$, $R \in \mathit{Reg}(P)$ und $S \in \mathit{Stack}(P)$. Wenn $(e, [\,]) \Downarrow (v, \eta)$, dann existiert genau eine (terminierende) Berechnung*

$$(R, S, 0) \xrightarrow{*} (R_v, S, |P| - 1),$$

und es gilt $R_v \in \mathit{Reg}(v, \eta, P)$.
Insbesondere gilt für $v = c$, dass genau eine Berechnung

$$(R, S, 0) \xrightarrow{*} (c, S, |P| - 1)$$

existiert.

Proof. Folgt unmittelbar aus dem vorangegangenen Lemma und der Eindeutigkeit der LazyCAM Semantik. □

4 Verbesserung der Codegenerierung

4.0.3 Closurebildung minimieren

Die naive Übersetzung von Ausdrücken führt zu einem einfachen Korrektheitsbeweis, hat allerdings den nicht unerheblichen Nachteil, dass der generierte Code sehr ineffizient ist, wie wir es im letzten Abschnitt bereits angemerkt haben. Dies wird vorallem dadurch bedingt, dass zu viele unnötige Closures erzeugt werden.

Der erste Ansatzpunkt ist die Umsetzung der (INDEX)-Regel. Steht ein De Bruijn-Index auf Closure-Position, so wird Closure-Code erzeugt, der zunächst die für den Index eingetragene Closure aus der Umgebung holt und anschliessend auswertet. Stattdessen könnte man einfach die existierende Closure aus der Umgebung nehmen. Dazu werden $\mathcal{C}$ und $\mathcal{V}$ wie folgt geändert:

$$\begin{aligned} \mathcal{C}[\![\iota]\!]\ell &= \underbrace{\textsc{Fst};\ldots;\textsc{Fst}}_{(\iota-1)\text{-mal}};\textsc{Snd} \\ \mathcal{V}[\![\iota]\!]\ell &= \mathcal{C}[\![\iota]\!];\textsc{Eval} \end{aligned}$$

Nun wird für einen De Bruijn-Index auf Closure-Position nur noch der Zugriff auf die Umgebung erzeugt, für einen Index auf Value-Position ändert sich nichts. Es ist offentsichtlich, dass die Übersetzung immer noch korrekt ist.

4.0.4 Rekursion

Der Fixpunktoperator wird bislang sehr ineffizient übersetzt. Zur Erinnerung, der Fixpunktoperator war wie folgt definiert:

$$Y = \lambda f.\,(\lambda x.\,f(x\,x))(\lambda x.\,f(x\,x))$$

Nun würde bei der Übersetzung folgendes generiert:

- eine Closurebildung für den Rumpf von f
- zwei Closurebildungen für den Rumpf von x

Zusätzlich spielen noch die Ausführungen aus dem letzten Abschnitt eine Rolle.

Statt des Fixpunktoperators könnte man einen speziellen Ausdruck für Rekursion einbauen, wie in [Pie02] beschrieben, d. h.

$$e \quad ::= \quad \mathbf{rec}\ .\ e$$

mit folgender Semantik:

$$(\textsc{Unfold}) \qquad \frac{(e, (\mathbf{rec}.\, e, \eta); \eta) \Downarrow cl}{(\mathbf{rec}.\, e, \eta) \Downarrow cl}$$

Die Übersetzungsfunktionen werden dann wie folgt erweitert:

$$\begin{aligned} \mathcal{C}[\![\mathbf{rec}\ .\ e]\!]\ell &= \textsc{Goto}\,\ell'; \textsc{Push}; \textsc{Closure}\,(\ell+1); \textsc{Cons}; \\ &\quad P; \textsc{Return}; \textsc{Closure}\,(\ell+1) \\ &\quad \text{mit } P = \mathcal{V}[\![e]\!](\ell+4), \ell' = \ell + |P| + 5 \\ \\ \mathcal{V}[\![\mathbf{rec}\ .\ e]\!]\ell &= \mathcal{C}[\![\mathbf{rec}\ .\ e]\!]\ell; \textsc{Eval} \end{aligned}$$

Diese Übersetzung ist weitaus effizienter als die naive Übersetzung des Fixpunktoperators. Sie wirkt zunächst wenig intuitiv, die Bedeutung erschließt sich bei näherer Betrachtung aber schnell.

Angenommen, wir befinden uns bereits in einem Rekursionsschritt, dann wurde der Code von P bereits einmal ausgeführt und wir kommen zum nächsten Rekursionsschritt. In diesem wird mit Push die aktuelle Umgebung auf den Stack gesichert und als nächstes mit Closure eine Closure mit aktueller Umgebung erzeugt. Durch Cons wird aus der ursprünglichen und der veränderten Umgebung eine neue Umgebung erzeugt und somit um den Eintrag für $(\mathbf{rec}\ .\ e, \eta)$ erweitert.

5 Zusammenfassung

Ausgehend von der Programmiersprache $\mathbf{L_{db}}$ haben wir eine abstrakte Maschine, die *LazyCAM*, definiert und gesehen, wie gültige Registerinhalte und Instruktionen aussehen und dass die Übergangsrelation, welche die Semantik der *LazyCAM* beschreibt, wohldefiniert und eindeutig ist. Ausgehend von der Festlegung der Semantik der *LazyCAM* konnten wir nun die drei Übersetzungsfunktionen

- $\mathcal{P}[\![e]\!]$, zur Erzeugung des Programmcodes zur Berechnung eines Ausdrucks $e \in$ *dbExp*,
- $\mathcal{C}[\![e]\!]$, zur Erzeugung von Code, welcher eine Closure für $e \in$ *dbExp* in einem Register ablegt, sowie
- $\mathcal{V}[\![e]\!]$, zur Erzeugung von Code zur Auswertung von $e \in$ *dbExp*

und haben uns dann von der Wohldefiniertheit dieser Übersetzungsfunktionen überzeugt.

Aufbauend darauf ist es uns dann gelungen, uns von der Korrektheit und damit davon zu überzeugen, dass die Codeerzeugung semantikerhaltend ist.

Zum Schluß haben wir noch gesehen, dass es durchaus noch möglich ist, die Übersetzung eines Ausdrucks $e \in$ *dbExp* zu verbessern, indem wir die Closurebildung für z. B. De Bruijn-Indeces auf Closure-Position minimieren. Eine andere Möglichkeit der Verbesserung ist die Einführung eines speziellen Ausdrucks für Rekursion in die Sprache $\mathbf{L_{db}}$ und einer entsprechenden Anpassung der Übersetzungsfunktionen $\mathcal{C}$ und $\mathcal{V}$.

Literaturverzeichnis

[Hin99] Hinze, Ralf: The Categorical Abstract Machine: Basics and Enhancements. (1999)

[Meu09] Meurer, Simon: *De Bruijn Indizes*. 2009. – Seminar Umgebungssemantiken

[Pie02] Pierce, Benjamin C.: *Types and Programming Languages*. Cambridge, Massachusetts, USA ; London, England : MIT Press, 2002. – ISBN 0–262–16209–1

[Reh09] Reh, Philipp: *Umgebungssemantik*. 2009. – Seminar Umgebungssemantiken

[WM92] Wilhelm, R. ; Maurer, D.: *Übersetzerbau*. Springer-Verlag, 1992