Universität Siegen
Institut für Compilerbau und Softwareanalyse
Fachbereich Informatik & Elektrotechnik

SS 2010

Diplomarbeit

Umgebungssemantiken:
Von der operationellen Semantik zum Compiler

Simon Meurer

Matr.Nr.: 731120

27. April 2010

Gutachter: Privatdozent Dr. Kurt Sieber
Prof. Dr. Wolfgang Merzenich

Inhaltsverzeichnis

Einleitung

Das Ziel dieser Diplomarbeit ist es eine operationelle Semantik Schritt für Schritt zu einem Compiler oder einem „Compiler-nahen“ Interpreter umzuformen. Dies soll am Beispiel einer einfachen funktionalen Sprache durchgeführt werden.

Diese Sprache ist eine *call-by-value* Version der Sprache L_2[Sie10] welche wiederum eine Art von PCF[Plo81] ist. So eine Sprache wird auch in den meisten Semantiklehrbüchern verwendet, wie zum Beispiel [Mit96].

Als operationelle Semantik verwenden wir eine Art von SOS[Plo81]. Diese ist eine big step Semantik wie in [Kah87] vorgestellt. Sie beruht auf der rein syntaktischen Umformung, insbesondere der Substitution. Daher wird sie bei uns auch Substitutionssemantik genannt.

Die einzelnen Umformungsschritte sollen semantik-erhaltend sein, was auch jeweils bewiesen wird.

SOS kann als Beschreibung eines rekursiven Interpreters aufgefasst werden. Dieser ist noch ineffizient, da er die Substitution verwendet. In ihm ist keinerlei Unterscheidung zwischen statischen und dynamischen Aspekten eines Programms sichtbar was dazu führt, dass „alles zur Laufzeit passiert“.

Erste Schritte in die Richtung eines effizienteren Interpreters wurden bereits in [AO09] durchgeführt. Zunächst wurde aus der Substitutionssemantik eine Umgebungssemantik entwickelt [AO09, Kap. 3]. Die Substitutionen werden dann nicht mehr explizit ausgeführt, sondern durch Einträge in sogenannte Umgebungen „verzögert“. Der Vorteil ist, dass nicht nur die Substitutionen eingespart werden, sondern es entstehen auch keine neuen Ausdrücke mehr (außer Konstanten), d.h. alle Ausdrücke sind Teilausdrücke des Gesamtprogramms. Diesen Vorteil werden wir später nutzen.

Nachteil dieser „naiven“ Umgebungssemantik ist, dass nur das Auffalten von **rec**-Ausdrücken (aus der Substitutionssemantik) simuliert wird. Daher findet bei jedem rekursiven Aufruf immer wieder der gleiche Auffaltungsschritt statt. Um dies zu vermeiden wurde in [AO09, Kap. 5] die Pointersemantik entwickelt. Hier wird dann ein Trick benutzt, der als „tying the knot“ bekannt ist, um eben diesen Nachteil auszumerzen.

Wir wollen diese Pointersemantik mit leichten Änderungen als Ausgangspunkt verwenden und nennen sie hier einfach Umgebungssemantik.

Die in [AO09, Kap. 4] vorgestellte namenlose Umgebungssemantik, die auf *de Bruijn-Indices* beruht, wird nicht übernommen, da wir diese durch eine für unsere Zwecke passendere ersetzen.

Die Umformungsschritte von der Umgebungssemantik zum Compiler sind im einzelnen:

(1.) Einführung von expliziten Syntaxbäumen

Da nur Teilausdrücke des Gesamtprogramms vorkommen können, lässt sich jeder Ausdruck als Knoten im Syntaxbaum des Gesamtprogramms darstellen, durch die Wurzel des Syntaxbaums für ebendiesen Ausdruck. Aus den Knoten lässt sich nicht nur der Ausdruck, sondern auch der „Kontext“ im Gesamtprogramm ablesen. Unter diesem Kontext verstehen wir jede statische Information, die hilfreich sein könnte um die Auswertung des Ausdrucks effizienter zu machen.

(2.) Namenlose Umgebungen

Ein Beispiel für diesen Kontext ist die Liste von Namen, die bei jeder Auswertung des Ausdrucks „bekannt“ sind, d.h. in der (Laufzeit-) Umgebung enthalten sind. Diese findet man ausgehend vom Knoten auf dem Weg zur Wurzel des Programms (sogar in der „richtigen“ Reihenfolge).

Das ganze führt zu den namenlosen Umgebungen. In diesen sind im Unterschied zu [AO09, Kap. 4] keine Einträge für Funktionen, die durch **let rec** $id = v$ **in** e eingeführt wurden.[1]

(3.) Umwandlung von Rekursion in Iteration

Die bisherigen Semantiken sind allesamt big step Semantiken (natural semantics [Kah87]), d.h. rekursive Interpreter. Um näher zum Compiler zu gelangen, braucht man einen iterativen Interpreter (small step Semantik).

Jede Rekursion lässt sich in eine Iteration umwandelt, indem man einen Stack für Argumenten und Resultate benutzt. Diese systematische Umwandlung führt aber dazu, dass man Argumente mehrfach einträgt und Resultate zu lange aufbewahrt. Deshalb werden wir einen ad hoc Ansatz für eine bessere Verwaltung von Argumenten und Resultaten vorstellen.

Als Ergebnis erhalten wir eine small step Semantik, die mit einem Stack von (namenlosen) Umgebungen arbeitet (vgl. SECD-Maschine[Lan64]).

(4.) Effiziente Stacks

Die Umgebungen in (3.) unterscheiden sich nur geringfügig voneinander. Jede neue Umgebung entsteht aus einer früheren durch Hinzufügen eines einzigen Eintrags. Sie lässt sich also auch durch den neuen Eintrag und einen Zeiger auf die alte Umgebung darstellen (den sogenannten Accesslink [ASU99]). Dies führt zu einem „realistischeren“ Stack, der keine ganzen Umgebungen, sondern nur noch einzelne Einträge (sogenannte „stack frames“ [Mit02]) enthält.

Das Entfernen von Einträgen am Ende eines Blocks (gemäß der „Stack Disziplin“) ist aber nur möglich, wenn die Programmiersprache so begrenzt wird, dass Funktionen nicht als Resultate aus einem inneren Block entstehen können. Daher schränken wir die Sprache hier ein.

[1] Dies entspricht den üblichen Compilern, wie in [WM92] oder [ASU99]

(5.) Compiler

Die small step Semantik aus (4.) arbeitet bereits mit der passenden „Datenstruktur“, die auch ein compiliertes Programm benutzt. Aber sie benutzt noch den Syntaxbaum (indem sie auf seine Knoten zugreift), der dem compilierten Programm nicht mehr zur Verfügung steht.

Deshalb muss jeder small step durch eine Folge von Maschinenbefehlen simuliert werden, wobei die Knoten des Syntaxbaumes durch Programmspeicheradressen ersetzt werden.

Hier beschränken wir uns darauf das Prinzip der Übersetzung zu beschreiben und verzichten deshalb bewusst auf einige der technischen Details und Beweise.

Diese Auflistung verteilt sich dann wie folgt auf die Kapitel: Im ersten Kapitel wird die Umgebungssemantik, wie in [AO09, Kap. 5] entwickelt, vorgestellt. Die Syntaxbäume werden im zweiten Kapitel eingeführt und in die Umgebungssemantik eingebaut. Während das dritte Kapitel die Umwandlung der Umgebungssemantik mit Syntaxbäumen in eine small step Semantik behandelt, werden im vierten Kapitel die namenlosen Umgebungen eingeführt, sowie die namenlose small step Semantik vorgestellt. Der realistischere Stack wird dann im fünften Kapitel in die small step Semantik eingeführt. Im sechsten Kapitel wird abschließend der Compiler definiert.

1 Umgebungssemantik

In diesem Kapitel definieren wir die Syntax und die Semantik der Programmiersprache, von der wir ausgehen. Beide sind (bis auf kleine, unwesentliche Änderungen) aus [AO09, Kap. 5] übernommen.

Definition 1.1 (Syntax der Programmiersprache) *Vorgegeben seien*

- *die Menge* $Bool = \{true, false\}$ *der booleschen Konstanten* b*,*
- *die Menge* $Int = \mathbb{Z}$ *der Integerkonstanten* z*, und*
- *eine (unendliche) Menge* Id *von Namen* id*.*

Die Mengen Op *aller Operationen* op*,* $Const$ *aller* Konstanten c*,* V *aller* syntaktischen Werte v *und* Exp *aller* Ausdrücke e *sind wie folgt definiert:*

op	$::=$	$+ \mid - \mid * \mid \leq \mid \geq \mid < \mid > \mid =$	*(Operatoren)*
c	$::=$	b	*(Boolescher Wert)*
	$\mid$	z	*(ganze Zahl)*
	$\mid$	op	*(Operator)*
	$\mid$	$\#_i$	*(Projektion mit* $i \in \mathbb{N}$*)*
v	$::=$	c	*(Konstante)*
	$\mid$	$\lambda id.\, e$	*(λ-Ausdruck)*
	$\mid$	$(v_1, \ldots, v_n)$	*(Tupel mit* $n > 1$*)*
e	$::=$	c	*(Konstante)*
	$\mid$	id	*(Identifier)*
	$\mid$	$e_1\, e_2$	*(Applikation)*
	$\mid$	$(e_1, \ldots, e_n)$	*(Tupel mit* $n > 1$*)*
	$\mid$	**if** e_0 **then** e_1 **else** e_2	*(bedingter Ausdruck)*
	$\mid$	$\lambda id.\, e$	*(λ-Abstraktion)*
	$\mid$	**let** $id = e_1$ **in** e_2	*(***let***-Ausdruck)*
	$\mid$	**let rec** $id = v$ **in** e	*(***let rec***-Ausdruck)*

Syntaktische Werte v stammen aus der Substitutionssemantik [AO09, Kap. 2], wo sie als „fertig ausgewertete Ausdrücke“ dienten. Hier benutzen wir sie nur noch als Nichtterminalzeichen, um die rechte Seite einer rekursiven Deklaration einzuschränken.

Man beachte, dass wir Tupel von λ-Abstraktionen zu den syntaktischen Werten zählen. Damit lassen sich simultan rekursive Funktionen definieren, wie in folgendem Beispiel an den Funktionen *even* und *odd* zu sehen ist.[1]

[1] Natürlich lassen sich solche Beispiele schöner formulieren, wenn man passenden syntaktischen Zucker einführt (vgl. [Sie10]). Darauf wird in dieser Arbeit verzichtet.

Beispiel 1.1 $\textbf{let rec}\ evenodd =$
$(\lambda x.\,\textbf{if}\ =(x,0)\ \textbf{then}\ true\ \textbf{else}\ (\#_2\ evenodd)\,(-(x,1)),$
$\lambda x.\,\textbf{if}\ =(x,0)\ \textbf{then}\ false\ \textbf{else}\ (\#_1\ evenodd)\,(-(x,1)))$
$\textbf{in}\ (\#_1\ evenodd)\,1$

Definition 1.2 (Umgebungen, Closures und Werte) *Die Mengen Env aller* Umgebungen η*, Cl aller Closures cl und W aller* Werte w *der Umgebungssemantik sind durch die folgende kontextfreie Grammatik definiert:*

$$
\begin{array}{rcll}
\eta & ::= & [\,] & \\
 & | & id : (v, \circlearrowleft); \eta & \textit{(Eintrag für Rekursion)} \\
 & | & id : w; \eta & \textit{(normaler Eintrag)} \\
cl & ::= & (e, \eta) & \\
w & ::= & c & \textit{(Konstante)} \\
 & | & (\lambda id.\, e, \eta) & \textit{(Closure)} \\
 & | & (w_1, \ldots, w_n) & \textit{(Tupel mit } n > 1\textit{)}
\end{array}
$$

Definition 1.3 (Lookup & Expand)
Die totale Funktion $expand : Env \times Val \to W$ *ist wie folgt definiert:*

$$
expand(\eta, v) \quad = \quad \begin{cases} c & falls\, v = c \\ (\lambda id.\, e, \eta) & falls\, v = \lambda id.\, e \\ (expand(\eta, v_1), \ldots, expand(\eta, v_n)) & falls\, v = (v_1, \ldots, v_n) \end{cases}
$$

Die partielle Funktion $lookup : Env \times Id \hookrightarrow W$ *ist wie folgt definiert:*

$$
lookup(\eta, id) \quad = \quad \begin{cases} w & falls\, \eta = id : w; \eta' \\ expand(\eta, v) & falls\, \eta = id : (v, \circlearrowleft); \eta' \\ lookup(\eta', id) & falls\, \eta = id' : \_\_; \eta' \, und \, id \neq id' \\ undefiniert & falls\, \eta = [\,] \end{cases}
$$

Die Funktion op^I bezeichnet man als Interpretation des Operators op.

__ kennzeichnet eine Wildcard (vgl. O'Caml), die in diesem Fall entweder w oder $(v, \circlearrowleft)$, sein kann.

Definition 1.4 (Big step Regeln) *Ein* big step *in der Umgebungssemantik ist eine Formel der Gestalt* $(e, \eta) \Downarrow w$*, wobei* $e \in Exp$*,* $\eta \in Env$ *und* $w \in W$*. Ein derartiger big step heißt* gültig*, wenn er sich mit den in Abbildung 1.1 dargestellten Regeln herleiten lässt.*

Beispiel 1.2 Um die Bedeutung der Funktionen *lookup* und *expand* zu illustrieren, skizieren wir in Abbildung 1.2 die Semantik der simultan rekursiv definierten Funktionen aus Beispiel 1.1.

Hier wird im 5. Schritt über *lookup* die Funktion *expand* aufgerufen, welche die Umgebung auf die einzelnen Elemente des Tupels verteilt.

$$\text{(Const)}\quad (c,\eta) \Downarrow c$$

$$\text{(Closure)}\quad (\lambda id.\, e,\eta) \Downarrow (\lambda id.\, e,\eta)$$

$$\text{(Id)}\quad \frac{lookup(\eta, id) = w}{(id,\eta) \Downarrow w}$$

$$\text{(Op)}\quad \frac{(e_1,\eta) \Downarrow op \qquad (e_2,\eta) \Downarrow (z_1,z_2)}{(e_1\, e_2,\eta) \Downarrow op^I(z_1,z_2)}$$

$$\text{(Beta-V)}\quad \frac{(e_1,\eta) \Downarrow (\lambda id'.\, e',\eta') \qquad (e_2,\eta) \Downarrow w' \qquad (e', id' : w';\eta') \Downarrow w}{(e_1\, e_2,\eta) \Downarrow w}$$

$$\text{(Proj)}\quad \frac{(e_1,\eta) \Downarrow \#_i \qquad (e_2,\eta) \Downarrow (w_1,\ldots,w_n) \qquad 1 \le i \le n}{(e_1\, e_2,\eta) \Downarrow w_i}$$

$$\text{(Tuple)}\quad \frac{(e_1,\eta) \Downarrow w_1 \qquad \ldots \qquad (e_n,\eta) \Downarrow w_n}{((e_1,\ldots,e_n),\eta) \Downarrow (w_1,\ldots,w_n)}$$

$$\text{(Let)}\quad \frac{(e_1,\eta) \Downarrow w \qquad (e_2, id : w;\eta) \Downarrow w'}{(\textbf{let}\ id = e_1\ \textbf{in}\ e_2,\eta) \Downarrow w'}$$

$$\text{(Let-Rec)}\quad \frac{(e, id : (v, \circlearrowleft);\eta) \Downarrow w}{(\textbf{let rec}\ id = v\ \textbf{in}\ e,\eta) \Downarrow w}$$

$$\text{(Cond-True)}\quad \frac{(e_0,\eta) \Downarrow true \qquad (e_1,\eta) \Downarrow w}{(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2,\eta) \Downarrow w}$$

$$\text{(Cond-False)}\quad \frac{(e_0,\eta) \Downarrow false \qquad (e_2,\eta) \Downarrow w}{(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2,\eta) \Downarrow w}$$

Abbildung 1.1: Big step Regeln für die Umgebungssemantik

(1) $(\textbf{let rec}\ evenodd =$
$\quad (\lambda x.\, \textbf{if}\ =(x,0)\ \textbf{then}\ true\ \textbf{else}\ (\#_2\ evenodd)\,(-(x,1)),$
$\quad \lambda x.\, \textbf{if}\ =(x,0)\ \textbf{then}\ false\ \textbf{else}\ (\#_1\ evenodd)\,(-(x,1)))$
$\textbf{in}\ (\#_1\ evenodd)\,1,[\,]) \Downarrow\ false$
(Let-Rec)

(2) $((\#_1\ evenodd)\,1, evenodd : \ldots;[\,]) \Downarrow\ false$
(Beta-V)

- (3) $(\#_1\ evenodd, evenodd : \ldots;[\,]) \Downarrow$
  $(\lambda x.\, \textbf{if}\ =(x,0)\ \textbf{then}\ false\ \textbf{else}\ (\#_1\ evenodd)\,(-(x,1)), evenodd : \ldots;[\,])$
  (Proj)
  - (4) $(\#_1\ , evenodd : \ldots;[\,]) \Downarrow\ \#_1$
    (Const)
  - (5) $(evenodd, evenodd : \ldots;[\,]) \Downarrow$
    $((\lambda x.\, \textbf{if}\ =(x,0)\ \textbf{then}\ true\ \textbf{else}$
    $\quad (\#_2\ evenodd)\,(-(x,1)), evenodd : \ldots;[\,]),$
    $(\lambda x.\, \textbf{if}\ =(x,0)\ \textbf{then}\ false\ \textbf{else}$
    $\quad (\#_1\ evenodd)\,(-(x,1)), evenodd : \ldots;[\,]))$
    (Id)
- (6) ...
- (7) ...

Abbildung 1.2: Semantik von *even* und *odd*

2 Explizite Syntaxbäume

Die Umgebungssemantik besitzt einen entscheidenden Vorteil gegenüber der Substitutionssemantik: Sie vermeidet nicht nur die zeitaufwendigen Substitutionen, sondern sie kommt auch ganz ohne Umformung von Ausdrücken aus. Mit anderen Worten:

Bei der Herleitung eines big steps $(e, [\,]) \Downarrow w$ für ein „Gesamtprogramm“ e treten – abgesehen von neuen Konstanten, die durch die Regel (Op) entstehen – *nur* Teilausdrücke von e auf.

Das bedeutet, dass ein Interpreter, der auf der Umgebungssemantik beruht, gar nicht mehr mit Ausdrücken arbeiten muss. Anstelle von Ausdrücken kann er (Zeiger auf) Knoten im Syntaxbaum des Gesamtprogramms e benutzen.

In diesem Sinne wollen wir jetzt auch unsere Umgebungssemantik verändern. Alle Ausdrücke sollen durch entsprechende Knoten im Syntaxbaum des Gesamtprogramms ersetzt werden. Dazu benötigen wir zunächst eine formale Definition von Knoten und (Syntax-) Bäumen.

2.1 (Syntax-) Bäume

Im Allgemeinen werden Bäume häufig als eingeschränkte Graphen definiert. So auch in [Dör77, S.400]. Dort wird ein Baum als ein zusammenhängender Graph ohne Zykel beschrieben.

Eine andere Art Bäume zu definieren findet sich in [Knu68, S.305] und [Mer97, S.76]. Hier werden Bäume über Mengen definiert. Ein Baum ist eine endliche Menge T von Knoten. Einer dieser Knoten ist die Wurzel des Baumes. Die übrigen Knoten werden in $m \geq 0$ disjunkte Mengen $T_1, \ldots, T_m$ aufgeteilt, wobei jede dieser Mengen wiederum ein Baum ist.

Da wir in der Umgebungssemantik eine Bezeichnung für jeden Knoten eines Baumes benötigen, bevorzugen wir aber eine andere Definition, nämlich die aus [LMW86].

Definition 2.1 (Knoten, Baumbereiche und Bäume)

(a) *Die Menge Node aller Knoten κ ist definiert durch:*

$$\begin{array}{rcll} \kappa & ::= & \epsilon & \textit{(Wurzel)} \\ & | & \kappa.i & \textit{(i-tes Kind von κ ($i \in \mathbb{N}, i > 0$))} \end{array}$$

(b) *Ein* Baumbereich *ist eine Menge $K \subseteq Node$, für die gilt:*

Wenn $\kappa.i \in K$, dann ist auch $\kappa \in K$ und $\kappa.j$ für $j = 1, \ldots, i-1$.

(c) *Sei L eine Menge. Ein (L-markierter)* Baum *ist ein Paar $T = (K, \ell)$ mit:*

- *K ist ein Baumbereich*
- *$\ell : K \to L$*

$\ell(\kappa)$ heißt Markierung *des Knotens κ.*

Für die abstrakten Syntaxbäume unserer Programmiersprache wählen wir die Menge L der Markierungen wie folgt:

$$\begin{aligned} L \;=\; & \mathit{Const} \cup \mathit{Id} \cup \{\mathit{App}, \mathit{Cond}\} \\ & \cup \{\mathit{Tuple}\, n \mid n > 1\} \\ & \cup \{\lambda\, \mathit{id}, \mathbf{let}\, \mathit{id}, \mathbf{let\ rec}\, \mathit{id} \mid \mathit{id} \in \mathit{Id}\} \end{aligned}$$

Beispiel 2.1 Der abstrakte Syntaxbaum für den Ausdruck $((\lambda x.\, \lambda y.\, +\, (x, y))\, 1)\, 2$ sollte (intuitiv) so aussehen, wie in Abbildung 2.1.

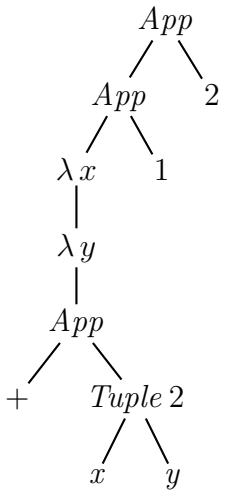

Abbildung 2.1: Syntaxbaum für $((\lambda x.\, \lambda y.\, +\, (x, y))\, 1)\, 2$

Formal lässt sich der Syntaxbaum für einen Ausdruck e wie folgt definieren:

Definition 2.2 (Abstrakter Syntaxbaum) *Sei $e \in \mathit{Exp}$.*

(a) *Der zu e gehörige* Baumbereich K_e *und der zu $\kappa \in K$ gehörige* Teilausdruck $e[\kappa]$ *seien induktiv definiert durch:*

(i) $\epsilon \in K_e$ *mit* $e[\epsilon] = e$

(ii) Wenn $\kappa \in K_e$ mit $e[\kappa] = \lambda id.\, e_1$, dann ist auch $(\kappa.1) \in K$ mit $e[\kappa.1] = e_1$.

(iii) Wenn $\kappa \in K_e$ mit $e[\kappa] = e_1\, e_2$ oder $e[\kappa] = \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2$ oder $e[\kappa] = \mathbf{let\,rec}\ id = e_1\ \mathbf{in}\ e_2$, dann ist $(\kappa.i) \in K$ mit $e[\kappa.i] = e_i$ für $i = 1, 2$.

(iv) *Wenn* $\kappa \in K_e$ *mit* $e[\kappa] = \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$, *dann ist für* $i = 1, 2, 3$, $(\kappa.i) \in K$ *mit* $e[\kappa.i] = e_{i-1}$.

(v) *Wenn* $\kappa \in K_e$ *mit* $e[\kappa] = (e_1, \ldots, e_n)$, *dann ist für* $i = 1, \ldots, n$, $(\kappa.i) \in K$ *mit* $e[\kappa.i] = e_i$.

(b) *Der (abstrakte)* Syntaxbaum *für* e *ist* $T_e = (K, \ell)$, *wobei* $K = K_e$ *der Baumbereich zu* e *ist und* $\ell : K \to L$ *definiert ist durch:*

$$
\begin{array}{lll}
\ell(\kappa) = c & \textit{falls} & e[\kappa] = c \\
\ell(\kappa) = id & \textit{falls} & e[\kappa] = id \\
\ell(\kappa) = App & \textit{falls} & e[\kappa] = e_1\, e_2 \\
\ell(\kappa) = Cond & \textit{falls} & e[\kappa] = \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \\
\ell(\kappa) = Tuple\, n & \textit{falls} & e[\kappa] = (e_1, \ldots, e_n) \\
\ell(\kappa) = \lambda\, id & \textit{falls} & e[\kappa] = \lambda id.\, e \\
\ell(\kappa) = \textbf{let } id & \textit{falls} & e[\kappa] = \textbf{let } id = e_1 \textbf{ in } e_2 \\
\ell(\kappa) = \textbf{let rec } id & \textit{falls} & e[\kappa] = \textbf{let rec } id = e_1 \textbf{ in } e_2
\end{array}
$$

2.2 Umgebungssemantik mit expliziten Syntaxbäumen

Definition 2.3 (Neue Umgebungen, Closures und Werte) *Umgebungen* η, *Closures* cl *und Werte* w *werden neu definiert durch:*

$$
\begin{array}{lcll}
\eta & ::= & [\,] & \\
 & | & id : w; \eta & \textit{(normaler Eintrag)} \\
 & | & id : (\kappa, \circlearrowleft); \eta & \textit{(Eintrag für Rekursion)} \\
cl & ::= & (\kappa, \eta) & \\
w & ::= & c & \textit{(Konstante)} \\
 & | & (w_1, \ldots, w_n) & \textit{(Tupel mit } n \geq 2\textit{)} \\
 & | & (\kappa, \eta) & \textit{(Closure mit } \ell(\kappa) = \lambda id.\, e\textit{)}
\end{array}
$$

Die Mengen der neuen Umgebungen, Closures und Werte bezeichnen wir mit Env_{node}, Cl_{node} und W_{node}.

Definition 2.4 (Neue Definition von Lookup & Expand)
Die partielle Funktion $expand : Env_{node} \times Node \hookrightarrow W_{node}$ *ist wie folgt definiert:*

$$
expand(\eta, \kappa) \;=\; \begin{cases}
c & \textit{falls } \ell(\kappa) = c \in Const \\
(\kappa, \eta) & \textit{falls } \ell(\kappa) = \lambda\, id \\
(expand(\eta, \kappa.1), \ldots, expand(\eta, \kappa.n)) & \textit{falls } \ell(\kappa) = Tuple\, n \\
undefiniert & sonst
\end{cases}
$$

Die partielle Funktion lookup : $Env_{node} \times Id \hookrightarrow W_{node}$ *ist wie folgt definiert:*

$$lookup(\eta, id) \quad = \quad \begin{cases} w & falls\, \eta = id : w; \eta' \\ expand(\eta, \kappa) & falls\, \eta = id : (\kappa, \circlearrowleft); \eta' \\ lookup(\eta', id) & falls\, \eta = id' : \_; \eta' \, und \, id \neq id' \\ undefiniert & falls\, \eta = [\,] \end{cases}$$

Zuletzt müssen noch die big step Regeln angepasst werden. Hier müssen wir zunächst auch wieder einfach die Ausdrücke durch die entsprechenden Knoten ersetzen. Aber dann muss man bei jeder Regel die Markierung des aktuellen Knotens betrachten, um wieder zum Inhalt des Knotens zurück zu kommen.

$$\text{(Const)}\ \frac{\ell(\kappa) = c}{(\kappa, \eta) \Downarrow c} \qquad \text{(Closure)}\ \frac{\ell(\kappa) = \lambda\, id}{(\kappa, \eta) \Downarrow (\kappa, \eta)} \qquad \text{(Id)}\ \frac{\ell(\kappa) = id \qquad lookup(\eta, id) = w}{(\kappa, \eta) \Downarrow w}$$

$$\text{(Op)}\ \frac{\ell(\kappa) = App \qquad ((\kappa.1), \eta) \Downarrow op \qquad ((\kappa.2), \eta) \Downarrow (z_1, z_2)}{(\kappa, \eta) \Downarrow op^I(z_1, z_2)}$$

$$\text{(Beta-V)}\ \frac{((\kappa.1), \eta) \Downarrow (\kappa', \eta') \qquad \ell(\kappa') = \lambda\, id' \qquad ((\kappa.2), \eta) \Downarrow w' \qquad ((\kappa'.1), id' : w'; \eta') \Downarrow w}{(\kappa, \eta) \Downarrow w}$$

$$\text{(Proj)}\ \frac{\ell(\kappa) = App \qquad ((\kappa.1), \eta) \Downarrow \#_i \qquad ((\kappa.2), \eta) \Downarrow (w_1, \ldots, w_n) \qquad 1 \leq i \leq n}{(\kappa, \eta) \Downarrow w_i}$$

$$\text{(Tuple)}\ \frac{\ell(\kappa) = Tuple\, n \qquad ((\kappa.i), \eta) \Downarrow w_i \qquad 1 \leq i \leq n}{(\kappa, \eta) \Downarrow (w_1, \ldots, w_n)}$$

$$\text{(Let)}\ \frac{\ell(\kappa) = \textbf{let}\ id \qquad ((\kappa.1), \eta) \Downarrow w' \qquad ((\kappa.2), id : w'; \eta) \Downarrow w}{(\kappa, \eta) \Downarrow w}$$

$$\text{(Let-Rec)}\ \frac{\ell(\kappa) = \textbf{let rec}\ id \qquad ((\kappa.2), id : ((\kappa.1), \circlearrowleft); \eta) \Downarrow w}{(\kappa, \eta) \Downarrow w}$$

$$\text{(Cond-True)}\ \frac{\ell(\kappa) = Cond \qquad ((\kappa.1), \eta) \Downarrow true \qquad ((\kappa.2), \eta) \Downarrow w}{(\kappa, \eta) \Downarrow w}$$

$$\text{(Cond-False)}\ \frac{\ell(\kappa) = Cond \qquad ((\kappa.1), \eta) \Downarrow false \qquad ((\kappa.3), \eta) \Downarrow w}{(\kappa, \eta) \Downarrow w}$$

Abbildung 2.2: Big step Regeln für die Umgebungssemantik mit Syntaxbäumen

Definition 2.5 (Neue big step Regeln) *Ein big step ist jetzt eine Formel der Gestalt* $(\kappa, \eta) \Downarrow w$*, mit* $\kappa \in Node$ *sowie* $\eta \in Env_{node}$ *und* $w \in W_{node}$*. Ein derartiger big step heißt* gültig *für das Gesamtprogramm* e_0 *mit Syntaxbaum* $T_{e_0} = (K, \ell)$*, wenn er sich mit denen in Abbildung 2.2 beschriebenen Regeln herleiten lässt.*

2.3 Äquivalenz der beiden Umgebungssemantiken

Es soll jetzt noch präzise formuliert werden, in welchem Sinne die alte und neue Umgebungssemantik zueinander äquivalent sind.

Als erstes benötigen wir einen Zusammenhang zwischen den verschiedenen Umgebungen und damit einhergehend einen zwischen den verschiedenen Werten. Dazu wird die Relation $\equiv$ eingeführt. Diese muss sich allerdings immer auf einen Gesamtausdruck beziehen, da wir in der big step Semantik die Knoten des Syntaxbaumes betrachten müssen.

Definition 2.6 *Für jedes (Gesamt-) Programm $e \in Exp$ sei die Relation $\equiv_e \subseteq (Env \times Env_{node}) \cup (W \times W_{node})$ induktiv definiert durch die Regeln in Abbildung 2.3.*

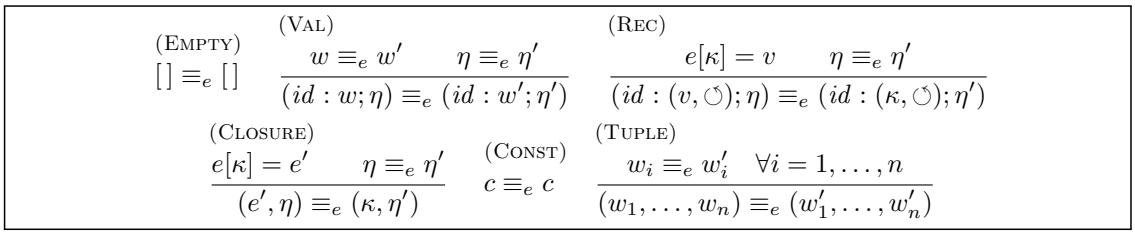

Abbildung 2.3: Definition der Relation $\equiv_e$

Dann lässt sich die Äquivalenz der beiden Semantiken so formulieren:

Satz 2.1 (Äquivalenz der beiden Umgebungssemantiken) *Sei $e \in Exp$ mit dem Syntaxbaum $T_e = (K, \ell)$, $\kappa \in K$ und $\eta \equiv_e \eta'$. Dann gilt:*

(a) *Wenn $(e[\kappa], \eta) \Downarrow w$, dann existiert ein w' mit $w \equiv_e w'$, so dass $(\kappa, \eta') \Downarrow w'$ gültig für e.*

(b) *Wenn $(\kappa, \eta') \Downarrow w'$ gültig für e, dann existiert ein w mit $w \equiv_e w'$, so dass $(e[\kappa], \eta) \Downarrow w$.*

Zum Beweis des Satzes benötigt man noch folgendes Lemma.

Lemma 2.1 *Seien e, κ, η und η' wie in Satz 2.1. Dann gilt:*

(a) *Wenn $e[\kappa] \in Val$, dann ist $expand(\eta, e[\kappa]) \equiv_e expand(\eta, \kappa)$*

(b) *Wenn $dom(\eta) = dom(\eta')$, dann ist $lookup(\eta, id) \equiv_e lookup(\eta', id)$ für alle $id \in dom(\eta)$*

Die Beweise für Lemma und Satz sind trivial, da in allen Definitionen und Regeln lediglich Ausdrücke durch die entsprechenden Knoten ersetzt wurden.

Als Spezialfall von Satz 2.1 erhält man für das Gesamtprogramm e:

Korollar 2.1 $(e, [\,]) \Downarrow c \Leftrightarrow (\epsilon, [\,]) \Downarrow c$

3 Von der Rekursion zur Iteration

Die big step Umgebungssemantik aus Kapitel 2 kann als rekursive Definition eines Interpreters aufgefasst werden, d.h. als rekursive Implementierung der partiellen Funktion $eval : Node \times Env_{node} \hookrightarrow W_{node}$

$$eval(\kappa, \eta) \quad = \quad \begin{cases} w & \text{falls } (\kappa, \eta) \Downarrow w \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Um näher an einen Compiler zu kommen, wollen wir die big step Semantik in eine small step Semantik, d.h. eine rekursive Implementierung von *eval* in eine iterative umwandeln. Eine solche Umwandlung lässt sich für jede rekursive Funktion durchführen, indem man einen Stack benutzt, auf dem die Argumente und (Zwischen-) Resultate der rekursiven Aufrufe verwaltet werden. Um das Prinzip dieser Stack-Verwaltung zu erkennen, betrachten wir anstelle von *eval* zunächst ein einfaches Beispiel einer rekursiven Funktion, nämlich die Fibonacci-Funktion. Diese ist definiert durch die beiden Anfangswerte $\mathbf{fib}(0) = 1$, $\mathbf{fib}(1) = 1$ und den rekursiven Aufruf $\mathbf{fib}(n+2) = \mathbf{fib}(n+1) + \mathbf{fib}(n)$.

fib lässt sich (wie jede Funktion) als Relation $\Downarrow \subseteq \mathbb{N} \times \mathbb{N}$ auffassen und mit Hilfe von Regeln definieren (wie man es aus der logischen Programmierung kennt vgl. Prolog), nämlich durch die Regeln in Abbildung 3.1.

$$\frac{}{0 \Downarrow 1}\ (\textsc{Fib-0}) \qquad \frac{}{1 \Downarrow 1}\ (\textsc{Fib-1}) \qquad \frac{n-1 \Downarrow n_1 \qquad n-2 \Downarrow n_2}{n \Downarrow n_1 + n_2}\ (\textsc{Fib-n})$$

Abbildung 3.1: Big step Regeln für die Fibonacci-Funktion

Damit entspricht jeder Aufrufbaum für fib einem Beweisbaum, z.B. ergibt sich für $\mathbf{fib}(3)$ der Beweisbaum in Abbildung 3.2.

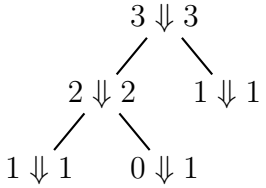

Abbildung 3.2: Big step-Beweisbaum von $\mathbf{fib}(3)$

Am Beweisbaum kann man (aufgrund der Relationsschreibweise) sehr schön die zeitliche Reihenfolge erkennen, in der die Argumente und Resultate entstehen: Man läuft einfach „am Baum entlang“ wie es durch die Pfeile in Abbildung 3.3 angedeutet ist.

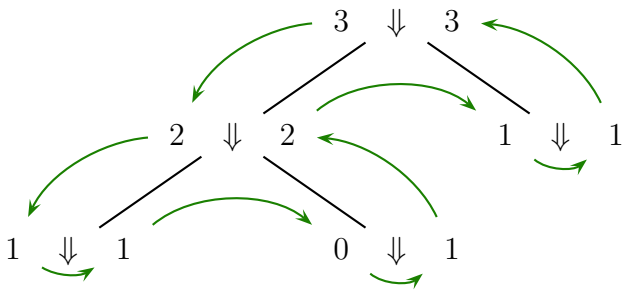

Abbildung 3.3: Reihenfolge im big step-Beweisbaum **fib**(3)

Diese zeitliche Reihenfolge wird man auch bei der iterativen Implementierung beibehalten, d.h. man kann die Pfeile, die am big step Beweisbaum entlang führen, als Folge von small steps auffassen. Es bleibt aber noch die Frage zu klären, wie man den Stack verwaltet, auf dem diese small steps arbeiten.

3.1 Ansätze zur Stack-Verwaltung

Der einfachste Ansatz zur Stack-Verwaltung (der für jede Funktion funktioniert) ist der folgende, den wir als *naive Stackverwaltung* bezeichnen:

- Wenn man einen Schritt nach unten oder zu einem benachbarten Knoten macht, wird ein neues Argument a' berechnet. Es kann vom letzten Argument a und den darüber liegenden Zwischenresultaten abhängen. Die Anzahl dieser Zwischenresultate ist 0 beim Schritt nach unten und > 0 beim Schritt zum Nachbarn. Das neue Argument a' wird auf dem Stack abgelegt und es wird nichts vom Stack entfernt.

- Wenn man einen Schritt nach oben oder innerhalb eines Knotens macht, wird ein Resultat b berechnet. Es kann ebenfalls vom letzten Argument a und den darüber liegenden Zwischenresultaten abhängen. Die Anzahl dieser Zwischenresultate ist hier 0 beim Schritt innerhalb des Knotens und > 0 beim Schritt nach oben. Auch das Resultat b wird oben auf den Stack gelegt, aber zuvor wird a zusammen mit den darüberliegenden Zwischenresultaten entfernt.

Beispiel 3.1 Für unser Beispiel mit **fib**(3) würde sich der Stack, auf dem Argumente mit A und Resultate mit R markiert werden, dann wie folgt entwickeln:
$\mathrm{A}\,3; \varepsilon \to \mathrm{A}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{A}\,1; \mathrm{A}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{R}\,1; \mathrm{A}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{A}\,0; \mathrm{R}\,1; \mathrm{A}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{R}\,1; \mathrm{R}\,1; \mathrm{A}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{R}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{A}\,1; \mathrm{R}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{R}\,1; \mathrm{R}\,2; \mathrm{A}\,3; \varepsilon \to \mathrm{R}\,3; \varepsilon$

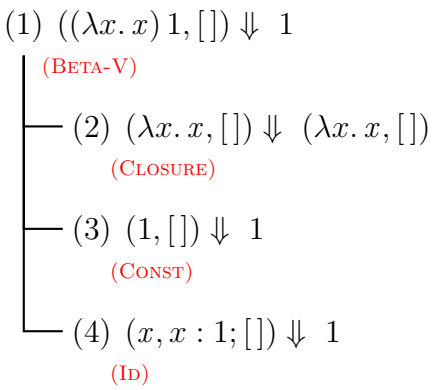

Abbildung 3.4: Big step-Beweisbaum von $((\lambda x.\, x)\, 1, [\,])$

Die naive Stackverwaltung lässt sich - wie gesagt - für jede rekursive Funktion durchführen, also auch für die Funktion *eval* aus der big step Semantik.

Beispiel 3.2 Für den Aufruf[1] $eval((\lambda x.\, x)\, 1, [\,])$, dessen Beweisbaum in Abbildung 3.4 dargestellt ist, entwickelt sich der Stack dann wie folgt:

$\mathrm{A}\,((\lambda x.\, x)\, 1, [\,]); \varepsilon$
$\to \mathrm{A}\,(\lambda x.\, x, [\,]); \mathrm{A}\,((\lambda x.\, x)\, 1, [\,]); \varepsilon$
$\to \mathrm{R}\,(\lambda x.\, x, [\,]); \mathrm{A}\,((\lambda x.\, x)\, 1, [\,]); \varepsilon$
$\to \mathrm{A}\,(1, [\,]); \mathrm{R}\,(\lambda x.\, x, [\,]); \mathrm{A}\,((\lambda x.\, x)\, 1, [\,]); \varepsilon$
$\to \mathrm{R}\,1; \mathrm{R}\,(\lambda x.\, x, [\,]); \mathrm{A}\,((\lambda x.\, x)\, 1, [\,]); \varepsilon$
$\to \mathrm{A}\,(x, x : 1; [\,]); \mathrm{R}\,1; \mathrm{R}\,(\lambda x.\, x, [\,]); \mathrm{A}\,((\lambda x.\, x)\, 1, [\,]); \varepsilon$
$\to \mathrm{R}\,1; \mathrm{R}\,1; \mathrm{R}\,(\lambda x.\, x, [\,]); \mathrm{A}\,((\lambda x.\, x)\, 1, [\,]); \varepsilon$
$\to \mathrm{R}\,1; \varepsilon$

Schon an diesem kleinen Beispiel kann man erkennen, dass die naive Stack-Verwaltung für die Funktion *eval* einige Nachteile mit sich bringt:

- Resultate werden unnötig lange aufgehoben, z.B. könnte man das Resultat der 1. und 2. Prämisse von (Beta-V) entfernen, sobald man aus ihnen das Argument für die 3. Prämisse errechnet hat.

- Aufeinanderfolgende Argumente (κ, η) und (κ', η') sind sich oft sehr ähnlich. Meistens gilt:

  (1) $\kappa' = \kappa.i$ und
  (2) $\eta' = \eta$

  Einzige Ausnahme von (1) ist das Argument der 3. Prämisse von (Beta-V), bei der ein „Sprung“ zum Rumpf der λ-Abstraktion stattfindet.

[1] Aus Gründen der Lesbarkeit werden hier die Ausdrücke anstelle der Knoten verwendet

Die einzige Ausnahme von (2) sind die Prämissen, bei denen η um einen Eintrag erweitert wird, also die 3. Prämisse von (Beta-V), die 2. von (Let) und die einzige von (Let-Rec).

Für eine *effizientere Stackverwaltung* im Falle der Funktion *eval* bieten sich deshalb folgende Verbesserungen an:

(a) Resultate werden entfernt, sobald man sie nicht mehr benötigt (was leicht an den big step Regeln zu erkennen ist).

(b) Knoten und Umgebung eines Arguments werden getrennt voneinander auf dem Stack abgelegt. Eine Umgebung wird nur dann aufgenommen, wenn sie sich von der Vorhergehenden unterscheidet. Ein Knoten κ wird nur dann aufgenommen, wenn durch (Beta-V) ein neuer Knoten κ' entsteht, der kein Kind oder Vorgänger von κ ist. (κ dient dann als „Rücksprungadresse“)

(c) Wegen (b) ist das aktuelle Argument (κ, η) nicht mehr aus dem Stack ablesbar, denn κ liegt gar nicht auf dem Stack und η kann „weit unten“ im Stack liegen, weil sich darüber beliebig viele Zwischenresultate angesammelt haben können. Deshalb merkt man sich κ und η *außerhalb* des Stacks. (Intuitiv kann man das η außerhalb des Stacks als Pointer in den Stack auffassen, den sogenannten Umgebungszeiger[WM92])

(d) Auch eine andere Information lässt sich nicht mehr aus dem Stack ablesen: Man erkennt – im Gegensatz zur naiven Stack-Verwaltung – nicht, ob man gerade ein Argument oder ein Resultat berechnet hat, d.h. ob man sich (im Sinne von Abbildung 3.3) links oder rechts von einem Knoten des Beweisbaums befindet. Diese Information merken wir uns durch einen Punkt, den wir vor oder hinter den aktuellen Knoten κ (des Syntaxbaums) setzen.

Diese (etwas vagen) Überlegungen werden jetzt formal durch eine small step Semantik zum Ausdruck gebracht. Anschließend wird die Äquivalenz zwischen big step und small step Semantik bewiesen.

3.2 Small step Semantik

Definition 3.1 (Positionen, Stacks und Konfigurationen) *Die Mengen Pos aller Positionen pos, Stack aller Stacks S und Conf aller Konfigurationen γ sind definiert durch:*

$$
\begin{array}{rcll}
pos & ::= & \cdot\kappa \mid \kappa\cdot & \\
S & ::= & \varepsilon & \textit{(leerer Stack)} \\
 & \mid & \kappa; S & \textit{(Rücksprungadresse)} \\
 & \mid & w; S & \textit{((Zwischen-)Ergebnis)} \\
 & \mid & \eta; S & \textit{(Umgebung)} \\
\gamma & ::= & (pos, \eta, S) &
\end{array}
$$

Die Funktion size : Stack → ℕ liefert die Anzahl der Einträge im Stack.

Definition 3.2 (Small step) *Ein small step ist eine Formel der Gestalt $\gamma \to \gamma'$. Ein solcher small step heißt gültig für das Gesamtprogramm e mit Syntaxbaum $T_e = (K, \ell)$, wenn er sich mit einer der in Abbildung 3.5 dargestellten Regeln herleiten lässt.*

Bei der Regel (Beta-V) wird eine Rücksprungadresse, sowie die alte Umgebung gesichert. Diese alte Umgebung kann man auch als *Controllink* bezeichnen. Sie wird gesichert, damit man diese nicht bei (Beta-V-End) wieder aus dem Stack heraus suchen müsste, da beliebig viele Zwischenresultate über der Umgebung liegen können.

Um es Konsistent zu halten, sichern wir bei (Let-Exec) auch die alte Umgebung in Form des Controllinks, wir ersparen uns aber die Rücksprungadresse, da hier nicht gesprungen wird.

Bei (Beta-V-End) und (Let-End) wird dann die Umgebung (und die Rücksprungadresse) wieder vom Stack genommen und als aktuelle Umgebung (und Position) in der Konfiguration genutzt.

Beispiel 3.3 In Abbildung 3.7 ist die komplette Berechnungsfolge für den Beispielausdruck $(\lambda x. * (x, 1))\, 7$, dessen Syntaxbaum in 3.6 zu sehen ist, in der optimierten small step Semantik gegeben.

3.3 Äquivalenz zwischen small step und big step Semantik

Es soll gezeigt werden, dass beide Semantiken in gewissem Sinne äquivalent sind. Dazu sind zwei Richtungen zu zeigen:

(1) Jeder big step lässt sich durch eine Folge von small steps simulieren.

(2) Für ein Gesamtprogramm e mit Syntaxbaum $T_e = (K, \ell)$ lässt sich die Folge $(\cdot\epsilon, [\,], [\,]; \varepsilon) \xrightarrow{*} (\epsilon\cdot, [\,], w; [\,]; \varepsilon)$ durch den big step $(\epsilon, [\,]) \Downarrow w$ simulieren.

Wir bezeichnen (1) als Vollständigkeit und (2) als Korrektheit der small step Semantik. (1) und (2) sind natürlich sehr vage formuliert und werden im Folgenden noch präzisiert.

Im weiteren Verlauf wird stillschweigend vorausgesetzt, dass sich big step und small step Semantik auf das gleiche „Gesamtprogramm“ e mit Syntaxbaum $T_e = (K, \ell)$ beziehen.

(Const)
$$\frac{\ell(\kappa) = c}{(\cdot\kappa, \eta, S) \to (\kappa\cdot, \eta, c; S)}$$

(Closure)
$$\frac{\ell(\kappa) = \lambda\, id}{(\cdot\kappa, \eta, S) \to (\kappa\cdot, \eta, (\kappa, \eta); S)}$$

(Id)
$$\frac{\ell(\kappa) = id \qquad lookup(\eta, id) = w}{(\cdot\kappa, \eta, S) \to (\kappa\cdot, \eta, w; S)}$$

(App-Left)
$$\frac{\ell(\kappa) = App}{(\cdot\kappa, \eta, S) \to (\cdot(\kappa.1), \eta, S)}$$

(App-Right)
$$\frac{\ell(\kappa) = App}{((\kappa.1)\cdot, \eta, S) \to (\cdot(\kappa.2), \eta, S)}$$

(Op)
$$\frac{\ell(\kappa) = App}{((\kappa.2)\cdot, \eta, (z_1, z_2); op; S) \to (\kappa\cdot, \eta, op^I(z_1, z_2); S)}$$

(Beta-V)
$$\frac{\ell(\kappa) = App \qquad \ell(\kappa') = \lambda\, id}{((\kappa.2)\cdot, \eta, w'; (\kappa', \eta'); S) \to (\cdot(\kappa'.1), (id : w'; \eta'), (id : w'; \eta'); \eta; \kappa; S)}$$

(Beta-V-End)
$$\frac{\ell(\kappa) = \lambda\, id}{((\kappa.1)\cdot, \eta, w; \eta''; \eta'; \kappa'; S) \to (\kappa'\cdot, \eta', w; S)}$$

(Proj)
$$\frac{\ell(\kappa) = App \qquad 1 \le i \le n}{((\kappa.2)\cdot, \eta, (w_1, \ldots, w_n); \#_i\ ; S) \to (\kappa\cdot, \eta, w_i; S)}$$

(Tuple)
$$\frac{\ell(\kappa) = Tuple\, n}{(\cdot\kappa, \eta, S) \to (\cdot(\kappa.1), \eta, S)}$$

(Tuple-Next)
$$\frac{\ell(\kappa) = Tuple\, n \qquad 1 \le m < n}{((\kappa.m)\cdot, \eta, S) \to (\cdot(\kappa.m+1), \eta, S)}$$

(Tuple-End)
$$\frac{\ell(\kappa) = Tuple\, n}{((\kappa.n)\cdot, \eta, w_n; \ldots; w_1; S) \to (\kappa\cdot, \eta, (w_1, \ldots, w_n); S)}$$

(Let-Eval)
$$\frac{\ell(\kappa) = \textbf{let}\ id}{(\cdot\kappa, \eta, S) \to (\cdot(\kappa.1), \eta, S)}$$

(Let-Exec)
$$\frac{\ell(\kappa) = \textbf{let}\ id}{((\kappa.1)\cdot, \eta, w; S) \to (\cdot(\kappa.2), (id : w; \eta), (id : w; \eta); \eta; S)}$$

(Let-End)
$$\frac{\ell(\kappa) = \textbf{let}\ id}{((\kappa.2)\cdot, \eta, w; \eta; \eta'; S) \to (\kappa\cdot, \eta', w; S)}$$

(Let-Rec)
$$\frac{\ell(\kappa) = \textbf{let rec}\ id}{(\cdot\kappa, \eta, S) \to (\cdot(\kappa.2), (id : ((\kappa.1), \circlearrowleft); \eta), (id : ((\kappa.1), \circlearrowleft); \eta); \eta; S)}$$

(Let-Rec-End)
$$\frac{\ell(\kappa) = \textbf{let rec}\ id}{((\kappa.2)\cdot, \eta, w; \eta; \eta'; S) \to (\kappa\cdot, \eta', w; S)}$$

(Cond-Eval)
$$\frac{\ell(\kappa) = Cond}{(\cdot\kappa, \eta, S) \to (\cdot(\kappa.1), \eta, S)}$$

(Cond-True)
$$\frac{\ell(\kappa) = Cond}{((\kappa.1)\cdot, \eta, true; S) \to (\cdot(\kappa.2), \eta, S)}$$

(Cond-False)
$$\frac{\ell(\kappa) = Cond}{((\kappa.1)\cdot, \eta, false; S) \to (\cdot(\kappa.3), \eta, S)}$$

(Cond-End)
$$\frac{\ell(\kappa) = Cond \qquad n \in \{2, 3\}}{((\kappa.n)\cdot, \eta, S) \to (\kappa\cdot, \eta, S)}$$

Abbildung 3.5: Small step Regeln

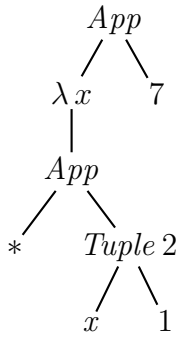

Abbildung 3.6: Syntaxbaum für $(\lambda x.\ *\ (x, 1))\, 7$

$$
\begin{aligned}
& (\cdot\epsilon, [\,], [\,]; \varepsilon) \\
\xrightarrow{\text{(App-Left)}}\ & (\cdot(\epsilon.1), [\,], [\,]; \varepsilon) \\
\xrightarrow{\text{(Closure)}}\ & ((\epsilon.1)\cdot, [\,], (1, [\,]); [\,]; \varepsilon) \\
\xrightarrow{\text{(App-Right)}}\ & (\cdot(\epsilon.2), [\,], (1, [\,]); [\,]; \varepsilon) \\
\xrightarrow{\text{(Const)}}\ & ((\epsilon.2)\cdot, [\,], 7; (1, [\,]); [\,]; \varepsilon) \\
\xrightarrow{\text{(Beta-V)}}\ & (\cdot(\epsilon.1.1), (x : 7; [\,]), (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(App-Left)}}\ & (\cdot(\epsilon.1.1.1), (x : 7; [\,]), (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon)) \\
\xrightarrow{\text{(Const)}}\ & ((\epsilon.1.1.1)\cdot, (x : 7; [\,]), *; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(App-Right)}}\ & (\cdot(\epsilon.1.1.2), (x : 7; [\,]), *; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Tuple)}}\ & (\cdot(\epsilon.1.1.2.1), (x : 7; [\,]), *; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Id)}}\ & ((\epsilon.1.1.2.1)\cdot, (x : 7; [\,]), 7; *; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Tuple-1)}}\ & (\cdot(\epsilon.1.1.2.2), (x : 7; [\,]), 7; *; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Const)}}\ & ((\epsilon.1.1.2.2)\cdot, (x : 7; [\,]), 1; 7; *; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Tuple-End)}}\ & ((\epsilon.1.1.2)\cdot, (x : 7; [\,]), (7, 1); *; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Op)}}\ & ((\epsilon.1.1)\cdot, (x : 7; [\,]), 7; (x : 7; [\,]); \epsilon; [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Beta-V-End)}}\ & (\epsilon\cdot, [\,], 7; [\,]; \varepsilon)
\end{aligned}
$$

Abbildung 3.7: Small step Berechnungsfolge für $(\epsilon, [\,], [\,]; \varepsilon)$ mit $e[\epsilon] = (\lambda x.\ *\ (x, 1))\, 7$

3.3.1 Vollständigkeit der small step Semantik

Satz 3.1 (Vollständigkeit der small step Semantik) *Für alle $\kappa \in K$, $\eta \in Env_{node}$ und $w \in W_{node}$ gilt:*

$$\textit{Wenn } (\kappa, \eta) \Downarrow w, \textit{ dann } (\cdot\kappa, \eta, S) \xrightarrow{*} (\kappa\cdot, \eta, w; S).$$

(In Worten: Jeder big step $(\kappa, \eta) \Downarrow w$ wird durch eine Folge von small steps simuliert, die „vor“ dem Knoten κ starten und „hinter“ dem Knoten κ endet und als „Nettoeffekt“ den Wert w auf dem Stack ablegt und zwar unabhängig davon, wie S aussieht.)

Beweis: Induktion über die Herleitung von $(\kappa, \eta) \Downarrow w$. (Nur die wichtigen Fälle)

- (Closure): Nach Voraussetzung gilt $(\kappa, \eta) \Downarrow (\kappa, \eta)$, sowie $\ell(\kappa) = \lambda\, id$, dann kann man auf die Konfiguration $(\cdot\kappa, \eta, S)$ die small step-Regel (Closure) anwenden und es gilt:
  - $(\cdot\kappa, \eta, S) \to (\kappa\cdot, \eta, (\kappa, \eta); S)$
- (Id): Nach Voraussetzung gilt $(\kappa, \eta) \Downarrow w$ mit den Prämissen $\ell(\kappa) = id$ und $lookup(\eta, id) = w$. Dann kann auf die Konfiguration $(\cdot\kappa, \eta, S)$ die small step-Regel (Id) angewendet werden und es gilt
  - $(\cdot\kappa, \eta, S) \to (\kappa\cdot, \eta, w; S)$.
- (Beta-V): Nach Voraussetzung gilt $(\kappa, \eta) \Downarrow w$. Dazu müssen folgende Prämissen gelten:

  1.) $\ell(\kappa) = App$
  2.) $((\kappa.1), \eta) \Downarrow (\kappa', \eta')$
  3.) $\ell(\kappa') = \lambda\, id'$
  4.) $((\kappa.2), \eta) \Downarrow w'$
  5.) $((\kappa'.1), id' : w'; \eta') \Downarrow w$

  Wegen 1. lässt sich (App-Left) anwenden auf die Konfiguration $(\cdot\kappa, \eta, S)$:

  6.) $(\cdot\kappa, \eta, S) \to (\cdot(\kappa.1), \eta, S)$

  Aus 2. erhält man mit der I.V.:

  7.) $(\cdot(\kappa.1), \eta, S) \xrightarrow{*} ((\kappa.1)\cdot, \eta, (\kappa', \eta'); S)$

  Wegen 1. ist nun (App-Right) anwendbar:

  8.) $((\kappa.1)\cdot, \eta, (\kappa', \eta'); S) \to (\cdot(\kappa.2), \eta, (\kappa', \eta'); S)$

  Aus 4. erhält man mit der I.V.:

  9.) $(\cdot(\kappa.2), \eta, (\kappa', \eta'); S) \xrightarrow{*} ((\kappa.2)\cdot, \eta, w'; (\kappa', \eta'); S)$

  Wegen 1. und 3. ist small step-Regel (Beta-V) anwendbar und es gilt:

  10.) $((\kappa.2)\cdot, \eta, w'; (\kappa', \eta'); S) \to (\cdot(\kappa'.1), (id' : w'; \eta'), (id' : w'; \eta'); \eta; \kappa; S)$

Auf 5. kann man I.V. anwenden und es folgt:

11.) $(\cdot(\kappa'.1), (id' : w'; \eta'), (id' : w'; \eta'); \eta; \kappa; S) \xrightarrow{*} ((\kappa'.1)\cdot, (id' : w'; \eta'), w; (id' : w'; \eta'); \eta; \kappa; S)$

Wegen 3. ist (Beta-V-End) anwendbar:

12.) $((\kappa'.1)\cdot, (id' : w'; \eta'), w; (id' : w'; \eta'); \eta; \kappa; S) \rightarrow (\kappa\cdot, \eta, w; S)$

Insgesamt ergibt sich aus 6. bis 12.: $(\cdot\kappa, \eta, S) \xrightarrow{*} (\kappa\cdot, \eta, w; S)$

- (Let): Nach Voraussetzung gilt $(\kappa, \eta) \Downarrow w$. Dazu müssen die folgenden Prämissen gelten:

  1.) $\ell(\kappa) = \textbf{let}\ id$

  2.) $((\kappa.1), \eta) \Downarrow w'$

  3.) $((\kappa.2), id : w'; \eta) \Downarrow w$

  Wegen 1. ist auf die Konfiguration $(\cdot\kappa, \eta, S)$ (Let-Eval) anwendbar:

  4.) $(\cdot\kappa, \eta, S) \rightarrow (\cdot(\kappa.1), \eta, S)$

  Auf 2. lässt sich I.V. anwenden und damit folgt:

  5.) $(\cdot(\kappa.1), \eta, S) \xrightarrow{*} ((\kappa.1)\cdot, \eta, w'; S)$

  Nun lässt sich wegen 1. (Let-Exec) anwenden:

  6.) $((\kappa.1)\cdot, \eta, w'; S) \rightarrow (\cdot(\kappa.2), (id : w'; \eta), (id : w'; \eta); \eta; S)$

  Auf 3. kann I.V. angewendet werden also gilt:

  7.) $(\cdot(\kappa.2), (id : w'; \eta), (id : w'; \eta); \eta; S) \xrightarrow{*}$
  $((\kappa.2)\cdot, (id : w'; \eta), w; (id : w'; \eta); \eta; S)$

  Dann ist wegen 1. die Regel (Let-End) anwendbar und es folgt:

  8.) $((\kappa.2)\cdot, (id : w'; \eta), w; (id : w'; \eta); \eta; S) \rightarrow (\kappa\cdot, \eta, w; S)$

  Insgesamt ergibt sich aus 4. bis 8.: $(\cdot\kappa, \eta, S) \xrightarrow{*} (\kappa\cdot, \eta, w; S)$

- (Let-Rec): Nach Voraussetzung gilt: $(\kappa, \eta) \Downarrow w$. Dazu müssen die folgenden Prämissen gelten:

  1.) $\ell(\kappa) = \textbf{let rec}\ id$

  2.) $((\kappa.2), id : ((\kappa.1), \circlearrowleft); \eta) \Downarrow w$

  Wegen 1. kann auf die Konfiguration $(\cdot\kappa, \eta, S)$ die small step-Regel (Let-Rec) angewendet werden:

  3.) $(\cdot\kappa, \eta, S) \rightarrow (\cdot(\kappa.2), (id : ((\kappa.1), \circlearrowleft); \eta), (id : ((\kappa.1), \circlearrowleft); \eta); \eta; S)$

  Auf 2. kann I.V. angewendet werden und es ergibt sich:

  4.) $(\cdot(\kappa.2), (id : ((\kappa.1), \circlearrowleft); \eta), (id : ((\kappa.1), \circlearrowleft); \eta); \eta; S) \xrightarrow{*}$
  $((\kappa.2)\cdot, (id : ((\kappa.1), \circlearrowleft); \eta), w; (id : ((\kappa.1), \circlearrowleft); \eta); \eta; S)$

  Dann kann wegen 1. die Regel (Let-Rec-End) angewendet werden:

5.) $((\kappa.2)\cdot, (id : ((\kappa.1), \circlearrowleft); \eta), w; (id : ((\kappa.1), \circlearrowleft); \eta); \eta; S) \to (\kappa\cdot, \eta, w; S)$

Insgesamt ergibt sich aus 3. bis 5.: $(\cdot\kappa, \eta, S) \xrightarrow{*} (\kappa\cdot, \eta, w; S)$ □

3.3.2 Korrektheit der small step Semantik

Um die Korrektheit beweisen zu können, brauchen wir zunächst noch zwei Lemmata, die uns später beim Beweis helfen werden.

Zunächst müssen wir etwas über die Knoten wissen die während einer Berechnung auf dem Stack abgelegt werden.

Lemma 3.1 *Jeder Knoten κ, der während einer Berechnungsfolge auf dem Stack abgelegt wird, ist ein Applikationsknoten, d.h. es gilt $\ell(\kappa) = App$*

Beweis: Die einzige Regel, mit der ein Knoten κ auf dem Stack abgelegt wird, ist (Beta-V) und in der Prämisse wird verlangt, dass $\ell(\kappa) = App$ ist. □

Nun benötigen wir noch eine Aussage darüber, wie man zu gewissen Punkten in einer Berechnungsfolge kommt. Man stellt sich also die Fragen: wie kommt man zu einer Konfiguration, in der der Punkt vor einem Knoten steht und wie kommt man zu einer, in der der Punkt hinter einem Knoten steht. Außerdem bringt uns dieses Lemma eine zwar relativ klare aber auch sehr wichtige Eigenschaft, nämlich, dass man nicht mehr vor die Wurzel des Syntaxbaumes innerhalb einer Berechnung springen kann.

Lemma 3.2 *Sei γ eine Konfiguration in der small step Semantik. Dann gilt:*

(a) *Wenn $\gamma \xrightarrow{R} (\cdot(\kappa.i), \eta, S)$, dann ist*

- $R =$ (App-Left), *wenn* $\ell(\kappa) = App$ *und* $i = 1$.
- $R =$ (App-Right), *wenn* $\ell(\kappa) = App$ *und* $i = 2$.
- $R =$ (Beta-V), *wenn* $\ell(\kappa) = \lambda\, id$.
- $R =$ (Cond-Eval), *wenn* $\ell(\kappa) = Cond$ *und* $i = 1$.
- $R =$ (Cond-True), *wenn* $\ell(\kappa) = Cond$ *und* $i = 2$.
- $R =$ (Cond-False), *wenn* $\ell(\kappa) = Cond$ *und* $i = 3$.
- $R =$ (Tuple), *wenn* $\ell(\kappa) = Tuple\, n$ *und* $i = 1$.
- $R =$ (Tuple-Next), *wenn* $\ell(\kappa) = Tuple\, n$ *und* $1 < i \leq n$.
- $R =$ (Let-Eval), *wenn* $\ell(\kappa) =$ **let** id *und* $i = 1$.
- $R =$ (Let-Exec), *wenn* $\ell(\kappa) =$ **let** id *und* $i = 2$.
- $R =$ (Let-Rec), *wenn* $\ell(\kappa) =$ **let rec** id *und* $i = 2$.

(b) *Wenn $\gamma \xrightarrow{R} (\kappa\cdot, \eta, S)$, dann ist*

- $R =$ (Const), *wenn* $\ell(\kappa) = c$.

- $R = (\textsc{Closure})$, *wenn* $\ell(\kappa) = \lambda\, id$.
- $R = (\textsc{Id})$, *wenn* $\ell(\kappa) = id$.
- *entweder* $R = (\textsc{Beta-V-End})$, $R = (\textsc{Op})$ *oder* $R = (\textsc{Proj})$, *wenn* $\ell(\kappa) = App$.
- $R = (\textsc{Cond-End})$, *wenn* $\ell(\kappa) = Cond$.
- $R = (\textsc{Tuple-End})$, *wenn* $\ell(\kappa) = Tuple\, n$.
- $R = (\textsc{Let-End})$, *wenn* $\ell(\kappa) = \textbf{let}\, id$.
- $R = (\textsc{Let-Rec-End})$, *wenn* $\ell(\kappa) = \textbf{let rec}\, id$.

(c) $\gamma \xrightarrow{R} (\cdot\epsilon, \eta', S')$ *ist nicht möglich.*

Beweis: Ergibt sich leicht aus den jeweiligen Regeln. Der einzige nicht triviale Fall ist, dass man mit (Beta-V-End) immer hinter einem Applikationsknoten ankommt. In (Beta-V) wird nämlich ein Knoten vom Stack als neuer aktueller Knoten genommen und nach Lemma 3.1 kann dies nur ein Applikationsknoten sein. □

Nun sind wir so weit, dass wir die Korrektheit formulieren und beweisen können.

Satz 3.2 (Korrektheit der small step Semantik) *Sei* $\kappa \in K$, $\eta_1 \in Env_{node}$ *und* $S' \in Stack$.

> *Wenn* $(\cdot\epsilon, \eta_1, S') \xrightarrow{n} (\kappa\cdot, \eta, w; S)$, *dann existiert* $m < n$ *mit* $(\cdot\epsilon, \eta_1, S') \xrightarrow{m} (\cdot\kappa, \eta, S)$ *und* $(\kappa, \eta) \Downarrow w$.

(In Worten: Wenn man ausgehend von der Wurzel hinter einem Knoten κ *angekommen ist, dann hat man als letztes einen big step für* κ *simuliert.)*

Beweis: Induktion über n und Fallunterscheidung nach der Form von $\ell(\kappa)$ (Nur die interessanten Fälle)

- $\ell(\kappa) = \lambda\, id$

  Dann kann nach Lemma 3.2 nur die small step Regel (Closure) angewendet worden sein und es gilt $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} (\cdot\kappa, \eta, S) \xrightarrow{(\textsc{Closure})} (\kappa\cdot, \eta, (\kappa, \eta); S)$. Man kann also $m = n - 1$ wählen.

  Auf die Konfiguration (κ, η) ist wegen der Voraussetzung $\ell(\kappa) = \lambda\, id$ die big step Regel (Closure) anwendbar und es gilt $(\kappa, \eta) \Downarrow (\kappa, \eta)$.

- $\ell(\kappa) = id$

  Wegen Lemma 3.2 muss, da nur die small step Regel (Id) anwendbar ist, $(\cdot\epsilon, \eta_1, S')$
  $\xrightarrow{n-1} (\cdot\kappa, \eta, S) \xrightarrow{(\textsc{Id})} (\kappa\cdot, \eta, w; S)$, sowie die Prämisse $lookup(\eta, id) = w$ gelten. Dann kann man $m = n - 1$ wählen.

  Auf die Konfiguration (κ, η) kann man, da die beiden Prämissen erfüllt sind, die big step Regel (Id) anwenden und es gilt $(\kappa, \eta) \Downarrow w$.

- $\ell(\kappa) = Cond$

  Dann gilt, da $\epsilon \neq (\kappa.2)$ und daher $n > 0$:

  (a) entweder $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} ((\kappa.2)\cdot, \eta, w; S) \xrightarrow{(\text{Cond-End})} (\kappa\cdot, \eta, w; S)$

  (b) oder $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} ((\kappa.3)\cdot, \eta, w; S) \xrightarrow{(\text{Cond-End})} (\kappa\cdot, \eta, w; S)$

  – Im Fall (a) existiert nach I.V. ein $m_1 < n-1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1} (\cdot(\kappa.2), \eta, S)$ und $((\kappa.2), \eta) \Downarrow w$.

  Nach Lemma 3.2 kann nur die small step Regel (Cond-True) angewendet worden sein und es gilt $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1-1} ((\kappa.1)\cdot, \eta, true; S) \xrightarrow{(\text{Cond-True})} (\cdot(\kappa.2), \eta, S)$. Nach I.V. existiert ein $m_2 < m_1 - 1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2} (\cdot(\kappa.1), \eta, S)$ und $((\kappa.1), \eta) \Downarrow true$.

  Es kann, wegen Lemma 3.2, nur (Cond-Eval) angewendet worden sein und es folgt $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2-1} (\cdot\kappa, \eta, S) \xrightarrow{(\text{Cond-Eval})} (\cdot(\kappa.1), \eta, S)$, also kann man $m = m_2 - 1$ wählen.

  Insgesamt sind alle Prämissen für die big step Regel (Cond-True) erfüllt und es ergibt sich $(\kappa, \eta) \Downarrow w$.

  – (b) analog

- $\ell(\kappa) = App$

  Dann gilt wegen Lemma 3.2:

  (a) entweder es gilt $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} ((\kappa.2)\cdot, \eta, (z_1, z_2); op; S) \xrightarrow{(\text{Op})} (\kappa\cdot, \eta, op^I(z_1, z_2); S)$ mit $w = op^I(z_1, z_2)$

  (b) oder $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} ((\kappa.2)\cdot, \eta, (w_1, \ldots, w_n); \#_i\,; S) \xrightarrow{(\text{Proj})} (\kappa\cdot, \eta, w_i; S)$ mit $w = w_i$

  (c) oder $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} ((\kappa'.1)\cdot, \eta', w; \eta'''; \kappa; \eta; S) \xrightarrow{(\text{Beta-V-End})} (\kappa\cdot, \eta, w; S)$.

  – Im Fall (a) existiert nach I.V. ein $m_1 < n-1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1} (\cdot(\kappa.2), \eta, op; S)$ und $((\kappa.2), \eta) \Downarrow (z_1, z_2)$.

  Wegen Lemma 3.2 folgt, dass nur (App-Right) angewendet werden konnte, $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1-1} ((\kappa.1)\cdot, \eta, op; S) \xrightarrow{(\text{App-Right})} (\cdot(\kappa.2), \eta, op; S)$. Nach I.V. existiert ein $m_2 < m_1 - 1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2} (\cdot(\kappa.1), \eta, S)$ und

  1.) $((\kappa.1), \eta) \Downarrow op$.

  Aus Lemma 3.2 folgt, dass nur (App-Left) angewendet wurde und es ergibt sich $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2-1} (\cdot\kappa, \eta, S) \xrightarrow{(\text{App-Left})} (\cdot(\kappa.1), \eta, S)$. Also kann man $m = m_2 - 1$ wählen.

  Insgesamt sind alle drei Prämissen der big step Regel (Op) erfüllt und es ergibt sich $(\kappa, \eta) \Downarrow w$.

  – Fall (b) analog.

– Im Fall (c) existiert nach I.V. ein $m_1 < n-1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1} (\cdot(\kappa'.1), \eta', \eta'''; \kappa; \eta; S)$ und

1.) $((\kappa'.1), \eta') \Downarrow w$.

Dann muss, da wegen Lemma 3.2 nur small step Regel (Beta-V) angewendet werden konnte,
$(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1-1} ((\kappa.2)\cdot, \eta, w'; (\kappa', \eta''); S)$
$\xrightarrow{\text{(Beta-V)}} (\cdot(\kappa'.1), (id : w'; \eta''), (id : w'; \eta''); \eta; \kappa; S)$ folgen und es gilt $\eta' = \eta''' = (id : w'; \eta'')$ und

2.) $\ell(\kappa') = \lambda\, id$.

Nach I.V. existiert $m_2 < m_1 - 1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2} (\cdot(\kappa.2), \eta, (\kappa', \eta''); S)$ und

3.) $((\kappa.2), \eta) \Downarrow w'$.

Da, wegen Lemma 3.2, nun nur (App-Right) angewendet wurde, folgt $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2-1} ((\kappa.1)\cdot, \eta, (\kappa', \eta''); S) \xrightarrow{\text{(App-Right)}} (\cdot(\kappa.2), \eta, (\kappa', \eta''); S)$.
Dann existiert nach I.V. $m_3 < m_2 - 1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_3} (\cdot(\kappa.1), \eta, S)$ und

4.) $((\kappa.1), \eta) \Downarrow (\kappa', \eta'')$.

Wegen Lemma 3.2 ist klar, dass nur (App-Left) angewendet wurde und es folgt $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_3-1} (\cdot\kappa, \eta, S) \xrightarrow{\text{(App-Left)}} (\cdot(\kappa.1), \eta, S)$. Damit kann man $m = m_3 - 1$ wählen.

Insgesamt sind alle fünf Prämissen der big step Regel (Beta-V) erfüllt ($\ell(\kappa) = App$ nach Voraussetzung, sowie 1.,2.,3. und 4.) und es gilt $(\kappa, \eta) \Downarrow w$.

- $\ell(\kappa) = \textbf{let}\ id$

  Dann kann nach Lemma 3.2 nur (Let-End) angewendet worden sein und es gilt $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} ((\kappa.2)\cdot, \eta', w; \eta'; \eta; S) \xrightarrow{\text{(Let-End)}} (\kappa\cdot, \eta, w; S)$

  Nun existiert nach I.V. ein $m_1 < n - 1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1} (\cdot(\kappa.2), \eta', \eta'; \eta; S)$ und $((\kappa.2), \eta') \Downarrow w$.

  Wegen Lemma 3.2 muss (Let-Exec) angewendet worden sein und es folgt $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1-1} ((\kappa.1)\cdot, \eta, w'; S) \xrightarrow{\text{(Let-Exec)}} (\cdot(\kappa.2), (id : w'; \eta), (id : w'; \eta); \eta; S)$ mit $(id : w'; \eta) = \eta'$.

  Nach I.V. existiert ein $m_2 < m_1 - 1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2} (\cdot(\kappa.1), \eta, S)$ und $((\kappa.1), \eta) \Downarrow w'$. Aus Lemma 3.2 folgt, dass nur (Let-Eval) angewendet werden konnte und es gilt $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_2-1} (\cdot\kappa, \eta, S) \xrightarrow{\text{(Let-Eval)}} (\cdot(\kappa.1), \eta, S)$. Also kann man $m = m_2 - 1$ wählen.

  Insgesamt sind alle Prämissen der big step Regel (Let) erfüllt und es ergibt sich $(\kappa, \eta) \Downarrow w$.

- $\ell(\kappa) = \textbf{let rec}\ id$

Dann kann nach Lemma 3.2 nur (Let-Rec-End) angewendet worden sein und es gilt $(\cdot\epsilon, \eta_1, S') \xrightarrow{n-1} ((\kappa.2)\cdot, \eta', w; \eta'; \eta; S) \xrightarrow{\text{(Let-Rec-End)}} (\kappa\cdot, \eta, w; S)$.

Nach I.V. existiert ein $m_1 < n-1$ mit $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1} (\cdot(\kappa.2), \eta', \eta'; \eta; S)$ und $((\kappa.2), \eta') \Downarrow w$.

Wegen Lemma 3.2 kann nur (Let-Rec) angewendet worden sein und es gilt $(\cdot\epsilon, \eta_1, S') \xrightarrow{m_1-1} (\cdot\kappa, \eta, S) \xrightarrow{\text{(Let-Rec)}} (\cdot(\kappa.2), \eta', \eta'; \eta; S)$
mit $\eta' = (id : ((\kappa.1), \circlearrowleft); \eta)$. Also kann man $m = m_1 - 1$ wählen.

Auf (κ, η) kann man die big step Regel (Let-Rec) anwenden, da alle Prämissen erfüllt sind, und es ergibt sich $(\kappa, \eta) \Downarrow w$ □

Aus Satz 3.2 ergibt sich das folgende Korollar, welches die für uns interessante Berechnung von vor der Wurzel bis hinter die Wurzel beinhaltet.

Korollar 3.1 *Sei $e_0 \in Exp$, $T_{e_0} = (K, \ell)$ der Syntaxbaum von e_0 und $\kappa \in K$, sowie $\eta_1 \in Env_{node}$ und $S' \in Stack$.*

Wenn $(\cdot\epsilon, \eta_1, S') \xrightarrow{n} (\epsilon\cdot, \eta, w; S)$, dann kann nur $m = 0$ gelten, also $(\epsilon, \eta_1) \Downarrow w$.

3.3.3 Äquivalenz

Nun folgt die Äquivalenz einfacherweise aus dem Satz bzw. dem Korollar aus den letzten beiden Abschnitten.

Satz 3.3 (Äquivalenz zwischen small step und big step Semantik)
Für die Wurzel ϵ des Gesamtprogramms gilt:

$(\epsilon, [\,]) \Downarrow w$ *genau dann, wenn* $(\cdot\epsilon, [\,], [\,]; \varepsilon) \xrightarrow{*} (\epsilon\cdot, [\,], w; [\,]; \varepsilon)$.

Beweis: Folgt unmittelbar aus Satz 3.1, sowie Korollar 3.1. □

Im weiteren werden wir dann nur noch diese Semantiken benutzen. Also immer wenn dann von Umgebungssemantik, big step oder small step Semantik die Rede ist, sind die in diesem Kapitel definierten Semantiken gemeint.

4 Namenlose Umgebungen

Bisher suchen wir, um den Eintrag für einen Namen id in einer Umgebung η zu finden, nach (dem ersten Vorkommen von) id in η. Diese Suche kann man sich ersparen, da man „zur Compilezeit“ vorhersagen kann, an welcher Stelle der Eintrag für id steht:

Für jede Closure (κ, η), die während der Auswertung des Gesamtprogramms e auftritt, lässt sich schon aus κ erkennen, welche Namen in η vorkommen und an welcher Stelle ein Name steht. Es sind nämlich genau die Namen, die auf dem Weg von κ zur Wurzel von e durch $\lambda\, id$, **let** id und **let rec** id gebunden wurden, wobei der i-te Name, den man auf dem Weg findet, auch an der i-ten Stelle in η steht.

Beispiel 4.1 Betrachten wir den Ausdruck $((\lambda x.\, \lambda y.\, +\, (x, y))\, 1)\, 2$ dessen Syntaxbaum in Abbildung 4.1 dargestellt ist.

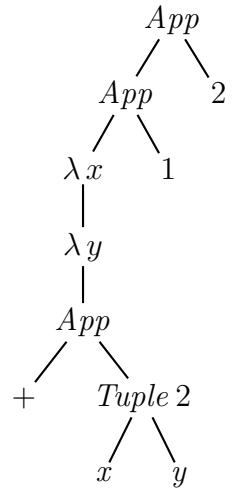

Abbildung 4.1: Syntaxbaum für $((\lambda x.\, \lambda y.\, +\, (x, y))\, 1)\, 2$

Im Knoten *Tuple* 2 würde die Umgebung η dann von der Form $y : w_y; x : w_x$ sein. Die Werte kann man natürlich zur Compilezeit nicht unbedingt bestimmen.

Diese Beobachtung kann man nutzen, um die Namen aus den (Laufzeit-) Umgebungen zu entfernen. Für jeden Knoten κ sei Γ_κ die Liste der Namen, die man auf dem Weg von κ zur Wurzel findet. Wenn $\Gamma_\kappa = id_1; \ldots; id_n$, dann ist jede (Laufzeit-) Umgebung η, die zusammen mit κ auftritt, von der Form $id_1 : w_1; \ldots; id_n : w_n$, also kann man η durch eine *namenlose Umgebung* $\rho = w_1; \ldots; w_n$ ersetzen, weil man schon zur Compilezeit weiß, dass man den Eintrag für id_i an der i-ten Stelle von ρ

findet. Diese Beobachtung geht auf de Bruijn zurück [DB72] und wurde in [AO09] bereits verwendet. Hier wollen wir aber auf die (unnötige) Übersetzung von Ausdrücken verzichten. Statt einen Namen id durch einen *de Bruijn Index* zu ersetzen, ordnen wir jedem Knoten κ mit $\ell(\kappa) = id$ nur einen de Bruijn Index $\imath_\kappa$ zu.

Darüber hinaus nutzen wir noch eine weitere Einsparmöglichkeit:

Für einen Namen id, der durch **let rec** $id = v$ **in** e eingeführt wird, steht in der Laufzeitumgebung η stets der Eintrag $id : (\kappa, \circlearrowleft)$, wobei κ die Wurzel von v ist. In diesem Fall ist also sogar der Eintrag für id schon zur Compilezeit bekannt. Das bedeutet, dass man den Eintrag für id ganz aus der Laufzeitumgebung η entfernen kann. Dazu muss man sich in der Namensliste für einen solchen Namen id natürlich den Knoten κ merken (den man durch $lookup(id, \eta)$ finden würde) und die de Bruijn-Zählweise anpassen, damit solche Namen nicht mehr mitgezählt werden.

Diese Überlegungen führen zu folgenden Definitionen.

Definition 4.1 (Indizes und Namenskontexte) *Die Menge Index aller Indizes $\imath$ ist definiert durch:*

$$\begin{array}{rcll} \imath & ::= & d & \textit{(normaler Index mit } d \in \mathbb{N}\textit{)} \\ & | & (d, \kappa) & \textit{(Index für Rekursion mit } d \in \mathbb{N}\textit{)} \end{array}$$

Die Menge NC aller Namenskontexte Γ ist definiert durch:

$$\begin{array}{rcll} \Gamma & ::= & [\,] & \\ & | & id; \Gamma & \textit{(Normaler Eintrag)} \\ & | & id : \kappa; \Gamma & \textit{(Eintrag für Rekursion)} \end{array}$$

Für jedes κ sei der zu κ gehörige Namenskontext Γ_κ definiert durch:

$$\begin{array}{rcll} \Gamma_\epsilon & = & [\,] & \\ \Gamma_{\kappa.i} & = & id; \Gamma_\kappa & \textit{falls } \ell(\kappa) = \lambda\, id \textit{ und } i = 1 \\ & & & \textit{oder } \ell(\kappa) = \textbf{let}\, id \textit{ und } i = 2 \\ \Gamma_{\kappa.i} & = & id : (\kappa.1); \Gamma_\kappa & \textit{falls } \ell(\kappa) = \textbf{let rec}\, id \textit{ und } i \in \{1, 2\} \\ \Gamma_{\kappa.i} & = & \Gamma_\kappa & \textit{sonst} \end{array}$$

Definition 4.2 (Namenlose Umgebungen und Werte)
Die Menge $Env_{nameless}$ der namenlosen Umgebungen ρ ist definiert durch:

$$\begin{array}{rcll} \rho & ::= & [\,] & \\ & | & \omega; \rho & \textit{(Eintrag)} \end{array}$$

Damit ist eine namenlose Umgebung von der Form: $\rho = [\omega_0; \ldots; \omega_m]$. Außerdem ist $\rho(d)$ der d-te Eintrag in ρ (undefiniert wenn ρ weniger Einträge hat).

Die Menge Ω der namenlosen Werte ω ist definiert durch:

$$\begin{array}{rcll} \omega & ::= & c & \textit{(Konstante)} \\ & | & (\kappa, \rho) & \textit{(Closure)} \\ & | & (\omega_1, \ldots, \omega_n) & \textit{(Tuple mit } n > 1\textit{)} \end{array}$$

Zusätzlich definieren wir uns noch eine Funktion *inc* die einen Index inkrementiert. Diese wird uns bei der nun folgenden Definition von *lookup* und *expand* die Schreibweise etwas vereinfachen.

Jetzt müssen wir wieder die Funktionen *lookup* und *expand* anpassen. Dazu benötigen wir jetzt noch eine neue *slookup*-Funktion (statische *lookup*-Funktion), die im Namenskontext Γ den Namen *id* nachschlägt und dessen Index liefert. Bei *slookup* müssen wir nun aufpassen dass der Index nicht durch Rekursionseinträge vergrößert wird, da wir uns diese ja bei den Umgebungen gespart haben und sonst der Index nicht zur Umgebung passt.

Die Funktion *slookup* kann man bereits zur Compilezeit berechnen und zur Laufzeit werden dann nur noch die Indizes verwendet. Dies wird dadurch sichergestellt, dass für jeden *id*-Knoten κ, der Index $\imath_\kappa$ gebildet wird.

Definition 4.3 (Namenloses *lookup* und *expand*)
Die partielle Funktion $slookup : NC \times Id \to Index$ *ist induktiv definiert durch:*

$$slookup(\Gamma, id) \quad = \quad \begin{cases} 0 & falls\ \Gamma = (id'; \Gamma')\ und\ id = id' \\ inc(slookup(\Gamma', id)) & falls\ \Gamma = (id'; \Gamma')\ und\ id \neq id' \\ (0, \kappa') & falls\ \Gamma = (id' : \kappa'; \Gamma')\ und\ id = id' \\ slookup(\Gamma', id) & falls\ \Gamma = (id' : \kappa'; \Gamma')\ und\ id \neq id' \\ undef. & sonst\ (\Gamma = [\,]) \end{cases}$$

Die partielle Funktion $lookup : Env_{nameless} \times Index \to \Omega$ *ist definiert durch:*

$$lookup(\rho, \imath) \quad = \quad \begin{cases} expand(\rho, \kappa) & falls\ \imath = (0, \kappa) \\ \omega & falls\ \imath = 0\ und\ \rho = \omega; \rho \\ lookup(\rho', d) & falls\ \imath = d + 1\ und\ \rho = \omega; \rho \\ lookup(\rho', (d, \kappa)) & falls\ \imath = (d + 1, \kappa)\ und\ \rho = \omega; \rho \\ undef. & sonst \end{cases}$$

Die partielle Funktion $expand : Env_{nameless} \times Node \to \Omega$ *ist definiert durch:*

$$expand(\rho, \kappa) \quad = \quad \begin{cases} c & falls\ \ell(\kappa) = c \in Const \\ (\kappa, \rho) & falls\ \ell(\kappa) = \lambda\, id \\ (expand(\rho, \kappa.1), \ldots, expand(\rho, \kappa.n)) & falls\ \ell(\kappa) = Tuple\, n \\ undef. & sonst \end{cases}$$

Für alle κ *mit* $\ell(\kappa) = id$ *wird ein Index* $\imath_\kappa$ *definiert durch:*

$$\imath_\kappa = slookup(\Gamma_\kappa, id)$$

4.1 Namenlose big step Semantik

Nun können wir auch die big step Regeln anpassen. Dabei werden jetzt überall die neuen Umgebungen verwendet. Ansonsten ändern sich eigentlich nur die Regeln (Id), dort wird das neue *lookup* mit dem Index $\imath_\kappa$ genutzt und (Let-Rec), wo jetzt gar nichts mehr in die Umgebung eingetragen wird, da dies alles über das *lookup* funktioniert.

Definition 4.4 (Big step) *Ein big step in der namenlosen Umgebungssemantik ist eine Formel der Gestalt $(\kappa, \rho) \Downarrow \omega$, wobei $\kappa \in Node$ sowie $\rho \in Env_{nameless}$ und $\omega \in \Omega$. Ein deratiger big step heißt* gültig *bzgl. des Syntaxbaums $T_{e_0} = (K, \ell)$, wenn er sich mit den in Abbildung 4.2 dargestellten Regeln herleiten lässt.*

$$\frac{\ell(\kappa) = c}{(\kappa,\rho) \Downarrow c}\ (\textsc{Const}) \qquad \frac{\ell(\kappa) = \lambda\, id}{(\kappa,\rho) \Downarrow (\kappa,\rho)}\ (\textsc{Closure}) \qquad \frac{\ell(\kappa) = id \qquad lookup(\rho, \imath_\kappa) = \omega}{(\kappa,\rho) \Downarrow \omega}\ (\textsc{Id})$$

$$\frac{\ell(\kappa) = App \qquad ((\kappa.1),\rho) \Downarrow op \qquad ((\kappa.2),\rho) \Downarrow (z_1, z_2)}{(\kappa,\eta) \Downarrow op^I(z_1, z_2)}\ (\textsc{Op})$$

$$\frac{\ell(\kappa) = App \qquad ((\kappa.1),\rho) \Downarrow (\kappa',\rho') \qquad \ell(\kappa') = \lambda\, id' \qquad ((\kappa.2),\rho) \Downarrow \omega' \qquad ((\kappa'.1),\omega';\rho') \Downarrow \omega}{(\kappa,\rho) \Downarrow \omega}\ (\textsc{Beta-V})$$

$$\frac{\ell(\kappa) = App \qquad ((\kappa.1),\rho) \Downarrow \#_i \qquad ((\kappa.2),\rho) \Downarrow (\omega_1,\ldots,\omega_n) \qquad 1 \le i \le n}{(\kappa,\rho) \Downarrow \omega_i}\ (\textsc{Proj})$$

$$\frac{\ell(\kappa) = Tuple\, n \qquad ((\kappa.i),\rho) \Downarrow \omega_i \qquad 1 \le i \le n}{(\kappa,\rho) \Downarrow (\omega_1,\ldots,\omega_n)}\ (\textsc{Tuple})$$

$$\frac{\ell(\kappa) = \textbf{let}\ id \qquad ((\kappa.1),\rho) \Downarrow \omega' \qquad ((\kappa.2),\omega';\rho) \Downarrow \omega}{(\kappa,\rho) \Downarrow \omega}\ (\textsc{Let}) \qquad \frac{\ell(\kappa) = \textbf{let rec}\ id \qquad ((\kappa.2),\rho) \Downarrow \omega}{(\kappa,\rho) \Downarrow \omega}\ (\textsc{Let-Rec})$$

$$\frac{\ell(\kappa) = Cond \qquad ((\kappa.1),\rho) \Downarrow true \qquad ((\kappa.2),\rho) \Downarrow \omega}{(\kappa,\rho) \Downarrow \omega}\ (\textsc{Cond-True})$$

$$\frac{\ell(\kappa) = Cond \qquad ((\kappa.1),\rho) \Downarrow false \qquad ((\kappa.3),\rho) \Downarrow \omega}{(\kappa,\rho) \Downarrow \omega}\ (\textsc{Cond-False})$$

Abbildung 4.2: Big step Regeln für die namenlose Umgebungssemantik

Im Folgenden wird wieder stillschweigend vorausgesetzt, dass wir immer mit einem „Gesamtprogramm“ e mit Syntaxbaum $T_e = (K, \ell)$ arbeiten.

4.1.1 Zusammenhang zwischen namenloser big step und big step Semantik

Um einen Zusammenhang zwischen den beiden Umgebungssemantiken herzustellen brauchen wir zunächst die Definition einer Äquivalenz der beiden Umgebungsarten, sowie eine Äquivalenz der beiden Wertearten.

Definition 4.5 *Sei $\kappa \in K$, $\eta \in Env$ und $\rho \in Env_{nameless}$, sowie $w \in W$ und $\omega \in \Omega$. Dann heißt $w \sim \omega$ und $\kappa \models \eta \sim \rho$ gültig, wenn es sich mit den Regeln aus Abbildung 4.3 herleiten lässt.*

$$\text{(Const)}\quad c \sim c$$

$$\text{(Closure)}\quad \frac{\kappa \models \eta \sim \rho}{(\kappa, \eta) \sim (\kappa, \rho)}$$

$$\text{(Tuple)}\quad \frac{w_i \sim \omega_i \quad i = 1, \ldots, n}{(w_1, \ldots, w_n) \sim (\omega_1, \ldots, \omega_n)}$$

$$\text{(Empty)}\quad \epsilon \models [\,] \sim [\,]$$

$$\text{(Abstr)}\quad \frac{\kappa \models \eta \sim \rho \quad w \sim \omega \quad \ell(\kappa) = \lambda\, id}{(\kappa.1) \models id : w; \eta \sim \omega; \rho}$$

$$\text{(Let)}\quad \frac{\kappa \models \eta \sim \rho \quad w \sim \omega \quad \ell(\kappa) = \textbf{let}\, id}{(\kappa.2) \models id : w; \eta \sim \omega; \rho}$$

$$\text{(Rec)}\quad \frac{\kappa \models \eta \sim \rho \quad \ell(\kappa) = \textbf{let rec}\, id \quad i \in \{1, 2\}}{(\kappa.i) \models id : ((\kappa.1), \circlearrowleft); \eta \sim \rho}$$

$$\text{(Child)}\quad \frac{\kappa \models \eta \sim \rho \quad \ell(\kappa) \notin \{\lambda\, id, \textbf{let}\, id, \textbf{let rec}\, id\}}{(\kappa.i) \models \eta \sim \rho}$$

Abbildung 4.3: Äquivalenzregeln für Werte und Umgebungen

Um nun eine Äquivalenz zwischen den beiden Semantiken beweisen zu können benötigen wir noch einige Lemmata.

Zunächst müssen wir wieder etwas über die beiden *expand*-Funktionen aussagen, damit wir dies beim Lemma über die *lookup*-Funktionen nutzen können.

Da diese wieder beide im Prinzip das gleiche machen, sollten sie natürlich auch für die äquivalente Eingaben die äquivalente Ausgaben generieren.

Lemma 4.1 *Sei $\kappa \in K$, $\eta \in Env$ und $\rho \in Env_{nameless}$.*

Wenn $\kappa \models \eta \sim \rho$, dann $expand(\eta, \kappa) \sim expand(\rho, \kappa)$.

Beweis: Einfache Induktion über Herleitung von $\kappa \models \eta \sim \rho$. □

Nun sollte dies natürlich auch für die beiden *lookup*-Funktionen gelten. Hier setzen wir noch als Voraussetzung das $slookup(\Gamma_\kappa, id)$ einen Index $\imath$ liefert hinzu, damit wir uns nur um die Fälle kümmern müssen, in denen auch ein Eintrag für den Identifier *id* existiert.

Lemma 4.2 *Sei $\kappa \in K$, $\eta \in Env$ und $\rho \in Env_{nameless}$.*

Wenn $\kappa \models \eta \sim \rho$ und $slookup(\Gamma_\kappa, id) = \imath$, dann gilt $lookup(\eta, id) \sim lookup(\rho, \imath)$.

Beweis: Induktion über die Herleitung von $\kappa \models \eta \sim \rho$.

- (EMPTY) Klar, denn wenn $\kappa = \epsilon$, ist $\Gamma_\kappa = [\,]$.
- (CHILD) Es gilt $(\kappa'.i) \models \eta \sim \rho$, sowie die Prämisse $\kappa' \models \eta \sim \rho$. Da $\Gamma_{\kappa'} = \Gamma_{\kappa'.i}$ ist auf $\kappa' \models \eta \sim \rho$ I.V. anwendbar und es gilt $lookup(\eta, id) \sim lookup(\rho, \imath)$.
- (ABSTR) Dann gilt $(\kappa'.1) \models id' : w; \eta' \sim \omega; \rho'$, sowie die folgenden Prämissen:
  1.) $\kappa' \models \eta' \sim \rho'$
  2.) $w \sim \omega$
  3.) $\ell(\kappa') = \lambda\, id$

  Wegen 3. gilt $\Gamma_{\kappa'.1} = id; \Gamma_{\kappa'}$
  - $id = id'$: Dann gilt $slookup(id'; \Gamma_{\kappa'}, id) = 0$, sowie $lookup(\omega; \rho', 0) = \omega$ und $lookup(id' : w; \eta', id) = w$. Dann folgt mit 2. $lookup(id' : w; \eta', id) \sim lookup(\omega; \rho', \imath)$, also $lookup(\eta, id) \sim lookup(\rho, \imath)$.
  - $id \neq id'$: Dann gilt $slookup(id'; \Gamma_{\kappa'}, id) = \imath = inc(slookup(\Gamma_{\kappa'}, id))$. Mit $\imath' = slookup(\Gamma_{\kappa'}, id)$ folgt mit I.V. $lookup(\eta', id) \sim lookup(\rho', \imath')$. Außerdem gilt
    $lookup(id' : w; \eta', id) = lookup(\eta', id)$.
    * $\imath = d + 1$: $lookup(\omega; \rho', \imath) = lookup(\rho', \imath')$, dann gilt $lookup(id' : w; \eta', id) \sim lookup(\omega; \rho', \imath)$, also $lookup(\eta, id) \sim lookup(\rho, \imath)$.
    * $\imath = (d+1, \kappa'')$, dann gilt $lookup(\omega; \rho', \imath) = lookup(\omega; \rho', (d+1, \kappa'')) = lookup(\rho', (d, \kappa'')) = lookup(\rho', \imath')$.
      Damit gilt insgesamt $lookup(\eta, id) \sim lookup(\rho, \imath)$.
- (LET) äquivalent.
- (REC) Dann gilt $(\kappa'.i) \models id' : (\kappa'.1), \circlearrowleft; \eta' \sim \rho$, sowie die Prämissen
  1.) $\kappa' \models \eta' \sim \rho$
  2.) $\ell(\kappa') = \textbf{let rec}\, id$

  Wegen 2. gilt $\Gamma_{\kappa'.i} = id : (\kappa'.1); \Gamma_{\kappa'}$.
  - $id = id'$: Dann gilt $slookup(id' : (\kappa'.1); \Gamma_{\kappa'}, id) = (0, (\kappa'.1))$ und $lookup(\rho, (0, (\kappa'.1))) = expand(\rho, (\kappa'.1))$, sowie $lookup(id' : w; \eta', id) = expand(id' : (\kappa'.1), \circlearrowleft; \eta', (\kappa'.1))$.
    Damit gilt wegen Lemma 4.1: $expand(id' : (\kappa'.1), \circlearrowleft; \eta', (\kappa'.1)) \sim expand(\rho, (\kappa'.1))$ und daher $lookup(id' : w; \eta', id) \sim lookup(\rho, (0, (\kappa'.1)))$, also $lookup(\eta, id) \sim lookup(\rho, \imath)$.
  - $id \neq id'$: Dann gilt $slookup(id' : (\kappa'.1); \Gamma_{\kappa'}, id) = slookup(\Gamma_{\kappa'}, id) = \imath$. Nach I.V. gilt dann $lookup(\eta', id) \sim lookup(\rho, \imath)$.
    Da $lookup(id' : (\kappa'.1), \circlearrowleft; \eta', id) = lookup(\eta', id)$ gilt, ergibt sich insgesamt $lookup(\eta, id) \sim lookup(\rho, \imath)$. □

Jetzt brauchen wir noch ein kleines Lemma, welches uns einen Rückschluss über den Index liefert. Wenn in der Umgebung aus der Umgebungssemantik ein Eintrag für *id* gefunden wurde, dann muss in der namenlosen Semantik auch ein Index für das *id* zu finden sein.

Lemma 4.3 *Sei $\kappa \in K$, $\eta \in Env$ und $\rho \in Env_{nameless}$.*

Wenn $\kappa \models \eta \sim \rho$ und $lookup(\eta, id) = w$, dann existiert $\imath$ mit $slookup(\Gamma_\kappa, id) = \imath$.

Beweis: Trivial, da für beide der Eintrag existieren muss. □

Jetzt sind wir so weit, dass wir die Äquivalenz mit folgendem Satz formulieren und beweisen können.

Satz 4.1 (Äquivalenz) *Sei $\kappa \in K$, $\eta \in Env$ und $\rho \in Env_{nameless}$, mit $\kappa \models \eta \sim \rho$. Dann gilt:*

(a) *Wenn $(\kappa, \eta) \Downarrow w$, dann existiert $\omega \in \Omega$ mit $(\kappa, \rho) \Downarrow \omega$ und $w \sim \omega$.*

(b) *Wenn $(\kappa, \rho) \Downarrow \omega$, dann existiert $w \in W_{node}$ mit $(\kappa, \eta) \Downarrow w$ und $w \sim \omega$.*

Beweis:

(a) Induktion über die Herleitung von $(\kappa, \eta) \Downarrow w$ (nur die interessanten Fälle).

- (Id) Es gilt $(\kappa, \eta) \Downarrow w$ mit folgenden Prämissen:
  - $\ell(\kappa) = id$
  - $lookup(\eta, id) = w$

  Wegen Lemma 4.3 gilt $slookup(\Gamma_\kappa, id) = \imath_\kappa$. Dann folgt mit Lemma 4.2 $lookup(\eta, id) \sim lookup(\rho, \imath_\kappa)$ und mit $lookup(\rho, \imath_\kappa) = \omega$, $w \sim \omega$. Damit sind alle Prämissen für (Id) erfüllt und es gilt: $(\kappa, \rho) \Downarrow \omega$.
- (Beta-V) Es gilt $(\kappa, \eta) \Downarrow w$ mit folgenden Prämissen:

  1.) $\ell(\kappa) = App$
  2.) $((\kappa.1), \eta) \Downarrow (\kappa', \eta')$
  3.) $\ell(\kappa') = \lambda\, id'$
  4.) $((\kappa.2), \eta) \Downarrow w'$
  5.) $((\kappa'.1), id' : w'; \eta') \Downarrow w$

  Wegen 1. kann man auf $\kappa \models \eta \sim \rho$ (Child) anwenden und es gilt $\kappa.1 \models \eta \sim \rho$, sowie $\kappa.2 \models \eta \sim \rho$. Damit kann man auf 2. I.V. anwenden und es gilt

  6.) $((\kappa.1), \rho) \Downarrow (\kappa', \rho')$
  7.) $(\kappa', \eta') \sim (\kappa', \rho')$

  Auf 4. kann man I.V. anwenden:

  8.) $((\kappa.2), \rho) \Downarrow \omega'$
  9.) $w' \sim \omega'$

7. kann nur mit (Closure) hergeleitet worden sein und daher gilt:

10.) $\kappa' \models \eta' \sim \rho'$

Da mit 3., 9. und 10. alle Prämissen für die Regel (Abstr) erfüllt sind, folgt:

11.) $(\kappa'.1) \models id' : w'; \eta' \sim \omega'; \rho'$

Damit kann man dann auf 5. I.V. anwenden und es gilt:

12.) $((\kappa'.1), \omega'; \rho') \Downarrow \omega$

13.) $w \sim \omega$

Insgesamt sind damit alle Prämissen für (Beta-V) erfüllt (1., 3., 6., 8. und 12.) und es gilt $(\kappa, \rho) \Downarrow \omega$ mit $w \sim \omega$.

- (Let) Es gilt $(\kappa, \eta) \Downarrow w$ mit folgenden Prämissen:

  1.) $\ell(\kappa) = \textbf{let}\, id$

  2.) $((\kappa.1), \eta) \Downarrow w'$

  3.) $((\kappa.2), id : w'; \eta) \Downarrow w$

  Wegen 1. kann man auf $\kappa \models \eta \sim \rho$ (Child) anwenden und es gilt $\kappa.1 \models \eta \sim \rho$. Daher kann man auf 2. I.V. anwenden und es gilt:

  4.) $((\kappa.1), \rho) \Downarrow \omega'$

  5.) $w' \sim \omega'$

  Da $\kappa \models \eta \sim \rho$, sowie 1. und 5. gilt, kann (Let) angewendet werden:

  6.) $(\kappa.2) \models id : w'; \eta \sim \omega'; \rho$

  Damit kann auf 3. I.V. angewendet werden und es folgt:

  7.) $((\kappa.2), \omega'; \rho) \Downarrow \omega$

  8.) $w \sim \omega$

  Mit 1., 4. und 7. sind alle Prämissen für (Let) erfüllt und es gilt $(\kappa, \rho) \Downarrow \omega$ mit $w \sim \omega$.

- (Let-Rec) Es gilt $(\kappa, \eta) \Downarrow w$ mit folgenden Prämissen:

  1.) $\ell(\kappa) = \textbf{let rec}\, id$

  2.) $((\kappa.2), id : ((\kappa.1), \circlearrowleft); \eta) \Downarrow w$

  Mit (Rec) folgt:

  3.) $\kappa.i \models id : ((\kappa.1), \circlearrowleft); \eta \sim \rho$

  Dann kann man auf 2. I.V. anwenden und es gilt:

  4.) $((\kappa.2), \rho) \Downarrow \omega$

  5.) $w \sim \omega$

  Insgesamt sind damit alle Prämissen für (Let-Rec) erfüllt (1. und 3.) und es gilt $(\kappa, \rho) \Downarrow \omega$ mit $w \sim \omega$.

(b) Induktion über die Herleitung von $(\kappa, \rho) \Downarrow \omega$. (nur die interessanten Fälle)

- (Id) Es gilt $(\kappa, \rho) \Downarrow \omega$ mit folgenden Prämissen:
  - $\ell(\kappa) = id$
  - $lookup(\rho, \imath_\kappa) = \omega$

Da $\iota_\kappa = \mathit{slookup}(\Gamma_\kappa, id)$ existiert, folgt mit Lemma 4.2 $\mathit{lookup}(\eta, id) \sim \mathit{lookup}(\rho, \iota_\kappa)$, also, da $\mathit{lookup}(\eta, id) = w$, $w \sim \omega$. Damit sind alle Prämissen von (Id) erfüllt und es gilt $(\kappa, \eta) \Downarrow w$ mit $w \sim \omega$.

- (Beta-V) Es gilt $(\kappa, \rho) \Downarrow \omega$ mit folgenden Prämissen:

  1.) $\ell(\kappa) = App$
  2.) $((\kappa.1), \rho) \Downarrow (\kappa', \rho')$
  3.) $\ell(\kappa') = \lambda\, id'$
  4.) $((\kappa.2), \rho) \Downarrow \omega'$
  5.) $((\kappa'.1), \omega'; \rho') \Downarrow \omega$

  Wegen 1. kann man auf $\kappa \models \eta \sim \rho$ (Child) anwenden und es gilt $\kappa.1 \models \eta \sim \rho$, sowie $\kappa.2 \models \eta \sim \rho$. Damit kann man auf 2. I.V. anwenden und es gilt

  6.) $((\kappa.1), \eta) \Downarrow (\kappa', \eta')$
  7.) $(\kappa', \eta') \sim (\kappa', \rho')$

  Auf 4. kann man I.V. anwenden:

  8.) $((\kappa.2), \eta) \Downarrow w'$
  9.) $w' \sim \omega'$

  7. kann nur mit (Closure) hergeleitet worden sein und daher gilt:

  10.) $\kappa' \models \eta' \sim \rho'$

  Da mit 3., 9. und 10. alle Prämissen für die Regel (Abstr) erfüllt sind, folgt:

  11.) $(\kappa'.1) \models id' : w'; \eta' \sim \omega'; \rho'$

  Damit kann man dann auf 5. I.V. anwenden und es gilt:

  12.) $((\kappa'.1), id' : w'; \eta') \Downarrow w$
  13.) $w \sim \omega$

  Insgesamt sind damit alle Prämissen für (Beta-V) erfüllt (1., 3., 6., 8. und 12.) und es gilt $(\kappa, \eta) \Downarrow w$ mit $w \sim \omega$.

- (Let) Es gilt $(\kappa, \rho) \Downarrow \omega$ mit folgenden Prämissen:

  1.) $\ell(\kappa) = \mathbf{let}\ id$
  2.) $((\kappa.1), \rho) \Downarrow \omega'$
  3.) $((\kappa.2), \omega'; \rho) \Downarrow \omega$

  Wegen 1. kann man auf $\kappa \models \eta \sim \rho$ (Child) anwenden und es gilt $\kappa.1 \models \eta \sim \rho$. Daher kann man auf 2. I.V. anwenden und es gilt:

  4.) $((\kappa.1), \eta) \Downarrow w'$
  5.) $w' \sim \omega'$

  Da $\kappa \models \eta \sim \rho$, sowie 1. und 5. gilt, kann (Let) angewendet werden:

  6.) $(\kappa.2) \models id : w'; \eta \sim \omega'; \rho$

  Damit kann auf 3. I.V. angewendet werden und es folgt:

  7.) $((\kappa.2), id : w'; \eta) \Downarrow w$
  8.) $w \sim \omega$

Mit 1., 4. und 7. sind alle Prämissen für (LET) erfüllt und es gilt $(\kappa, \eta) \Downarrow w$ mit $w \sim \omega$.

- (LET-REC) Es gilt $(\kappa, \rho) \Downarrow \omega$ mit folgenden Prämissen:
  1.) $\ell(\kappa) = \textbf{let rec}\, id$
  2.) $((\kappa.2), \rho) \Downarrow \omega$

  Wegen $\kappa \models \eta \sim \rho$ und 1. kann man (REC) anwenden und es folgt:
  3.) $\kappa.i \models id : ((\kappa.1), \circlearrowleft); \eta \sim \rho$

  Dann kann man auf 2. I.V. anwenden und es gilt:
  4.) $((\kappa.2), id : ((\kappa.1), \circlearrowleft); \eta) \Downarrow w$
  5.) $w \sim \omega$

  Insgesamt sind damit alle Prämissen für (LET-REC) erfüllt (1. und 3.) und es gilt $(\kappa, \eta) \Downarrow w$ mit $w \sim \omega$. □

4.2 Namenlose small step Semantik

Nachdem wir jetzt eine big step Semantik ohne Namen haben, wollen wir wieder eine dazu äquivalente small step Semantik angeben. Dazu müssen wir einige Anpassungen an der small step Semantik mit Syntaxbäumen vornehmen. Zunächst müssen die Stackeinträge angepasst werden, so dass diese die namenlosen Umgebungen und Werte anstelle derer mit Namen verwendet.

Wir werden hier das in [ASU99] vorgestellte Prinzip der Controllinks aufgreifen. Der Controllink ist diese abgespeicherte Umgebung. Dieser wird gesichert, damit man die aktuelle Umgebung nach dem Zurückkehren aus einer Funktion direkt wiederfindet, ohne erst auf dem Stack suchen zu müssen. Wir haben also schon die ganze Zeit mit den Controllinks gearbeitet, aber da wir im nächsten Kapitel die Umgebungen durch Pointer ersetzen wollen, müssen wir nun eine Unterscheidung zwischen einer neuen und einer gesicherten Umgebung machen. Um diesen Unterschied deutlich zu machen, führen wir eine Markierung $\uparrow$ ein, welche vor dem Controllink steht.

Definition 4.6 *Die Menge $Entry_{nameless}$ aller namenlosen Stackeinträge ξ ist wie folgt definiert:*

$$\begin{array}{rcll} \xi & ::= & \kappa & \textit{(Rücksprungadresse)} \\ & | & \omega & \textit{((Zwischen-) Ergebnis)} \\ & | & \rho & \textit{(Umgebung)} \\ & | & \uparrow \rho & \textit{(Controllink)} \end{array}$$

Die Menge $Stack_{nameless}$ aller namenlosen Stacks S ist wie folgt definiert:

$$\begin{array}{rcl} S & ::= & [\,] \\ & | & \xi; S \end{array}$$

Eine Konfiguration sieht dann natürlich wieder fast genau so aus wie vorher, außer, dass hier die neuen Umgebungen und Stacks verwendet werden.

Definition 4.7 (Konfiguration) *Eine Konfiguration δ in der namenlosen small step Semantik ist ein Tripel der Form (pos, ρ, S) mit $pos \in Pos$, $\rho \in Env_{nameless}$ und $S \in Stack_{nameless}$.*

Bei der Definition der Regeln geht man nun genau wie bei den Regeln der small step Semantik mit Namen vor. Bei (ID) wird das neue *lookup* verwendet und bei (BETA-V) und (LET-EXEC) werden die Werte logischerweise ohne Identifier eingetragen. Auch hier wird wieder, auch wenn es nicht notwendig wäre, bei (LET-EXEC) der Controllink gesichert.

Im Falle von (LET-REC) wird wie bei der big step Semantik auch, nichts eingetragen, da dies alles im *lookup* beinhaltet ist.

Definition 4.8 (Small step) *Ein small step in der namenlosen small step Semantik ist eine Formel der Gestalt $\delta \rightarrow \delta'$. Ein derartiger small step heißt gültig, wenn er sich mit einer der in Abbildung 4.4 beschriebenen Regeln herleiten lässt.*

4.2.1 Zusammenhang zwischen namenloser big step und small step Semantik

Auch hier können wir wieder Korrektheit und Vollständigkeit der small step Semantik formulieren und analog zur Umgebungssemantik mit Namen beweisen.

Satz 4.2 (Vollständigkeit der namenlosen small step Semantik)
Sei $\kappa \in K$, $S \in Stack_{nameless}$ und $\rho \in Env_{nameless}$.

Wenn $(\kappa, \rho) \Downarrow \omega$, dann $(\cdot\kappa, \rho, S) \xrightarrow{*} (\kappa\cdot, \rho, \omega; S)$.

Beweis: Induktion über die Herleitung von $(\kappa, \rho) \Downarrow \omega$.
Analog zum Beweis von Satz 3.1 □

Satz 4.3 (Korrektheit der namenlosen small step Semantik)
Sei $\kappa \in K$, $S' \in Stack_{nameless}$ und $\rho_1 \in Env_{nameless}$, sowie $m, n \in \mathbb{N}$.
Wenn $(\cdot\epsilon, \rho_1, S') \xrightarrow{n} (\kappa\cdot, \rho, \omega; S)$, dann existiert $m < n$ mit $(\cdot\epsilon, \rho_1, S') \xrightarrow{m} (\cdot\kappa, \rho, S)$ und $(\kappa, \rho) \Downarrow \omega$.

Beweis: Induktion über n und Fallunterscheidung nach der Form von $\ell(\kappa)$.
Analog zum Beweis von Satz 3.2 □

Auch dieses mal legen wir fest, dass wir ab jetzt nur noch die namenlosen Semantiken betrachten und daher immer wenn im Folgenden von Umgebungssemantik, big step oder small step Semantik die Rede ist, die in diesem Kapitel definierten gemeint sind.

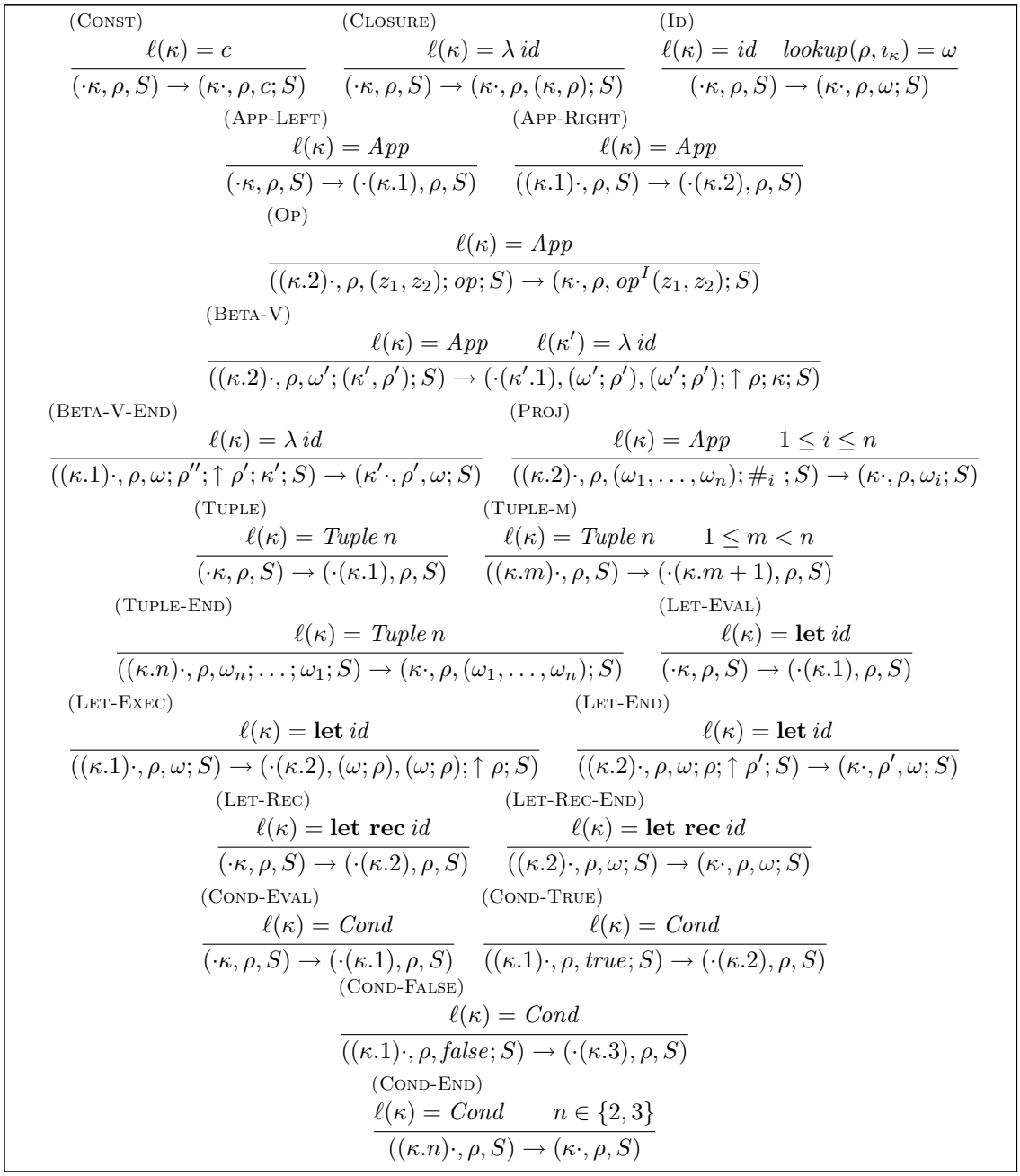

Abbildung 4.4: Small step Regeln für die namenlose Umgebungssemantik

5 Stacksemantik

Beim Übergang von der big step zur small step Semantik in den Kapiteln 3 und 4 wurde ein Stack S eingeführt. Dabei handelt es sich um einen "abstrakten" Stack, der (wie der Stack der SECD-Maschine[Lan64]) ganze Umgebungen und Closures als einzelne Einträge zulässt.

Dieser abstrakte Stack S soll nun durch einen realistische(re)n Stack σ ersetzt werden, der nur noch Einträge begrenzter Größe erlaubt.[1] Dazu gehen wir wie folgt vor:

Zu Beginn wird die leere Umgebung $[\,]$ in σ gelegt. Jede nicht leere Umgebung $\rho = \omega; \rho'$ in S wird in σ durch einen neuartigen Eintrag ersetzt, den wir als *Frame* bezeichnen. Ein solcher Frame enthält den Wert ω und einen Zeiger auf die Umgebung ρ', die zum Zeitpunkt der Aufnahme von ρ bereits (irgendwo weiter unten) im Stack liegt. Diesen Zeiger bezeichnen wir als *Accesslink*.

Zur Darstellung von Zeigern verwenden wir Zahlen $i \in \mathbb{N}$: i zeigt auf den i-ten Eintrag "von unten" in σ.

Analog zu den in S eingetragenen Umgebungen ρ werden auch Closures (κ, ρ) und die Einträge der Form $\uparrow \rho$ durch Einträge begrenzter Größe in σ dargestellt: (κ, ρ) wird durch (κ, i) ersetzt und $\uparrow \rho$ durch i, wobei i jeweils auf ρ zeigt. Im letzten Fall bezeichnen wir i als *Controllink*.

Beispiel 5.1 Für einen Stack $S = (\omega_2; (\omega_1; [\,])); \uparrow (\omega_1; [\,]); \kappa_2; (\omega_1; [\,]); \uparrow [\,]; \kappa_1; [\,]; \varepsilon$, sieht der realistischere Stack σ wie folgt aus: $(\omega_2; 4); 4; \kappa_2; (\omega_1; 1); 1; \kappa_1; [\,]; \varepsilon$.

Die Begriffe *Accesslink* und *Controllink* wurden aus der Compilerbauliteratur übernommen [ASU99, WM92]. Man beachte, dass die hier verwendeten Frames nur einen Wert und einen Pointer enthalten. In der Literatur enthalten *Stackframes* zusätzlich den Controllink und die Rücksprungadresse. Außerdem können dort mehrere Werte gleichzeitig, also in einem Stackframe, aufgenommen werden.

Eine Stack-Verwaltung der oben beschriebenen Art geht natürlich nicht gut, wenn wir unsere funktionale Programmiersprache in vollem Umfang beibehalten. Selbst dann nicht, wenn wir uns auf die wohlgetypten Programme des üblichen Typsystems beschränken (Abbildung 5.5).

Das *Hinzufügen* einer neuen Umgebung $\rho = \omega; \rho'$ zum abstrakten Stack S kann zwar stets korrekt durch einen Frame simuliert werden, weil ρ' zu diesem Zeitpunkt schon in S enthalten ist. Aber das *Entfernen* von ρ durch die Regeln (Beta-V-End)

[1] Bei Tupeln weichen wir von diesem Prinzip ab. Sie sollten eigentlich auf mehrere Einträge im Stack verteilt werden, aber um dies vernünftig umzusetzen, müsste man das Typsystem ins Spiel bringen, worauf wir hier verzichten.

oder (Let-End) lässt sich im allgemeinen *nicht* durch Entfernen des entsprechenden Frames simulieren.

Wenn nämlich über der Umgebung ρ eine Closure (κ, ρ) liegt, dann entsteht durch (Beta-V-End) oder (Let-End) ein Stack S, der immer noch die Closure (κ, ρ) enthält, aber *nicht mehr* die Umgebung ρ (als eigenständigen Eintrag). Für die entsprechende Closure (κ, i) in σ würde dies bedeuten, dass der Frame für ρ, auf den i ja zeigen soll, gar nicht mehr in σ enthalten ist.

Da (Beta-V-End) und (Let-End) jeweils den letzten Schritt in der small step Simulation der big step Regeln (Beta-V) und (LET) darstellen, kann die oben geschilderte Situation natürlich nur auftreten, wenn aus dem Rumpf einer λ-Abstraktion oder eines **let**-Ausdrucks eine Funktion zurückgeliefert wird. In diesem Fall kann der durch λ oder **let** eingeführte Name seinen Gültigkeitsbereich überleben, nämlich dann, wenn er frei in der Funktion vorkommt. Dadurch wird die sogenannte *Stack-Disziplin* zerstört.

Beispiel 5.2 Betrachten wir den Ausdruck (**let** $x = 1$ **in** $(\lambda y.\ +\ (x, y)))\, 2$ in der namenlosen small step Semantik:

$$
\begin{array}{ll}
 & (\cdot((\textbf{let}\ x = 1\ \textbf{in}\ (\lambda y.\ +\ (x, y)))\, 2), [\,], [\,]; \varepsilon) \\
\xrightarrow{\text{(App-Left)}} & (\cdot(\textbf{let}\ x = 1\ \textbf{in}\ (\lambda y.\ +\ (x, y))), [\,], [\,]; \varepsilon) \\
\xrightarrow{\text{(Let-Eval)}} & (\cdot 1, [\,], [\,]; \varepsilon) \\
\xrightarrow{\text{(Const)}} & (1\cdot, [\,], 1; [\,]; \varepsilon) \\
\xrightarrow{\text{(Let-Exec)}} & (\cdot(\lambda y.\ +\ (x, y)), (1; [\,]), (1; [\,]); \uparrow [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Closure)}} & ((\lambda y.\ +\ (x, y))\cdot, (1; [\,]), (\lambda y.\ +\ (x, y); (1; [\,])); (1; [\,]); \uparrow [\,]; [\,]; \varepsilon) \\
\xrightarrow{\text{(Let-End)}} & ((\textbf{let}\ x = 1\ \textbf{in}\ (\lambda y.\ +\ (x, y)))\cdot, [\,], (\lambda y.\ +\ (x, y); (1; [\,])); [\,]; \varepsilon) \\
\xrightarrow{\text{(App-Right)}} & (\cdot 2, [\,], (\lambda y.\ +\ (x, y); (1; [\,])); [\,]; \varepsilon) \\
\xrightarrow{\text{(Const)}} & (2\cdot, [\,], 2; (\lambda y.\ +\ (x, y); (1; [\,])); \varepsilon) \\
\xrightarrow{\text{(Beta-V)}} & (\cdot(\lambda y.\ +\ (x, y)), (2; 1; [\,]), (2; 1; [\,]); \uparrow [\,]; (\textbf{let}\ x = 1\ \textbf{in}\ (\lambda y.\ +\ (x, y)))\, 2; \\
 & [\,]; \varepsilon) \\
\ldots &
\end{array}
$$

Im Schritt (Let-End) wird die Umgebung $(1; [\,])$ vom Stack enfernt. Diese wird aber in der Closure $(\lambda y.\ +\ (x, y); (1; [\,]))$ noch gebraucht. Wenn man hier Pointer einsetzen würde, so würden diese auf die falschen Einträge zeigen.

Aus den bisherigen Überlegungen folgt, dass wir

(a) entweder zusätzlich zum Stack σ einen *heap* einführen

(b) oder unsere Programmiersprache einschränken müssen.

Wir entscheiden uns für (b), und zwar wie folgt:

In diesem Kapitel schränken wir einfach die Regeln (BETA-V) und (LET) - und damit natürlich auch die entsprechenden small step Regeln - so ein, dass sie keine Funktionen, sondern nur noch Konstanten als Resultate liefern können. Für diese

eingeschränkte Umgebungssemantik können wir die Äquivalenz zu einer Stacksemantik beweisen.

Die Einschränkung der semantischen Regeln hat natürlich den Nachteil, dass die Typsicherheit verloren geht, d.h. dass auch wohlgetypte Programme (wie das Programm aus Bsp. 5.2) stecken bleiben können. Deshalb definieren wir in Abschnitt 5.4 ein eingeschränktes Typsystem, das verhindert, dass Funktionen als Resultat aus einem inneren Block zurückgeliefert werden.

5.1 Einschränkung der Semantiken

Um die Regeln einzuschränken, benötigen wir eine andere Menge von Konstanten. Diese beinhaltet zusätzlich Tupel von Konstanten.

Definition 5.1 *Die Menge Const aller* Konstanten c *ist definiert durch die folgende kontextfreie Grammatik.*

$$
\begin{array}{rcll}
c & ::= & b & \textit{(bool'scher Wert)} \\
 & | & z & \textit{(ganze Zahl)} \\
 & | & op & \textit{(Operator)} \\
 & | & \#_i & \textit{(Projektion mit } i \in \mathbb{N}\textit{)} \\
 & | & (c_1, \ldots, c_n) & \textit{(Tupel von Konstanten)}
\end{array}
$$

Hier entsteht dann eine Überschneidung zwischen den Konstanten und den Ausdrücken. Da dies aber keinen großen Schaden anrichtet, soll es uns nicht weiter stören.

Dann können wir die Einschränkung zunächst für die big step Semantik und danach für die small step Semantik angeben.

$$
\frac{(e_1, \eta) \Downarrow (\lambda id'.\, e', \eta') \qquad (e_2, \eta) \Downarrow w' \qquad (e', id' : w'; \eta') \Downarrow c}{(e_1\, e_2, \eta) \Downarrow c}\ (\textsc{Beta-V})
\qquad
\frac{(e_1, \eta) \Downarrow w \qquad (e_2, id : w; \eta) \Downarrow c}{(\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2, \eta) \Downarrow c}\ (\textsc{Let})
$$

Abbildung 5.1: Einschränkung der big step Regeln für die Umgebungssemantik

$$
\frac{\ell(\kappa) = \lambda\, id}{((\kappa.1)\cdot, \rho, c; \rho''; \uparrow \rho'; \kappa'; S) \to (\kappa'\cdot, \rho', c; S)}\ (\textsc{Beta-V-End})
\qquad
\frac{\ell(\kappa) = \mathbf{let}\, id}{((\kappa.2)\cdot, \rho, c; \rho; \uparrow \rho'; S) \to (\kappa\cdot, \rho', c; S)}\ (\textsc{Let-End})
$$

Abbildung 5.2: Einschränkung der small step Regeln für die namenlose Umgebungssemantik

5.2 Definition der Stacksemantik

Die Überlegungen zu Beginn des Kapitels führen zu folgenden Definitionen:

Definition 5.2 *Die Menge Val_{stack} aller Werte ν ist wie folgt definiert:*

$$\begin{array}{rcll} \nu & ::= & c & \textit{(Konstante)} \\ & | & (\kappa, i) & \textit{(Closure mit } i \in \mathbb{N}\textit{)} \\ & | & (\nu_1, \ldots, \nu_n) & \textit{(Tupel mit } n > 1\textit{)} \end{array}$$

Die Menge $Entry_{Stack}$ aller Stackeinträge ξ ist wie folgt definiert:

$$\begin{array}{rcll} \xi & ::= & i & \textit{(}\text{Controllink } \textit{mit } i \in \mathbb{N}\textit{)} \\ & | & \kappa & \textit{(}\text{Rücksprungadresse}\textit{)} \\ & | & \nu & \textit{((Zwischen-) Ergebnis)} \\ & | & [\nu, i] & \textit{(}\text{Frame } \textit{mit } i \in \mathbb{N}\textit{)} \end{array}$$

Die Menge Σ aller Stacks σ ist wie folgt definiert:

$$\begin{array}{rcl} \sigma & ::= & [\,] \\ & | & \xi; \sigma \end{array}$$

Die Funktion $size(\sigma)$ liefert die Größe von σ.
Für einen Stack $\sigma = [\zeta_n; \ldots; \zeta_1]$ ist $\sigma[i] = [\zeta_i; \ldots; \zeta_1]$

Bei den Closures steht nun anstelle der Umgebung der Pointer i welcher auf den zur Closure passenden Frame zeigt. Auf dem Stack können nun keine Umgebungen mehr liegen, sondern lediglich Frames, welche aus dem Wert ν, welcher auf dem Stack abgelegt wurde, und dem Accesslink i bestehen. Dieser Accesslink zeigt im Prinzip auf den Rest der Umgebung, in dem er auf den nächsten Frame zeigt. Wir werden diesen bei der Definition von *lookup* genauer betrachten.

Als weiteren Stackeintrag lassen wir nur Pointer i zu, in Form eines Controllinks. Dieser wird verwendet, da wir nur einen Stack zur Verwaltung aller Einträge benutzen. Er wird gebraucht, wenn man eine Funktion verlässt, um wieder auf den alten Pointer (früher Umgebungszeiger) zu kommen, da man den ersten Frame auf dem Stack sonst erst suchen müsste, da andere Einträge über dem Frame liegen können. Daher wird der Pointer vor dem Eintritt in eine Funktion gesichert und beim Austritt wieder geladen. Wir werden dies bei der Definition der small step Regeln sehen.

Des weiteren müssen *lookup* und *expand* angepasst werden:

Definition 5.3 *Die partielle Funktion $lookup : \Sigma \times \mathbb{N} \times Index \to Val_{stack}$ ist definiert durch:*

$$lookup(\sigma, i, \imath) = \begin{cases} expand(i, \kappa) & \textit{falls } \imath = (0, \kappa) \\ \nu & \textit{falls } \imath = 0 \textit{ und } \sigma[i] = [\nu, j]; \sigma' \\ lookup(\sigma', j, d) & \textit{falls } \imath = d + 1 \textit{ und } \sigma[i] = [\nu, j]; \sigma' \\ lookup(\sigma', j, (d, \kappa)) & \textit{falls } \imath = (d + 1, \kappa) \textit{ und } \sigma[i] = [\nu, j]; \sigma' \\ undef. & sonst \end{cases}$$

Die partielle Funktion $expand : \mathbb{N} \times Node \to Val_{stack}$ *ist induktiv definiert durch:*

$$expand(i, \kappa) \quad = \quad \begin{cases} c & \text{falls } \ell(\kappa) = c \in Const \\ (\kappa, i) & \text{falls } \ell(\kappa) = \lambda\, id \\ (expand(i, \kappa.1), \ldots, expand(i, \kappa.n)) & \text{falls } \ell(\kappa) = Tuple\, n \\ undef. & sonst \end{cases}$$

Wie man an der Definition von *lookup* sieht, zählen wir wieder den Index runter. Allerdings folgen wir den Accesslinks und gehen über diese in den Stack hinein. Wenn der Index 0 ist, liefert *lookup* den Wert im aktuellen Frame und wenn der Index $(0, \kappa)$ ist, liefert *lookup* den expandierten Knoten.

Als nächstes wird wieder eine neue Art der Konfiguration definiert, auf denen die small steps dann arbeiten werden.

Definition 5.4 (Konfiguration) *Eine Konfiguration* ζ *in der Stacksemantik ist ein Tripel der Form* (pos, i, σ)*, wobei* $pos \in Pos$ *sowie* $\sigma \in \Sigma$ *und* $i \in \mathbb{N}$*.*

Wie man sieht, steht in einer Konfiguration nun anstelle der Umgebung η, die als Pointer genutzt wurde, ein „realistischerer“ Pointer i. Dieser muss in den Regeln dann natürlich an den gleichen Stellen aktualisiert werden wie die Umgebung auch. Bei (Beta-V) und (Let-Exec) müssen Neue generiert werden und bei (Beta-V-End) und (Let-End) müssen die Alten wieder vom Stack genommen werden.

Auch hier gilt wieder: Bei einem **let**-Ausdruck könnte man sich das Sichern des Controllinks auch sparen, da man diesen bei (Let-End) wieder aus dem Frame entnehmen könnte, aber um es konsistent zur λ-Abstraktion zu halten, wird auch hier der Pointer gesichert.

Definition 5.5 (Small step) *Ein small step in der Stacksemantik ist eine Formel der Gestalt* $\zeta \to \zeta'$*. Ein derartiger small step heißt gültig für ein Gesamtprogramm e mit dem Syntaxbaum* $T_e = (K, \ell)$*, wenn er sich mit einer der Regeln aus Abbildung 5.3 herleiten lässt.*

5.3 Zusammenhang

Nun werden wir, wie in den vergangenen Kapiteln, wieder einen Zusammenhang zwischen der alten und der neuen Semantik herstellen. Dazu brauchen wir wieder eine Äquivalenz zwischen den jeweiligen Konfigurationen. Diese wird dann heruntergebrochen auf Stacks und Stackeinträge.

Definition 5.6 *Sei* $\sigma \in \Sigma$*,* $S \in Stack$*,* $\xi \in Entry_{Stack}$ *und* $\xi' \in Entry_{nameless}$*, sowie* ζ *eine Konfiguration in der Stacksemantik und* δ *eine Konfiguration in der namenlosen small step Semantik. Dann heißt* $\sigma \sim S$*,* $\sigma \models \xi \sim \xi'$ *und* $\zeta \sim \delta$ *gültig, wenn es sich mit den Regeln aus Abbildung 5.4 herleiten lässt.*

$$\text{(Const)}\ \frac{\ell(\kappa) = c}{(\cdot\kappa, i, \sigma) \to (\kappa\cdot, i, c; \sigma)} \qquad \text{(Closure)}\ \frac{\ell(\kappa) = \lambda\, id}{(\cdot\kappa, i, \sigma) \to (\kappa\cdot, i, (\kappa, i); \sigma)} \qquad \text{(Id)}\ \frac{\ell(\kappa) = id \quad lookup(\sigma, i, \iota_\kappa) = \nu}{(\cdot\kappa, i, \sigma) \to (\kappa\cdot, i, \nu; \sigma)}$$

$$\text{(App-Left)}\ \frac{\ell(\kappa) = App}{(\cdot\kappa, i, \sigma) \to (\cdot(\kappa.1), i, \sigma)} \qquad \text{(App-Right)}\ \frac{\ell(\kappa) = App}{((\kappa.1)\cdot, i, \sigma) \to (\cdot(\kappa.2), i, \sigma)}$$

$$\text{(Op)}\ \frac{\ell(\kappa) = App}{((\kappa.2)\cdot, i, (z_1, z_2); op; \sigma) \to (\kappa\cdot, i, op^I(z_1, z_2); \sigma)}$$

$$\text{(Beta-V)}\ \frac{\ell(\kappa) = App \qquad \ell(\kappa') = \lambda\, id}{((\kappa.2)\cdot, i, \nu; (\kappa', i'); \sigma) \to (\cdot(\kappa'.1), size(\sigma) + 3, [\nu, i']; i; \kappa; \sigma)}$$

$$\text{(Beta-V-End)}\ \frac{\ell(\kappa) = \lambda\, id}{((\kappa.1)\cdot, i, c; [\nu', i'']; i'; \kappa'; \sigma) \to (\kappa'\cdot, i', c; \sigma)} \qquad \text{(Proj)}\ \frac{\ell(\kappa) = App \qquad 1 \le j \le n}{((\kappa.2)\cdot, i, (\nu_1, \ldots, \nu_n); \#_j\ ; \sigma) \to (\kappa\cdot, i, \nu_j; \sigma)}$$

$$\text{(Tuple)}\ \frac{\ell(\kappa) = Tuple\, n}{(\cdot\kappa, i, \sigma) \to (\cdot(\kappa.1), i, \sigma)} \qquad \text{(Tuple-m)}\ \frac{\ell(\kappa) = Tuple\, n \qquad 1 \le m < n}{((\kappa.m)\cdot, i, \sigma) \to (\cdot(\kappa.m + 1), i, \sigma)}$$

$$\text{(Tuple-End)}\ \frac{\ell(\kappa) = Tuple\, n}{((\kappa.n)\cdot, i, \nu_n; \ldots; \nu_1; \sigma) \to (\kappa\cdot, i, (\nu_1, \ldots, \nu_n); \sigma)} \qquad \text{(Let-Eval)}\ \frac{\ell(\kappa) = \textbf{let}\ id}{(\cdot\kappa, i, \sigma) \to (\cdot(\kappa.1), i, \sigma)}$$

$$\text{(Let-Exec)}\ \frac{\ell(\kappa) = \textbf{let}\ id}{((\kappa.1)\cdot, i, \nu; \sigma) \to (\cdot(\kappa.2), size(\sigma) + 2, [\nu, i]; i; \sigma)} \qquad \text{(Let-End)}\ \frac{\ell(\kappa) = \textbf{let}\ id}{((\kappa.2)\cdot, i, c; [\nu', i']; i'; \sigma) \to (\kappa\cdot, i', c; \sigma)}$$

$$\text{(Let-Rec)}\ \frac{\ell(\kappa) = \textbf{let rec}\ id}{(\cdot\kappa, i, \sigma) \to (\cdot(\kappa.2), i, \sigma)} \qquad \text{(Let-Rec-End)}\ \frac{\ell(\kappa) = \textbf{let rec}\ id}{((\kappa.2)\cdot, i, \nu; \sigma) \to (\kappa\cdot, i, \nu; \sigma)}$$

$$\text{(Cond-Eval)}\ \frac{\ell(\kappa) = Cond}{(\cdot\kappa, i, \sigma) \to (\cdot(\kappa.1), i, \sigma)} \qquad \text{(Cond-True)}\ \frac{\ell(\kappa) = Cond}{((\kappa.1)\cdot, i, true; \sigma) \to (\cdot(\kappa.2), i, \sigma)} \qquad \text{(Cond-False)}\ \frac{\ell(\kappa) = Cond}{((\kappa.1)\cdot, i, false; \sigma) \to (\cdot(\kappa.3), i, \sigma)}$$

$$\text{(Cond-End)}\ \frac{\ell(\kappa) = Cond \qquad n \in \{2, 3\}}{((\kappa.n)\cdot, i, \sigma) \to (\kappa\cdot, i, \sigma)}$$

Abbildung 5.3: Small step Regeln für die Stacksemantik

$$\text{(Empty)}\quad [\,] \sim [\,]; [\,] \qquad \text{(Entry)}\quad \frac{\sigma \models \xi \sim \xi' \qquad \sigma \sim S}{\xi; \sigma \sim \xi'; S} \qquad \text{(Addr)}\quad \sigma \models \kappa \sim \kappa \qquad \text{(Const)}\quad \sigma \models c \sim c \qquad \text{(Link)}\quad \frac{\sigma[i] \sim \rho; S}{\sigma \models i \sim\!\uparrow \rho}$$

$$\text{(Frame)}\quad \frac{\sigma \models \nu \sim \omega \qquad \sigma \models i \sim\!\uparrow \rho}{\sigma \models [\nu, i] \sim (\omega; \rho)} \qquad \text{(Tuple)}\quad \frac{\sigma \models \nu_i \sim \omega_i \qquad 1 \leq i \leq n}{\sigma \models (\nu_1, \ldots, \nu_n) \sim (\omega_1, \ldots, \omega_n)} \qquad \text{(Closure)}\quad \frac{\sigma \models i \sim\!\uparrow \rho}{\sigma \models (\kappa, i) \sim (\kappa, \rho)}$$

$$\text{(Conf)}\quad \frac{\sigma \models i \sim\!\uparrow \rho \qquad \sigma \sim S}{(\dot{\kappa}, i, \sigma) \sim (\dot{\kappa}, \rho, S)}$$

Abbildung 5.4: Äquivalenzregeln für Stacks, Stackeinträge und Konfigurationen

Bei der Regel (Empty) ist zu beachten, dass der Stack aus der leeren Umgebung gefolgt vom leeren Stack besteht. Dies wird benötigt, da der Pointer in einer Konfiguration immer auf ein Element des Stacks zeigen muss und wenn dieser Pointer auf die leere Umgebung zeigt, so muss diese natürlich auch auf dem Stack enthalten sein.

Bei der Regel (Link) wird in der Prämisse gefordert, dass das ρ am Anfang eines Stacks steht, welcher äquivalent zu dem Teil der Größe i von σ ist. Damit sind dann die beiden Pointer äquivalent bezogen auf σ.

Bei (Frame) kann man aus zwei äquivalenten Werten und zwei äquivalenten Pointern einen Frame auf den beiden Stacks generieren. Bei der Stacksemantik ist dies ein „wirklicher“ Frame und bei der Umgebungssemantik wird dies durch eine neue Umgebung gemacht.

(Conf) zieht das ganze dann hoch auf Konfigurationen, damit man eine einfachere Schreibweise für den Äquivalenzsatz bekommt.

Bevor wir nun allerdings zu diesem Satz kommen, benötigen wir ein paar Lemmata, welche uns bei dem Beweis der Äquivalenz der namenlosen small step Semantik und der Stacksemantik helfen werden.

Zunächst muss man etwas über die Einträge auf den Stacks sagen.

Lemma 5.1 *Seien σ und $\sigma' \in \Sigma$, sowie $\xi \in Entry_{Stack}$ und $\xi' \in Entry_{nameless}$. Wenn $\sigma' = \sigma[i]$ und $\sigma' \models \xi \sim \xi'$, dann gilt auch $\sigma \models \xi \sim \xi'$*

Beweis: Ergibt sich leicht aus den Regeln. □

Dieses Lemma sagt nichts anderes aus, als dass zwei Einträge, die bezogen auf einen Stack äquivalent sind, auch auf einem vergrößerten äquivalent sind.

Außerdem benötigen wir noch eine Aussage über die Äquivalenz von Einträgen im Bezug auf verschiedene Stacks σ.

Lemma 5.2 *Sei $\xi'' \in Entry_{Stack} \setminus \{[\nu, i]\}$. Dann gilt $\sigma \models \xi \sim \xi'$ genau dann, wenn $\xi''; \sigma \models \xi \sim \xi'$.*

Beweis: Ergibt sich leicht aus den Regeln. □

Dieses Lemma besagt nichts anderes, als dass man sowohl Einträge von σ wegnehmen, als auch zu σ hinzufügen kann, ohne das es sich auf die Äquivalenz von Einträgen auswirkt, solange es keine Frames sind.

Zusätzlich benötigen wir wieder ein Lemma, welches uns die Äquivalenz der beiden *expand*s liefert.

Lemma 5.3 *Wenn $\sigma \models i \sim\uparrow \rho$, dann gilt $\sigma \models expand(i, \kappa) \sim expand(\rho, \kappa)$.*

Beweis: Klar, da *expand* wieder nur die Umgebungen, bzw. die Pointer auf diese, auf die Knoten verteilt. □

Nachdem die *expand*s äquivalent sind, müssen nun auch die *lookup*s äquivalent sein, was folgendes Lemma zeigt.

Lemma 5.4 *Wenn $\sigma \models i \sim\uparrow \rho$, dann gilt $\sigma \models lookup(\sigma, i, \imath) \sim lookup(\rho, \imath)$.*

Beweis: Induktion über $\imath$:

- $\imath = (0, \kappa)$: Es gilt $lookup(\sigma, i, \imath) = expand(i, \kappa)$ und $lookup(\rho, \imath) = expand(\rho, \kappa)$. Nach Lemma 5.3 folgt $\sigma \models lookup(\sigma, i, \imath) \sim lookup(\rho, \imath)$.
- $\sigma[i] \neq [\nu, j]; \sigma'$: $\sigma \models i \sim\uparrow \rho$ kann nur mit Regel (Link) hergeleitet worden sein und es gilt $\sigma[i] \sim \rho; S$. Diese kann nur mit Regel (Empty) hergeleitet worden sein, da die Prämisse für (Entry) nicht erfüllt werden kann. Damit gilt $\sigma[i] = [\,]$, sowie $\rho = [\,]$ und daher sind beide lookups undefiniert.
- $\sigma[i] = [\nu, j]; \sigma'$:
  - $\imath = 0$: Da $\sigma \models i \sim\uparrow \rho$ nur mit Regel (Link) hergeleitet werden konnte, gilt die Prämisse $\sigma[i] \sim\uparrow \rho; S$. Es gilt $lookup(\sigma, i, \imath) = \nu$, sowie $[\nu, j]; \sigma' \sim \rho; S$. Dies muss mit Regel (Entry) hergeleitet worden sein und daher gilt auch: $\sigma' \models [\nu, j] \sim \rho$. Dieses muss mit Regel (Frame) hergeleitet worden sein und daher muss ρ von der Form $\omega; \rho'$ sein und es gilt: $\sigma' \models \nu \sim \omega$. Da $lookup(\rho, 0) = lookup(\omega; \rho', 0) = \omega$ folgt mit Lemma 5.1 $\sigma \models \nu \sim \omega$ und damit insgesamt $\sigma \models lookup(\sigma, i, \imath) \sim lookup(\rho, \imath)$.
  - $\imath = d+1$: Es gilt $lookup(\sigma, i, d+1) = lookup(\sigma', j, d)$. $[\nu, j]; \sigma' \sim \rho; S$ kann nur mit (Entry) hergeleitet worden sein und daher gilt: $\sigma' \models [\nu, j] \sim \rho$. Dies kann nur mit Regel (Frame) hergeleitet worden sein, daher muss ρ von der Form $\omega; \rho'$ sein und es gilt $\sigma' \models \nu \sim \omega$, sowie $\sigma' \models j \sim\uparrow \rho'$. Mit I.V. folgt dann $\sigma' \models lookup(\sigma', j, d) \sim lookup(\rho', d)$. Da $lookup(\rho, d+1) = lookup(\rho', d)$ gilt wegen Lemma 5.1 $\sigma \models lookup(\sigma, i, \imath) \sim lookup(\rho, \imath)$.
  - $\imath = (d + 1, \kappa)$: Analog □

Nachdem nun die Vorarbeiten gemacht sind, können wir nun die Äquivalenz von namenloser small step Semantik und Stacksemantik formulieren und mit Hilfe der eben genannten Lemmata beweisen. Hierzu werden natürlich die in Abbildung 5.2 genannten Einschränkungen verwendet.

Satz 5.1 (Äquivalenz) *Seien* $\zeta = (\dot{\kappa}, i, \sigma)$ *und* $\delta = (\dot{\kappa}, \rho, S)$, *mit* $\zeta \sim \delta$.

(a) *Wenn* $\zeta \to \zeta'$, *dann* $\delta \to \delta'$ *und* $\zeta' \sim \delta'$.

(b) *Wenn* $\delta \to \delta'$, *dann* $\zeta \to \zeta'$ *und* $\zeta' \sim \delta'$.

Beweis:

(a) Fallunterscheidung nach der angewendeten Regel.(Nur interessante Fälle)

- (Id): Es gilt $(\cdot\kappa, i, \sigma) \to (\kappa\cdot, i, \nu; \sigma)$ mit folgenden Prämissen:
  1.) $\ell(\kappa) = id$
  2.) $lookup(\sigma, i, \imath_\kappa) = \nu$

  Da $(\cdot\kappa, i, \sigma) \sim (\cdot\kappa, \rho, S)$ nur mit Regel (Conf) hergeleitet werden konnte muss außerdem gelten:
  3.) $\sigma \models i \sim\uparrow \rho$.
  4.) $\sigma \sim S$.

  Dann folgt mit Lemma 5.4
  5.) $\sigma \models lookup(\sigma, i, \imath_\kappa) \sim lookup(\rho, \imath_\kappa)$.

  Wegen 2. muss
  6.) $lookup(\rho, \imath_\kappa) = \omega$

  gelten. Damit sind alle Prämissen für (Id) erfüllt (1. und 6.) und es gilt $(\cdot\kappa, \rho, S) \to (\kappa\cdot, \rho, \omega; S)$. Aus 4. und 5. folgt mit Regel (Entry) $\nu; \sigma \sim \omega; S$. Nun kann man Regel (Conf) anwenden und es gilt $(\kappa\cdot, i, \nu; \sigma) \sim (\kappa\cdot, \rho, \omega; S)$.
- (Beta-V): Es gilt:
  $((\kappa''.2)\cdot, i, \nu; (\kappa', i'); \sigma') \to (\cdot(\kappa'.1), size(\sigma') + 3, [\nu, i']; i; \kappa''; \sigma')$ mit den Prämissen:
  1.) $\ell(\kappa'') = App$
  2.) $\ell(\kappa') = \lambda\, id$

  Außerdem gilt $((\kappa''.2)\cdot, i, \nu; (\kappa', i'); \sigma') \sim ((\kappa''.2)\cdot, \rho, S)$. Dies kann nur mit Regel (Conf) hergeleitet worden sein und deshalb gelten die Prämissen:
  3.) $\nu; (\kappa', i'); \sigma' \models i \sim\uparrow \rho$
  4.) $\nu; (\kappa', i'); \sigma' \sim S$

  Aus 4. folgt, dass $S = \omega; S'$ sein muss, da dies nur mit der Regel (Entry) hergeleitet worden sein kann. Daher gelten auch die folgenden Prämissen:
  5.) $\nu; (\kappa', i'); \sigma' \models \nu \sim \omega$
  6.) $(\kappa', i'); \sigma' \sim S'$

  6. kann wieder nur mit (Entry) hergeleitet worden sein und daher muss $S' = (\kappa', \rho'); S''$ gelten, sowie
  7.) $(\kappa', i'); \sigma' \models (\kappa', i') \sim \kappa', \rho'$
  8.) $\sigma' \sim S''$

Also gilt insgesamt $S = \omega; (\kappa', \rho'); S''$ und damit kann, wegen 1. und 2. auf $((\kappa''.2)\cdot, \rho, \omega; (\kappa', \rho'); S'')$ (Beta-V) angewendet werden und es folgt:

– $((\kappa''.2)\cdot, \rho, \omega; (\kappa', \rho'); S'') \to (\cdot(\kappa'.1), (\omega; \rho'), (\omega, \rho'); \kappa''; \uparrow \rho; S'')$

Zu zeigen bleibt nun:
$(\cdot(\kappa'.1), size(\sigma') + 3, [\nu, i']; i; \kappa''; \sigma') \sim (\cdot(\kappa'.1), (\omega; \rho'), (\omega, \rho'); \uparrow \rho; \kappa''; S'')$
Aus 3. folgt mit Lemma 5.2

9.) $\sigma' \models i \sim\uparrow \rho$

Dann folgt mit (Entry) auf 8. und 9. angewendet

10.) $i; \sigma' \sim\uparrow \rho; S''$

Ebenso folgt mit (Entry) auf 10. angewendet

11.) $i; \kappa''; \sigma' \sim\uparrow \rho; \kappa''; S''$

Aus 7. folgt mit (Closure) und Lemma 5.2 $\sigma' \models i' \sim\uparrow \rho'$. Wiederum mit Lemma 5.2 ergibt sich daraus

12.) $i; \kappa''; \sigma' \models i' \sim\uparrow \rho'$.

Aus 5. folgt durch mehrmaliges Anwenden von Lemma 5.2

13.) $i; \kappa''; \sigma' \models \nu \sim \omega$

Auf 12. und 13. kann man (Frame) anwenden

14.) $i; \kappa''; \sigma' \models [\nu, i'] \sim (\omega, \rho')$

Mit 11. und 14. folgt mit (Entry)

15.) $[\nu, i']; i; \kappa''; \sigma' \sim (\omega, \rho'); \uparrow \rho; \kappa''; S''$

Es gilt $size([\nu, i']; i; \kappa''; \sigma') = size(\sigma') + 3$. Nach Definition folgt dann $([\nu, i']; i; \kappa''; \sigma')[size(\sigma') + 3] = [\nu, i']; i; \kappa''; \sigma'$. Dann folgt mit (Link) aus 15:

16.) $[\nu, i']; i; \kappa''; \sigma' \models size(\sigma') + 3 \sim\uparrow (\omega; \kappa')$

Insgesamt folgt dann mit (Conf) auf 15. und 16. angewendet
$(\cdot(\kappa'.1), size(\sigma') + 3, [\nu, i']; i; \kappa''; \sigma') \sim (\cdot(\kappa'.1), (\omega; \rho'), (\omega, \rho'); \uparrow \rho; \kappa''; S'')$

- (Beta-V-End): Es gilt $((\kappa'.1)\cdot, i, c; [\nu', i'']; i'; \kappa''; \sigma') \to (\kappa''\cdot, i', c; \sigma')$, sowie die Prämisse $\ell(\kappa') = \lambda\, id$ und die Voraussetzung $((\kappa'.1)\cdot, i, c; [\nu', i'']; i'; \kappa''; \sigma') \sim ((\kappa'.1)\cdot, \rho, S)$. Dies muss mit (Conf) hergeleitet worden sein und daher gilt

  1.) $c; [\nu', i'']; i'; \kappa''; \sigma' \sim S$

  1. kann nur durch mehrere Anwendungen von (Entry) hergeleitet worden sein, und daher muss $S = c; (\omega''; \rho''); \kappa''; \uparrow \rho'; S'$ gelten mit

  2.) $\sigma' \models i' \sim\uparrow \rho'$
  3.) $\sigma' \sim S'$

  Zunächst kann man dann auf $((\kappa'.1)\cdot, \rho, c; (\omega''; \rho''); \uparrow \rho'; \kappa''; S')$ (Beta-V) anwenden und es folgt

  – $((\kappa'.1)\cdot, \rho, c; (\omega''; \rho''); \uparrow \rho'; \kappa''; S') \to (\kappa''\cdot, \rho'; c; S')$

  Mit (Entry) und (Const) folgt aus 3.

  4.) $c; \sigma' \sim c; S'$

Mit (Conf) folgt aus 2. und 4.

– $(\kappa''\cdot, i', c; \sigma') \sim (\kappa''\cdot, \rho', c; S')$

- (Let-Exec) Es gilt $((\kappa'.1)\cdot, i, \nu; \sigma') \to (\cdot(\kappa'.2), \mathit{size}(\sigma') + 2, [\nu, i]; i; \sigma')$, sowie $((\kappa'.1)\cdot, i, \nu; \sigma') \sim ((\kappa'.1)\cdot, \rho, S)$. Dies kann nur mit (Conf) hergeleitet worden sein und daher gilt

  1.) $\nu; \sigma' \models i \sim\uparrow \rho$

  2.) $\nu; \sigma' \sim S$

  S muss, da 2. nur mit (Entry) hergeleitet worden sein kann, von der Form $\omega; S'$ sein mit

  3.) $\sigma' \models \nu \sim \omega$

  4.) $\sigma' \sim S'$

  Dann gilt zunächst $((\kappa'.1)\cdot, \rho, S) \xrightarrow{\text{(Let-Exec)}} (\cdot(\kappa.2), (\omega; \rho), (\omega; \rho); \uparrow \rho; S')$.

  Aus 1. folgt mit zweimaliger Anwendung von Lemma 5.2

  5.) $i; \sigma' \models i \sim\uparrow \rho$

  Auf 4. und 5. kann Regel (Entry) angewendet werden

  6.) $i; \sigma' \sim\uparrow \rho; S'$

  Aus 3. folgt mit Lemma 5.2

  7.) $i; \sigma' \models \nu \sim \omega$

  Auf 5. und 7. kann Regel (Frame) angewendet werden und es ergibt sich

  8.) $i; \sigma' \models [\nu, i] \sim (\omega, \rho)$

  Mit Regel (Entry) folgt aus 6. und 8.

  9.) $[\nu, i]; i; \sigma' \sim (\omega, \rho); \uparrow \rho; S'$

  Nun sind die Stacks schon äquivalent, jetzt müssen nur noch die Pointer äquivalent sein. Es gilt $([\nu, i]; i; \sigma')[\mathit{size}(\sigma') + 2] = [\nu, i]; i; \sigma'$, daher folgt mit (Link) aus 9.

  10.) $[\nu, i]; i; \sigma' \models \mathit{size}(\sigma' + 2) \sim\uparrow (\omega, \rho)$

  Insgesamt ergibt sich aus 9. und 10. mit (Conf)

  11.) $(\cdot(\kappa'.2), \mathit{size}(\sigma') + 2, [\nu, i]; i; \sigma') \sim (\cdot(\kappa.2), (\omega; \rho), (\omega; \rho); \uparrow \rho; S')$

(b) Fallunterscheidung nach der angewendeten Regel.(Nur interessante Fälle)

- (Id): Es gilt $(\cdot\kappa, \rho, S) \to (\kappa\cdot, \rho, \omega; S)$ mit folgenden Prämissen:

  1.) $\ell(\kappa) = \mathit{id}$

  2.) $\mathit{lookup}(\rho, \imath_\kappa) = \omega$.

  Da $(\cdot\kappa, i, \sigma) \sim (\cdot\kappa, \rho, S)$ nur mit Regel (Conf) hergeleitet werden konnte muss außerdem gelten:

  3.) $\sigma \models i \sim\uparrow \rho$

  4.) $\sigma \sim S$.

  Dann folgt mit Lemma 5.4 aus 3.

  5.) $\sigma \models \mathit{lookup}(\sigma, i, \imath_\kappa) \sim \mathit{lookup}(\rho, \imath_\kappa)$.

Wegen 2. muss

6.) $lookup(\sigma, i, \imath_\kappa) = \nu$

gelten. Damit sind alle Prämissen für (Id) erfüllt (1. und 6.) und es gilt $(\cdot\kappa, i, \sigma) \to (\kappa\cdot, i, \nu; \sigma)$.

Bleibt zu zeigen, dass $\zeta' \sim \delta'$ gilt:

Aus 4. und 5. folgt mit Regel (Entry) $\nu; \sigma \sim \omega; S$. Nun kann man Regel (Conf) anwenden und es gilt $(\kappa\cdot, i, \nu; \sigma) \sim (\kappa\cdot, \rho, \omega; S)$.

- (Beta-V): Es gilt:
  $((\kappa'.2)\cdot, \rho, \omega''; (\kappa'', \rho''); S') \to (\cdot(\kappa''.1), (\omega''; \rho''), (\omega'', \rho''); \uparrow \rho; \kappa'; S')$ mit

  1.) $\ell(\kappa') = App$
  2.) $\ell(\kappa'') = \lambda\, id$

  $((\kappa'.2)\cdot, i, \sigma) \sim ((\kappa'.2)\cdot, \rho, \omega''; (\kappa'', \rho''); S')$ kann nur mit Regel (Conf) gelten die Prämissen:

  3.) $\sigma \models i \sim\uparrow \rho$
  4.) $\sigma \sim \omega''; (\kappa'', \rho''); S'$

  Wegen 2. muss $\sigma = \nu''; (\kappa'', i''); \sigma'$ gelten, da 2. nur durch mehrmaliges Anwenden von (Entry) hergeleitet werden konnte, mit

  5.) $(\kappa'', i''); \sigma' \models \nu'' \sim \omega''$
  6.) $\sigma' \models i'' \sim \rho''$
  7.) $\sigma' \sim S'$

  Dann kann man auf $((\kappa'.2)\cdot, i, \nu''; (\kappa'', i''); \sigma')$ (Beta-V) anwenden und es gilt $((\kappa'.2)\cdot, i, \nu''; (\kappa'', i''); \sigma') \to (\cdot(\kappa''.1), size(\sigma') + 3, [\nu'', i'']; i; \kappa'; \sigma')$.

  Es bleibt zu zeigen, dass $\zeta' \sim \delta'$ gilt:

  Durch mehrmaliges Anwenden von (Entry) auf 7., ergibt sich

  8.) $i; \kappa'; \sigma' \sim\uparrow \rho; \kappa'; S'$

  Wenn man Lemma 5.2 auf 5. und 6. anwendet, ergibt sich

  9.) $i; \kappa'; \sigma' \models \nu'' \sim \omega''$
  10.) $i; \kappa'; \sigma' \models i'' \sim \rho''$

  Mit (Frame) ergibt sich aus 9. und 10.

  11.) $i; \kappa'; \sigma' \models [\nu'', i''] \sim (\omega''; \rho'')$

  Mit (Entry) kann man dann aus 8. und 11. folgendes schließen

  12.) $[\nu'', i'']; i; \kappa'; \sigma' \sim (\omega''; \rho''); \uparrow \rho; \kappa'; S'$

  Nun ist gezeigt, dass die Stacks äquivalent sind und es bleibt noch zu zeigen, dass die Pointer dies ebenfalls sind:

  Nach Definition gilt $([\nu'', i'']; i; \kappa'; \sigma')[size(\sigma') + 3] = [\nu'', i'']; i; \kappa'; \sigma'$, d.h. $([\nu'', i'']; i; \kappa'; \sigma')[size(\sigma')+3] \sim (\omega''; \rho''); \uparrow \rho; \kappa'; S'$. Wenn man nun (Link) anwendet, ergibt sich

  13.) $[\nu'', i'']; i; \kappa'; \sigma' \models size(\sigma') + 3 \sim (\omega''; \rho'')$.

  Insgesamt folgt mit (Conf) dann aus 12. und 13.

– $(\cdot(\kappa''.1), size(\sigma') + 3, [\nu'', i'']; i; \kappa'; \sigma') \sim (\cdot(\kappa''.1), (\omega''; \rho''), (\omega'', \rho''); \uparrow \rho; \kappa'; S')$ □

5.4 Typsysteme

Momentan würde ein Programm, welches, wie in Beispiel 5.2, eine Funktion als Resultat liefert, in unserer Stacksemantik einfach stecken bleiben, da keine Regel mehr anwendbar ist. Dies wäre für den Programmierer nur schwer zu sehen. Er würde nicht wirklich wissen, wo sein Fehler lag, da dies erst zur Laufzeit passieren würde.

Jetzt wollen wir mit Hilfe eines Typsystems diese Fehler schon zur Compilezeit erkennen und dem Programmierer dadurch eine frühere und einfachere Fehlerkorrektur ermöglichen.

Dazu wird zunächst ein allgemeines Typsystem eingeführt, welches auf dem aus [Sie10] beruht, und dieses dann so eingeschränkt, dass die Ausdrücke die durch die Einschränkung der Semantik stecken bleiben auch durch das Typssystem abgelehnt werden. Damit sind dann nur noch Ausdrücke wohlgetypt, die sich auch mit Hilfe des realistischen Stacks auswerten lassen.

5.4.1 Allgemeines Typsystem

Zunächst müssen wir Typen einführen, die wir den Ausdrücken zuordnen. Hier reicht eine Art von Typen. Später bei der Einschränkung werden wir zwischen Basistypen und „normalen“ Typen unterscheiden.

Definition 5.7 (Typen) *Die Menge Type aller gültigen* Typen τ *ist durch die folgende kontextfreie Grammatik definiert:*

$$\begin{array}{rcll} \tau & ::= & \textbf{bool} \mid \textbf{int} & \textit{(Grundtypen)} \\ & \mid & \tau_1 \to \tau_2 & \textit{(Funktionstypen)} \\ & \mid & (\tau_1 \times \ldots \times \tau_n) & \textit{(Produkttypen)} \end{array}$$

Um zu bestimmen, welchen Typ ein Ausdruck besitzt, führen wir Typurteile ein, welche aus einer Typumgebung, dem Ausdruck und dem passenden Typ bestehen. Eine Typumgebung Γ wird benötigt um die Typen für Variablen zu speichern.

Definition 5.8 (Typumgebung) *Eine Typumgebung* Γ *ist eine partielle endliche Funktion* $\Gamma : Id \to Type$.

Γ *kann als Liste dargestellt werden:* $[id_1 : \tau_1, \ldots, id_n : \tau_n]$ *ist die Funktion* Γ *mit* $dom(\Gamma) = \{id_1, \ldots, id_n\}$ *und* $\Gamma(id_i) = \tau_i$ *für* $i = 1, ..., n$.

Nun können wir darauf aufbauend Typregeln aufstellen.

Definition 5.9 (Typregeln) *Ein Typurteil* $\Gamma \triangleright e :: \tau$ *heißt gültig, wenn es sich mit den Regeln in Abbildung 5.5 herleiten lässt.*

$$\frac{}{b :: \mathbf{bool}}\ (\textsc{Bool}) \qquad \frac{}{n :: \mathbf{int}}\ (\textsc{Int}) \qquad \frac{op \in \{=, \leq, \geq, <, >\}}{op :: \mathbf{int} \to \mathbf{int} \to \mathbf{bool}}\ (\textsc{Rop}) \qquad \frac{op \in \{+, -, *\}}{op :: \mathbf{int} \to \mathbf{int} \to \mathbf{int}}\ (\textsc{Aop})$$

$$\frac{1 \leq i \leq n \qquad n > 1}{\#_i \ :: (\tau_1 \times \ldots \times \tau_n) \to \tau_i}\ (\textsc{Proj}) \qquad \frac{c :: \tau}{\Gamma \rhd c :: \tau}\ (\textsc{Const}) \qquad \frac{\Gamma(id) = \tau}{\Gamma \rhd id :: \tau}\ (\textsc{Id}) \qquad \frac{(id : \tau; \Gamma) \rhd e :: \tau'}{\Gamma \rhd \lambda id.\, e :: \tau \to \tau'}\ (\textsc{Abstr})$$

$$\frac{\Gamma \rhd e_1 :: \tau \to \tau' \quad \Gamma \rhd e_2 :: \tau}{\Gamma \rhd e_1\, e_2 :: \tau'}\ (\textsc{App}) \qquad \frac{\Gamma \rhd e_1 :: \tau \quad (id : \tau; \Gamma) \rhd e_2 :: \tau'}{\Gamma \rhd \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 :: \tau'}\ (\textsc{Let})$$

$$\frac{(id : \tau; \Gamma) \rhd v :: \tau \quad (id : \tau; \Gamma) \rhd e :: \tau'}{\Gamma \rhd \mathbf{let\,rec}\ id = v\ \mathbf{in}\ e :: \tau'}\ (\textsc{Let-Rec}) \qquad \frac{\Gamma \rhd e_0 :: \mathbf{bool} \quad \Gamma \rhd e_1 :: \tau \quad \Gamma \rhd e_2 :: \tau}{\Gamma \rhd \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 :: \tau}\ (\textsc{Cond})$$

$$\frac{\Gamma \rhd e_1 :: \tau_1 \quad \ldots \quad \Gamma \rhd e_n :: \tau_n}{\Gamma \rhd (e_1, \ldots, e_n) :: (\tau_1 \times \ldots \times \tau_n)}\ (\textsc{Tuple})$$

Abbildung 5.5: Typregeln

5.4.2 Einschränkung des Typsystems

Nun schränken wir das Typsystem soweit ein, dass die wohlgetypten Programme mit der neuen Semantik nicht mehr stecken bleiben. Dafür führen wir zunächst Basistypen ein.

Definition 5.10 (Basistypen) *Die Menge aller gültigen* Basistypen β *ist definiert durch:*

$$\beta \quad ::= \quad \mathbf{bool} \mid \mathbf{int} \mid (\beta_1 \times \ldots \times \beta_n)$$

Die Typregeln müssen wir an zwei Stellen einschränken: λ-Abstraktion und **let**-Ausdrücke, dort dürfen nur Basistypen als Ergebnistypen auftreten. Dies ist in Abbildung 5.6 zu sehen.

$$\frac{(id : \tau; \Gamma) \rhd e :: \beta}{\Gamma \rhd \lambda id.\, e :: \tau \to \beta}\ (\textsc{Abstr}) \qquad \frac{\Gamma \rhd e_1 :: \tau \quad (id : \tau; \Gamma) \rhd e_2 :: \beta}{\Gamma \rhd \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 :: \beta}\ (\textsc{Let})$$

Abbildung 5.6: Eingeschränkte Typregeln

Zusammenhang zur eingeschränkten Semantik

Um einen Zusammenhang zu den Einschränkungen in der Semantik herzustellen, beginnen wir bei der Substitutionssemantik[AO09, Kap. 2]. Wenn man auch hier die Einschränkungen vornimmt, wie sie in Abbildung 5.1 für die Umgebungssemantik gemacht wurden, so kann man sagen, dass wenn dort ein big step existiert, er auch in der allgemeineren Semantik existiert. Also wenn $e \Downarrow v$ in der eingeschränkten, so

auch $e \Downarrow v$ in der allgemeineren Semantik, da es sich ja lediglich um Einschränkungen handelt.

Genau so kann man auch beim Typsystem argumentieren. Wenn $\Gamma \triangleright e :: \tau$ im eingeschränkten Typsystem gilt, dann gilt es auch im allgemeineren.

Dank der Typerhaltung, welche in [Sie10] für die Substitutionssemantik bewiesen wurde, gilt wenn $e \Downarrow v$ und $\Gamma \triangleright e :: \tau$, dann auch $\Gamma \triangleright v :: \tau$. Dies gilt auch für die Einschränkungen.

Insbesondere gilt, dass wenn e der Rumpf einer λ-Abstraktion oder eines **let**-Ausdrucks ist, also $\Gamma \triangleright e :: \beta$ im eingeschränkten Typsystem, das Resultat v von e nur eine Konstante sein kann.

Diese Beobachtung gilt nicht nur für die Substitutionssemantik, sie überträgt sich durch die Äquivalenzbeweise auch auf alle unsere big step Umgebungssemantiken. Damit ist klar, dass die eingeschränkten big step Regeln (Beta-V) und (Let) ausreichen, d.h. dass Programme auch mit den eingeschränkten small step Regeln nicht stecken bleiben.

6 Compiler

Die small steps der Stacksemantik aus Kapitel 5 arbeiten bereits auf einer "maschinennahen" Datenstruktur, nämlich auf dem Stack σ, der – wenn man von den Tupeln absieht – nur Einträge begrenzter Größe enthält. Neben dem Stack σ (und dem dazugehörigen Zeiger i) benutzt aber jeder small step auch noch den Syntaxbaum des Gesamtprogramms, indem er z.B. auf die Markierung des (in der Position *pos* enthaltenen) Knotens κ oder auf dessen Index $\imath_\kappa$ zugreift.

Wenn wir das Gesamtprogramm in ein Maschinenprogramm übersetzen wollen, so müssen wir die *statische Information*, die sich aus *pos* ablesen lässt, und die *dynamische Information*, die in i und σ steckt, voneinander trennen. Dazu ordnen wir jeder Position *pos* eine Folge von Maschinenbefehlen zu, die "auf dem Stack σ" so arbeitet, wie es der entsprechende small step $(pos, i, \sigma) \to \ldots$ verlangt.

Da wir nur das Prinzip der Übersetzung schildern wollen, wählen wir als Zielmaschine eine abstrakte Maschine, die auf einer ähnlichen Datenstruktur arbeitet wie die Stacksemantik und deren Befehlsvorrat sehr mächtige Befehle enthält. Letzteres ließe sich leicht ändern, indem man jeden der mächtigen Befehle durch eine Folge von elementaren Befehlen ersetzt. Die Umrechnung von Programmspeicheradressen, die dazu nötig wäre, ist in der nun folgenden Definition der Übersetzung bereits enthalten.

Der Stack $\hat{\sigma}$, auf dem die Maschine arbeitet, ergibt sich aus σ, indem man jeden Knoten durch die entsprechende Programmspeicheradresse ersetzt. Das bedeutet natürlich, dass man auch die Funktionen *lookup* und *expand* an $\hat{\sigma}$ anpassen muss. Auf die genauen Definitionen verzichten wir hier.

Wie oben erwähnt, soll jeder Position *pos* des Syntaxbaums eine Befehlsfolge zugeordnet werden, die den entsprechenden small step simuliert. Das gesamte Maschinenprogramm erhält man dann, indem man diese Befehlsfolgen in geeigneter Reihenfolge aneinanderhängt. Diese Reihenfolge sollte so gewählt sein, dass bei der Ausführung des Maschinenprogramms keine unnötigen Sprünge stattfinden. Es genügt natürlich, die Reihenfolge auf den Positionen festzulegen. Dazu definiert man

- eine Anfangsposition pos_0
- eine Endpositon pos_n $(n = |Pos| - 1)$
- eine bijektive Funktion $next : Pos \setminus \{pos_n\} \to Pos \setminus \{pos_n\}$

Eine einfache Festlegung der Reihenfolge (die nicht allzu viele unnötige Sprünge verursacht) erhält man, indem man "am Baum entlang" läuft, wie wir es schon von den Beweisbäumen kennen. Formal lässt sie sich so definieren:

Definition 6.1 *Die Funktion* $next : Pos \setminus \{pos_n\} \to Pos \setminus \{pos_0\}$ *ist definiert durch:*

$$next(\cdot\kappa) = \begin{cases} \kappa\cdot & \text{wenn } \kappa \text{ Blatt} \\ \cdot(\kappa.1) & \text{sonst} \end{cases}$$

$$next((\kappa.i)\cdot) = \begin{cases} \cdot(\kappa.i+1) & \text{wenn } \kappa.i+1 \in K \\ \kappa\cdot & \text{sonst} \end{cases}$$

Daraus können wir nun eine lineare Ordnung für die Positionen festlegen.

Definition 6.2 *Sei* e *das Gesamtprogramm mit Syntaxbaum* $T = (K, \ell)$ *und* Pos *die Menge der Positionen in* T*. Auf* Pos *legt man eine lineare Ordnung* $pos_0, \ldots, pos_n$ *fest durch*

- $pos_0 = \cdot\epsilon$
- $pos_n = \epsilon\cdot$
- $pos_{i+1} = next(pos_i)$

Beispiel 6.1 In Abbildung 6.1 ist die Reihenfolge der Programmadressen für den Ausdruck $(\lambda x.\, x)\, 1$ abgebildet. Der Nachfolger einer Position $next(pos)$ ist einfach die nächste Position pos' in der Reihenfolge der Pfeile.

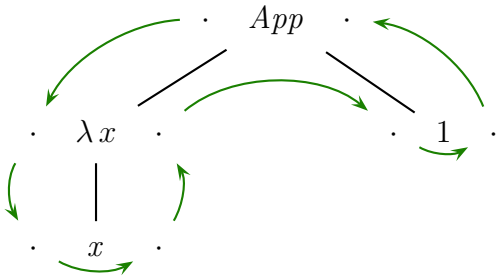

Abbildung 6.1: Programmadressenreihenfolge im Ausdruck $(\lambda x.\, x)\, 1$

Um Sprungadressen korrekt berechnen zu können, muss man sich nun überlegen, wie lang die Befehlsfolgen sind, die man zur Simulation eines small steps an der Position pos benötigt. Dies wird durch eine Funktion $\#_{cmds} : Pos \to \mathbb{N}$ festgelegt. Die Definition dieser Funktion hängt natürlich von der Reihenfolge der Positionen ab und vom Befehlsvorrat der Zielmaschine.

Die Reihenfolge der Positionen wurde oben so gewählt, dass für jeden small step, der nur die Position verändert, *gar kein* Befehl benötigt wird. Den Befehlsvorrat der Zielmaschine werden wir so festlegen, dass für alle übrigen small steps *ein einziger* Befehl ausreicht. Dementsprechend definieren wir:

Definition 6.3 *Die Funktion* $\#_{cmds} : Pos \to \mathbb{N}$ *ist definiert durch:*

- $pos = \cdot\kappa$:
  - $\ell(\kappa) = id$: $\#_{cmds}(pos) = 1$
  - $\ell(\kappa) = c$: $\#_{cmds}(pos) = 1$
  - $\ell(\kappa) = \lambda\, id$: $\#_{cmds}(pos) = 1$
  - *sonst* $\#_{cmds}(pos) = 0$.

- $pos = \kappa\cdot$:
  - $\kappa = (\kappa'.2)$ *und* $\ell(\kappa') = App$
    - *Wenn* $\ell(\kappa'.1) = \lambda\, id$, *dann* $\#_{cmds}(pos) = 1$.
    - *sonst:* $\#_{cmds}(pos) = 1$.
  - $\kappa = (\kappa'.1)$ *und* $\ell(\kappa') = Cond$: $\#_{cmds}(pos) = 1$.
  - $\kappa = (\kappa'.2)$ *und* $\ell(\kappa') = Cond$: $\#_{cmds}(pos) = 1$.
  - $\kappa = (\kappa'.n)$ *und* $\ell(\kappa') = Tuple\, n$: $\#_{cmds}(pos) = 1$.
  - $\kappa = (\kappa'.1)$ *und* $\ell(\kappa') = \textbf{let}\, id$: $\#_{cmds}(pos) = 1$.
  - $\kappa = (\kappa'.2)$ *und* $\ell(\kappa') = \textbf{let}\, id$: $\#_{cmds}(pos) = 1$.
  - $\kappa = (\kappa'.1)$ *und* $\ell(\kappa') = \lambda\, id$: $\#_{cmds}(pos) = 1$.
  - $\kappa = \epsilon$: $\#_{cmds}(pos) = 1$.
  - *sonst* $\#_{cmds}(pos) = 0$.

Diese Funktion nimmt sowohl auf den Befehlsvorrat der Maschine, als auch auf die Codegenerierung für unsere Sprache Bezug. Diese werden wir allerdings erst später definieren.

Mit Hilfe der Funktion $\#_{cmds}$ kann man die Adresse $addr(pos)$ berechnen, an der die zu *pos* gehörige Folge von Maschinenbefehlen beginnt:

Definition 6.4 *Sei* $Addr = \mathbb{N}$. *Dann ist die Funktion addr definiert durch:*

- $addr(pos_0) = 0$

- $addr(next(pos)) = addr(pos) + \#_{cmds}(pos)$

Jetzt definieren wir den Befehlsvorrat der Maschine. Die Befehle werden danach durch die Übergangsrelation „mit Leben gefüllt“.

Definition 6.5 (Befehle der Maschine) *Die Menge Command aller Maschninenbefehle command ist definiert durch:*

command	::=	LDC c	*(Konstante laden)*
	|	LDV d	*(Wert laden mit $d \in \mathbb{N}$)*
	|	LDF (d, pc)	*(Funktion laden ($d \in \mathbb{N}$ und $pc \in Addr$))*
	|	CLOS pc	*(Closure bilden mit $pc \in Addr$)*
	|	OP op	*(Operation ausführen $op \in Op$)*
	|	PROJ i	*(i-te Projektion ($i \in \mathbb{N}$))*
	|	TUPLE n	*(Tupel der Größe n erstellen ($n > 1$))*
	|	FRAME	*(Stackframe erstellen)*
	|	RETURN	*(Rücksprung aus Funktion)*
	|	LET-FRAME	*(Stackframe erstellen für* **let***)*
	|	LET-RETURN	*(Rücksprung aus* **let***)*
	|	JMP pc	*(Sprung mit $pc \in Addr$)*
	|	JMPF pc	*(Bedingter Sprung mit $pc \in Addr$)*
	|	STOP	*(Programmende)*

Mit Hilfe dieser Befehle ergibt sich nun die Übergangsrelation. An dieser sieht man sofort die Ähnlichkeit zur Stacksemantik.

Definition 6.6 *Ein Maschinenprogramm prog ist eine nicht leere Folge von Maschinenbefehlen. prog[i] bezeichnet den i-ten Befehl des Programms prog(wobei wir bei 0 beginnen). Die Übergangsrelation für ein Programm prog ist in Abbildung 6.2 zu sehen.*

Die Trennung zwischen LDV d und LDF (d, κ) wird vorgenommen, um so viel statische Information wie möglich zu nutzen. Diese Trennung müsste sonst das *lookup* zur Laufzeit übernehmen.

Zunächst müssen wir die Sprache einschränken, da wir sonst zur Laufzeit überprüfen müssten, ob auf dem Stack ein Operator op, eine Projektion $\#_i$ oder eine Closure liegt. Wir lassen deshalb op und $\#_i$ nicht mehr als eigenständige Konstanten zu, sondern nur noch als linke Seite von Applikationen. Also ersetzen wir die Produktionen:

c	::=	op	(Operator)
	|	$\#_i$	(Projektion mit $i \in \mathbb{N}$)

durch:

e	::=	$op\, e'$	(Operator angewendet auf einen Ausdruck)
	|	$\#_i\, e'$	(Projektion angewendet auf einen Ausdruck mit $i \in \mathbb{N}$)

Die zur Position *pos* gehörige Befehlsfolge definieren wir wie folgt:

Definition 6.7 *Die Funktion cmds : Pos → Command* ist definiert durch:*

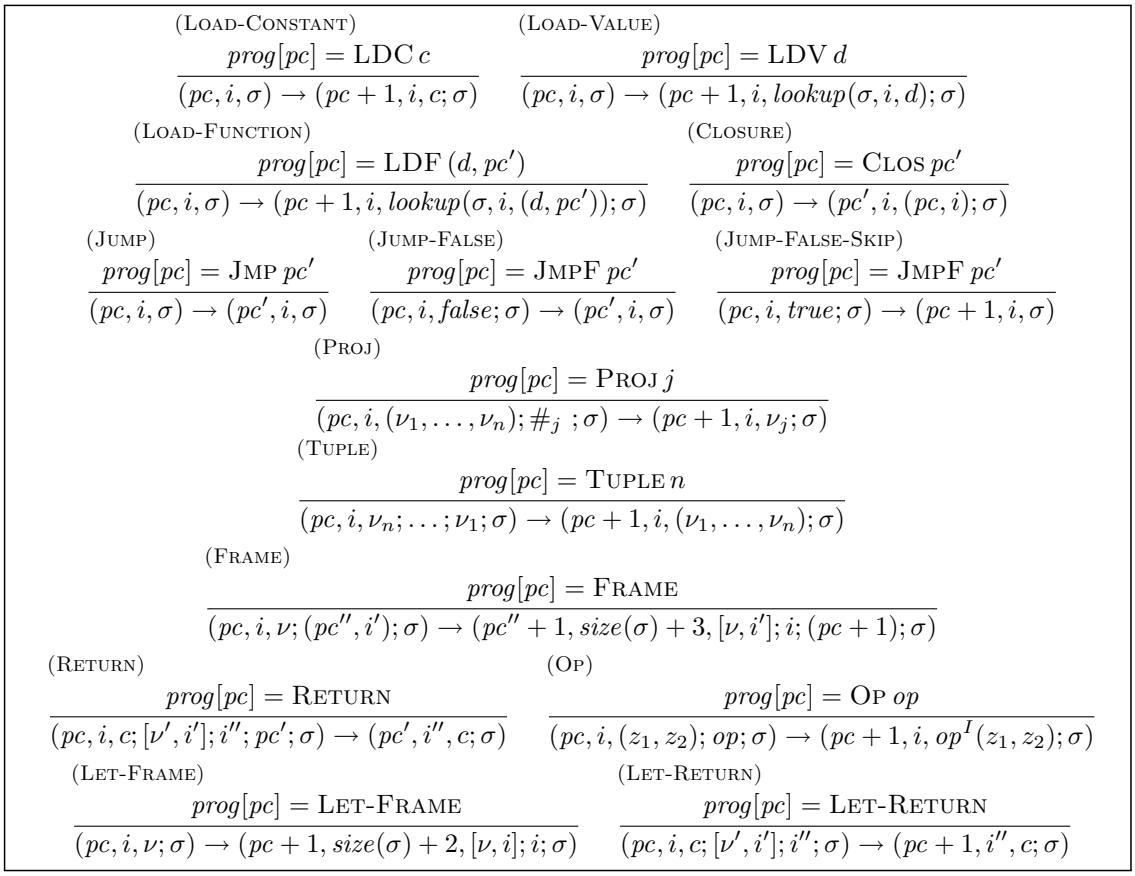

Abbildung 6.2: Small steps für das Maschinenprogramm *prog*

- $pos = \cdot\kappa$:
  - $\ell(\kappa) = id$ *und* $\iota_\kappa = d$, *dann ist* $cmds(pos) = \text{LDV}\, d$.
  - $\ell(\kappa) = id$ *und* $\iota_\kappa = (d, \kappa)$, *dann ist* $cmds(pos) = \text{LDF}\, (d, addr(\cdot\kappa))$.
  - $\ell(\kappa) = c$, *dann ist* $cmds(pos) = \text{LDC}\, c$.
  - $\ell(\kappa) = \lambda\, id$, *dann ist* $cmds(pos) = \text{Clos}\, addr(\kappa\cdot)$.
  - *sonst ist* $cmds(pos) = \epsilon$
- $pos = \kappa\cdot$:
  - $\kappa = (\kappa'.2)$, $\ell(\kappa') = App$ *und*
    - $\ell(\kappa'.1) = op$, *dann ist* $cmds(pos) = \text{Op}\, op$.
    - $\ell(\kappa'.1) = \#_i$, *dann ist* $cmds(pos) = \text{Proj}\, i$.
    - *sonst ist* $cmds(pos) = \text{Frame}$.
  - $\kappa = (\kappa'.1)$ *und* $\ell(\kappa') = Cond$, *dann ist* $cmds(pos) = \text{JmpF}\, addr(\cdot(\kappa'.3))$
  - $\kappa = (\kappa'.2)$ *und* $\ell(\kappa') = Cond$, *dann ist* $cmds(pos) = \text{Jmp}\, addr(\kappa'\cdot)$
  - $\kappa = (\kappa'.n)$ *und* $\ell(\kappa') = Tuple\, n$, *dann ist* $cmds(pos) = \text{Tuple}\, n$
  - $\kappa = (\kappa'.1)$ *und* $\ell(\kappa') = \textbf{let}\, id$, *dann ist* $cmds(pos) = \text{Let-Frame}$

- $\kappa = (\kappa'.2)$ *und* $\ell(\kappa') = \textbf{let}\, id$*, dann ist* $cmds(pos) =$ LET-RETURN
- $\kappa = (\kappa'.1)$ *und* $\ell(\kappa') = \lambda\, id$*, dann ist* $cmds(pos) =$ RETURN
- $\kappa = \epsilon$*, dann ist* $cmds(pos) =$ STOP.
- *sonst ist* $cmds(pos) = \epsilon$

Definition 6.8 *Sei* $e \in Exp$ *mit Syntaxbaum* $T = (K, \ell)$*. Das zu* e *gehörige Maschinenprogramm prog ist definiert durch:*

$$prog = cmds(pos_0) \ldots cmds(pos_n).$$

Beispiel 6.2 Betrachten wir als Beispiel die Fibonacci-Funktion $e =$
$\textbf{let rec}\ fib = \lambda x.\, \textbf{if}\ <(x, 2)\ \textbf{then}\ 1\ \textbf{else}\ +(fib(-(x, 1)), fib(-(x, 2)))\ \textbf{in}\ fib(4)$
Der Syntaxbaum zu e ist in Abbildung 6.3 zu sehen.

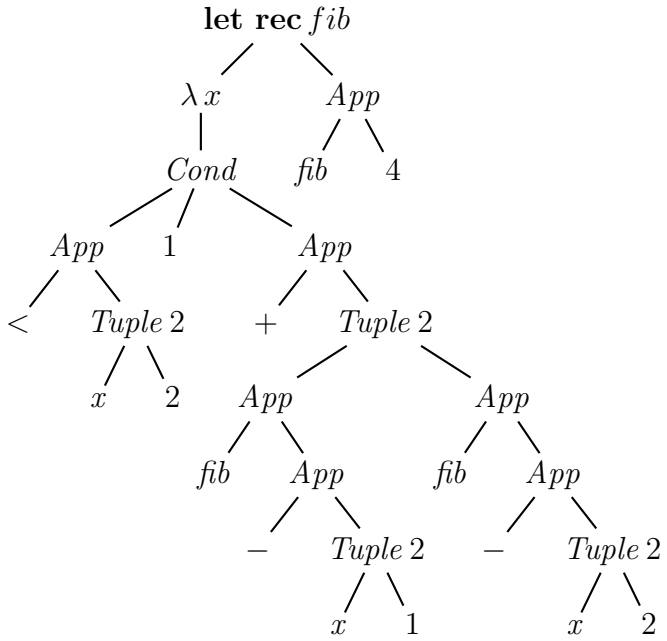

Abbildung 6.3: Syntaxbaum für Fibonacci-Funktion

Das zu e gehörige Programm *prog* sähe dann wie folgt aus.[1]

0 :	CLOS 25	8 :	JMP 25	16 :	FRAME	24 :	RETURN
1 :	LDC <	9 :	LDC +	17 :	LDF (1, 1)	25 :	LDF (0, 1)
2 :	LDV 0	10 :	LDF (1, 1)	18 :	LDC −	26 :	LDC 5
3 :	LDC 2	11 :	LDC −	19 :	LDV 0	27 :	FRAME
4 :	TUPLE 2	12 :	LDV 0	20 :	LDC 2	28 :	STOP
5 :	OP <	13 :	LDC 1	21 :	TUPLE 2		
6 :	JMPF 10	14 :	TUPLE 2	22 :	OP −		
7 :	LDC 1	15 :	OP −	23 :	FRAME		

[1] Zum besseren Verständnis wurde die Programmspeicheradresse direkt vor den Befehl geschrieben.

Verbindung zur Stacksemantik

Es bleibt hier noch zu zeigen, dass sich jeder small step in der Stacksemantik durch eine Folge von Maschinenbefehlen simulieren lässt.

Behauptung 6.1 (Korrektheit) *Sei p das Programm zum Ausdruck e, $T_e = (K, \ell)$ der Syntaxbaum zu e und Pos die Menge der Positionen in T_e, sowie $pos \in Pos$. Dann gilt:*

- $(pos, i, \sigma) \rightarrow_e (pos', i', \sigma') \Rightarrow (addr(pos), i, \hat{\sigma}) \rightarrow_p^* (addr(pos'), i', \hat{\sigma}')$
- $(pos, i, \sigma) \not\mapsto_e \Rightarrow (addr(pos), i, \hat{\sigma}) \not\mapsto_p$

Auf den Beweis verzichten wir hier. Wie bereits erwähnt fehlen dazu noch die Definitionen der Funktionen *lookup* und *expand*, sowie die genaue Defintion der Stacks $\hat{\sigma}$.

Ausblick

In dieser Arbeit ist es gelungen, für eine relativ einfache Sprache einen Compiler zu entwickeln. Dabei haben wir einige Optimierungen vorgenommen. Dennoch gibt es noch ein paar Dinge, die verbessert, bzw. realistischer gemacht werden können.

Zunächst könnte man am Compiler noch einiges machen. Bislang wird hier eine relativ unrealistische Maschine verwendet. Diese haben wir absichtlich so gewählt, damit sie zur Stacksemantik passt. Hier könnte man eine realistischere Maschine und damit eine realistischere Zielsprache verwenden. Außerdem wird bislang für jede small step Regel maximal ein Befehl erzeugt. Auch hier könnte man direkt eine ganze Befehlsfolge für die einzelnen Regeln angeben, was bei einer realistischen Maschinensprache notwendig wäre. Man könnte also elementarere Maschinenbefehle benutzen, die sich dann erst zu den mächtigeren zusammensetzen, die wir bislang definiert haben.

Im letzten Kapitel haben wir lediglich die Korrektheit (unbewiesen) in Form einer Behauptung angegeben, auch hier könnte man wieder versuchen eine Äquivalenz zu formulieren (und zu beweisen).

In Kapitel 5 mussten wir die Sprache einschränken, damit die Sprache die Stackdisziplin erfüllt. Hier könnte man einen Heap für die uneingeschränkte Sprache einführen. Dort wäre dann die Frage, welche Variablen im Heap und welche auf dem Stack landen. Wenn man schon einen Heap einführt, könnte man die funktionale Sprache, die wir bislang betrachtet haben, um imperative Konzepte erweitern. Dies wäre vom Umfang her vermutlich eine Diplomarbeit für sich.

Zusätzlich könnte man auch syntaktischen Zucker angeben, z.B. für die simultane Rekursion. Bislang ist es, wie in Beispiel 1.1 zu sehen, relativ kompliziert zwei Funktionen simultan zu definieren. Hier könnte man direkt zwei Namen, für die einzelnen Funktionen angeben.

Bei den Stacks ist es auch unrealistisch, dass ein ganzes Tupel in einem Slot steht. Hier würde man für jedes Element des Tupel einen Slot machen. Um dieses Tupel dann im Stack zu markieren, müsste man mit einem Typsystem arbeiten, welches die Größe des Tupels angibt.

Die Umwandlung der Rekursion in eine Iteration in Kapitel 3 könnte man systematischer angehen. Hier gäbe es die Möglichkeit ein Prinzip anzugeben mit dem man jede Rekursion in eine Iteration umwandeln und optimieren kann. Dieses Thema würde sich für eine ganze Diplomarbeit anbieten.

Zwischen den Semantiken wurden Äquivalenzen aufgestellt. Dabei wurde allerdings keine wirkliche Äquivalenz bewiesen, sondern lediglich eine „Ergebnisäquivalenz“, d.h. wenn die eine Semantik ein Ergebnis liefert, liefert die andere ein äquivalentes Ergebnis. Das Problem bei big step Semantiken ist, dass man bei die-

sen nicht zwischen Divergenz und stecken bleiben unterscheiden kann. Daher ist es schwierig eine Äquivalenz herzustellen, die angibt, ob beide Semantiken auch im gleichen Fall stecken bleiben, bzw. divergieren. Da es uns in diesem Zusammenhang auch nur darauf ankam eine Äquivalenz auf den Ergebnissen zu erreichen, wurde dies in dieser Arbeit nicht weiter vertieft.

Index

Literaturverzeichnis

[AO09] Afshar-Oromieh, Mehrnush: *Umgebungssemantiken*, Universität Siegen, Diplomarbeit, September 2009

[ASU99] Aho, Alfred V. ; Sethi, Ravi ; Ullmann, Jeffrey D.: *Compilerbau Teil 1*. 2. Auflage. München ; Wien : Oldenbourg Verlag, 1999. – ISBN 3–486–25294–1

[DB72] De Bruijn, Nicolaas G.: Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. In: *Indagationes Mathematicae (Elsevier)* 34 (1972), S. 381–392

[Dör77] Dörfler, Willibald: *Mathematik für Informatiker - Band 1 Finite Methoden und Algebra*. Carl Hanser Verlag, 1977. – ISBN 3–446–12365–2

[Kah87] Kahn, Gilles: Natural Semantics. In: *STACS '87: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*. London, UK : Springer-Verlag, 1987. – ISBN 3–540–17219–X, S. 22–39

[Knu68] Knuth, Donald E.: *The Art of Computer Programming*. Addison-Wesley, 1968. – ISBN 0–201–48541–9

[Lan64] Landin, Peter J.: The Mechanical Evaluation of Expressions. In: *The Computer Journal* 6 (1964), Nr. 4, 308-320. `http://dx.doi.org/10.1093/comjnl/6.4.308`. – DOI 10.1093/comjnl/6.4.308

[LMW86] Loeckx, Jacques ; Mehlhorn, Kurt ; Wilhelm, Reinhard: *Grundlagen der Programmiersprachen*. Stuttgart : Teubner, 1986. – ISBN 3–519–02254–0

[Mer97] Merzenich, Wolfgang: *Programmiersprachen und Compilerbau*. 1997. – Vorlesungsskript

[Mit96] Mitchell, John C.: *Foundations for Programming Languages*. MIT Press, 1996. – ISBN 0–262–13321–0

[Mit02] Mitchell, John C.: *Concepts in Programming Languages*. New York, NY, USA : Cambridge University Press, 2002. – ISBN 0–521–78098–5

[Plo81] Plotkin, G. D.: A Structural Approach to Operational Semantics. University of Aarhus, 1981 (DAIMI FN-19). – Forschungsbericht

[Sie10] Sieber, Kurt: *Theorie der Programmierung I.* 2010. – Vorlesungsmitschrift

[WM92] Wilhelm, Reinhard ; Maurer, Dieter: *Übersetzerbau - Theorie, Konstruktion, Generierung.* Springer, 1992. – ISBN 3–540–55704–0

Erklärung

Hiermit versichere ich, dass ich vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.

Siegen, den 26. April 2010 _____

(Simon Meurer)