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

Seminar Umgebungssemantiken

Umgebungssemantik

Reh, Carl Philipp

4. Mai 2009

Betreut durch Dipl.-Inf. Benedikt Meurer

Inhaltsverzeichnis

1 Einleitung

Die bisher bekannten Semantiken von TMPL sind die small-step- und big-step-Semantik nach [Sie09]. Die big-step-Semantik eignet sich bereits besser zur Implementierung eines Interpreters als die small-step-Semantik, da sich die Rückwärtsherleitung gut mit einer rekursiven Funktion implementieren lässt. Im Weiteren geht es darum, wie man (eine) neue Semantik(en) definieren kann, um die Umsetzung des Interpreters effizienter zu gestalten. Zunächst wird eine big-step-Substitutionssemantik betrachtet, die eine leichte Veränderung der nach [Sie09] definierten Semantik ist, um Beweise zu vereinfachen. Danach wird die big-step-Umgebungssemantik definiert, um eine effizientere Implementierung als die der big-step-Substitutionssemantik zu ermöglichen. Im Gegensatz zu [Sie09] wird zunächst eine call-by-name-Semantik betrachtet.

2 Substitutionssemantik

Definition: 1 (Syntax der Programmiersprache)**.** *Vorgegeben seien*

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

Die Mengen Op *aller* Operatoren op*,* $Const$ *aller* Konstanten c*,* Exp *aller* Ausdrücke e *und* Val *aller* Werte v *sind durch folgende kontextfreie Grammatik definiert:*

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

Hier ist Folgendes zu beachten:

- Die Konstante **fix** wird benutzt, um Rekursion in die Sprache einzuführen.
- Identifier sind keine Werte, denn sie müssen in der Umgebungssemantik noch ausgewertet werden, d.h.: $Id \cap Val = \emptyset$.
- Es wird festgelegt, dass Ausdrücke gleich modulo-Umbenennung gebundener Namen sind.

Definition: 2 (Bisherige Definition der Substitutionssemantik)**.** *Die in [Sie09] betrachtete Substitutionssemantik ist eine call-by-value-Semantik, was sich durch Regel* (Beta-v) *auszeichnet.*

(Val) $v \Downarrow v$

(Beta-v) $$\frac{e[v'/id] \Downarrow v}{(\lambda id.e)\, v' \Downarrow v}$$

(Op) $op\, z_1\, z_2 \Downarrow op^I(z_1, z_2)$

(Cond-True) $$\frac{e_0 \Downarrow \mathit{true} \quad e_1 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$$

(Cond-False) $$\frac{e_0 \Downarrow \mathit{false} \quad e_2 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$$

(Unfold) $$\frac{e[\textbf{rec } id.\ e/id] \Downarrow v}{\textbf{rec } id.\ e \Downarrow v}$$

(App) $$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 v_2 \Downarrow v}{e_1 e_2 \Downarrow v}$$

Hier wird zunächst eine call-by-name-Semantik betrachtet. Außerdem ist zu beachten, dass (App) entfällt. (App) muss in der bisherigen Semantik in jedem Fall als letzte mögliche Wahl versucht werden, da die Herleitung sonst unendlich lang sein kann.

(1) $(+)\ 1\ 2 \Downarrow (+)\ 1\ 2$
- (App)
  - (2) $(+)\ 1 \Downarrow (+)\ 1$
    - (Val)
  - (3) $2 \Downarrow 2$
    - (Val)
  - (4) $(+)\ 1\ 2 \Downarrow \ldots$
    - $(\ldots)$

Um dies zu vermeiden, wurde (App) in (Op-1), (Op-2) und (Beta) aufgeteilt.

Definition: 3 (Big step Regeln)**.** *Ein* big-step *in der Substitutionssemantik ist eine Formel der Gestalt* $e \Downarrow v$ *mit* $e \in Exp$ *und* $v \in Val$*. Ein solcher big-step heißt* gültig*, wenn er sich mit den folgenden Regeln herleiten lässt:*

(Val) $v \Downarrow v$

(Beta) $$\frac{e_1 \Downarrow \lambda id.\, e \quad e[e_2/id] \Downarrow v}{e_1\, e_2 \Downarrow v}$$

(Op-1) $$\frac{e_1 \Downarrow op \quad e_2 \Downarrow z}{e_1\, e_2 \Downarrow op\, z}$$

(Op-2) $$\frac{e_1 \Downarrow op\, z_1 \quad e_2 \Downarrow z_2 \quad op^I(z_1, z_2) = z}{e_1\, e_2 \Downarrow z}$$

(Cond-True) $$\frac{e_0 \Downarrow true \quad e_1 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$$

(Cond-False) $$\frac{e_0 \Downarrow false \quad e_2 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$$

(Unfold) $$\frac{e_1 \Downarrow \textbf{fix} \quad e_2\,(\textbf{fix}\, e_2) \Downarrow v}{e_1\, e_2 \Downarrow v}$$

3 Umgebungssemantik

3.1 Motivation

Wie schon angedeutet, ist die Substitutionssemantik ungeeignet für eine Implementierung, denn die Ersetzung eines Ausdrucks hat lineare Komplextität in der Länge des zu ersetzenden Ausdrucks und das Suchen von zu ersetzenden Namen hat lineare Komplexität im betrachteten Ausdruck. Es folgen ein paar Beispiele, die unnötigen Aufwand der Substitutionssemantik veranschaulichen sollen.

(a) $(\lambda x.1+2+3+4+5)\,0$ - Der Ausdruck $1+2+3+4+5$ wird nach Vorkommen von x durchsucht, was aber gar nicht nötig ist

(b) $(\lambda x.\textbf{if } true \textbf{ then } x \textbf{ else } x{+}x{+}x{+}x)(2{*}3{*}4)$ - x wird sehr viel häufiger substituiert als nötig

Diese ineffizienten Eigenschaften der Substitutionssemantik können beseitigt werden, indem der zu ersetzende Ausdruck in einer Umgebung gemerkt wird. Die Intuition dabei ist, dass nicht substituiert wird, sondern der Ausdruck bei Bedarf (wenn der Identifier ausgewertet wird) in der Umgebung nachgeschlagen wird (formale Definition folgt im Anschluss).

(a) $((\lambda x.1+2+3+4+5)\,0, [])$ - Durch Eintragen ergibt sich: $(1+2+3+4+5, [x:0])$. Vorteil: es wird nicht mehr nach x gesucht.

(b) $((\lambda x.\textbf{if } true \textbf{ then } x \textbf{ else } x+x+x+x)(2*3*4), [])$ - x wird nur im True-Zweig nachgeschlagen: $(x, [x : 2*3*4])$.

3.2 Definitionen

Definition: 4 (Closures und Umgebungen)**.** *Die Mengen Env aller* (Laufzeit-)Umgebungen η *und Cl aller* Closures cl *sind durch die folgende kontextfreie Grammatik definiert:*

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

Der Definitionsbereich $dom(\eta)$ *einer Umgebung* η *ist wie folgt induktiv definiert:*

$$\begin{aligned} dom([\,]) &= \emptyset \\ dom(id : cl; \eta) &= \{id\} \cup dom(\eta) \end{aligned}$$

Eine Closure $cl = (e, \eta)$ *heißt* gültig*, wenn* $free\,(e) \subseteq dom(\eta)$ *und* η *gültig ist. Eine Umgebung* η *heißt* gültig*, wenn alle eingetragenen Closures gültig sind.*

Schreibweisen

(a) Für $(id_1 : cl_1; \ldots; id_n : cl_n; [\,]) \in Env$ schreiben wir

$$[id_1 : cl_1; \ldots; id_n : cl_n]$$

(b) Für $\eta = (id_1 : cl_1; \ldots; id_n : cl_n; [\,]) \in Env$ und $id \in dom(\eta)$ sei

$$\eta(id) = cl_i, \text{ wobei } i = \min\{j \in \{1, \ldots, n\} \mid id_j = id\}$$

(c) Für $(e, [])$ schreiben wir verkürzt e.

Wird ein neuer Identifier in eine Umgebung eingetragen, so wird er links angehängt. Die Funktion η, die einen Identifier in der Umgebung nachschlägt, liefert den am weitesten links stehenden. So wird erreicht, dass neuere Einträge immer Vorrang haben. Beispiel: Sei $\eta = [x : 5, x : false]$. Dann ist $\eta(x) = 5$.

Definition: 5 (Big step Regeln)**.** *Ein* big-step *in der Umgebungssemantik ist eine Formel der Gestalt* $(e, \eta) \Downarrow (e', \eta')$*, wobei* $e, e' \in Exp$ *und* $\eta, \eta' \in Env$*. Ein derartiger big-step heißt* gültig*, wenn er sich mit den folgenden Regeln herleiten lässt:*

(Val) $(v, \eta) \Downarrow (v, \eta)$

(Id) $$\frac{\eta(id) \Downarrow cl}{(id, \eta) \Downarrow cl}$$

(Beta) $$\frac{(e_1, \eta) \Downarrow (\lambda id.\, e, \eta_1) \quad (e, id : (e_2, \eta); \eta_1) \Downarrow cl}{(e_1\, e_2, \eta) \Downarrow cl}$$

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

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

(Cond-True) $$\frac{(e_0, \eta) \Downarrow (true, \eta_0) \quad (e_1, \eta) \Downarrow cl}{(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2, \eta) \Downarrow cl}$$

(Cond-False) $$\frac{(e_0, \eta) \Downarrow (false, \eta_0) \quad (e_2, \eta) \Downarrow cl}{(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2, \eta) \Downarrow cl}$$

(Unfold) $$\frac{(e_1, \eta) \Downarrow (\textbf{fix}, \eta_1) \quad (e_2\, (\textbf{fix}\, e_2), \eta) \Downarrow cl}{(e_1\, e_2, \eta) \Downarrow cl}$$

Das folgende Beispiel soll eine einfache Herleitung in der Umgebungssemantik zeigen.

(1) $((\lambda x.(+)\, x\, x)\, 5, []) \Downarrow (10, [])$
- (Beta)
  - (2) $(\lambda x.(+)\, x\, x, []) \Downarrow (\lambda x.(+)\, x\, x, [])$
    - (Val)
  - (3) $((+)\, x\, x, [x : (5, [])]) \Downarrow (10, [])$
    - (Op-2)
      - (4) $((+)\, x, [x : (5, [])]) \Downarrow ((+)\, 5, [])$
        - (Op-1)
          - (5) $((+), [x : (5, [])]) \Downarrow ((+), [x : (5, [])])$
            - (Val)
          - (6) $(x, [x : (5, [])]) \Downarrow (5, [])$
            - (Id)
              - (7) $(5, []) \Downarrow (5, [])$
                - (Val)
      - (8) $(x, [x : (5, [])]) \Downarrow (5, [])$
        - (Id)
          - (9) $(5, []) \Downarrow (5, [])$
            - (Val)

Lemma 1 (Wohldefiniertheit der Umgebungssemantik)**.** *Seien* $(e, \eta), (\hat{e}, \hat{\eta}) \in Cl$*. Wenn* (e, η) *gültig und* $(e, \eta) \Downarrow (\hat{e}, \hat{\eta})$*, dann ist auch* $(\hat{e}, \hat{\eta})$ *gültig und es gilt* $\hat{e} \in Val$*.*

Proof. Der Beweis wird per Induktion über die Länge der Herleitung des big-steps und Fallunterscheidung der zuletzt angewandten Regel geführt. Für die Definition von $free$ siehe [Sie09].

- (v, η) ist gültig nach Voraussetzung.
- (id, η) ist gültig nach Voraussetzung, somit ist insbesondere auch η gültig. Dies impliziert, dass $\eta(id)$ ebenfalls gültig ist.
- $(e_1 e_2, \eta)$ ist gültig nach Voraussetzung. Da $free(e_1) \subseteq free(e_1 e_2)$, ist ebenfalls (e_1, η) gültig. (e_2, η) analog. Nach I.V. gilt, dass $(\lambda id.e, \eta_1)$ gültig ist. $\eta_2 = id : (e_2, \eta); \eta_1$ ist also auch gültig. Da $free(\lambda id.e) = free(e) \setminus id$, kann e höchstens noch id als zusätzliche freie Variable zu den freien Variablen von $(\lambda id.e)$ haben. Daraus folgt, dass (e, η_2) ebenfalls gültig ist.
- (Op-1) und (Op-2) sind trivialerweise erfüllt.
- (**if** e_0 **then** e_1 **else** e_2, η) ist nach Voraussetzung gültig. Da $free(e_1) \subseteq free($**if** e_0 **then** e_1 **else** $e_2)$ ist (e_1, η) ebenfalls gültig.
- (Cond-False) analog.
- $(e_1 e_2, \eta)$ ist gültig nach Voraussetzung. Da $fix \in c$ und $free(e_2\,($**fix** $e_2)) \subseteq free(e_1 e_2)$, ist $(e_2\,($**fix** $e_2), \eta)$ ebenfalls gültig.

□

Im Weiteren betrachten wir nur noch gültige Closures und Umgebungen, d.h. wir nehmen an, *Cl* und *Env* enthalten nur noch gültige Elemente.

Definition: 6. *Sei $(e, \eta) \in Cl$. Der Ausdruck $e\,\eta$ ist wie folgt induktiv definiert:*

$$\begin{aligned}
c\,\eta &= c \\
id\,\eta &= e'\,\eta',\ \textit{wobei}\ \eta(id) = (e', \eta') \\
(e_1\,e_2)\,\eta &= (e_1\,\eta)\,(e_2\,\eta) \\
(\lambda id.\,e)\,\eta &= \lambda id'.\,(e[id'/id]\,\eta),\ \textit{wobei}\ id' \notin dom(\eta) \\
(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\,\eta &= \textbf{if}\ (e_0\,\eta)\ \textbf{then}\ (e_1\,\eta)\ \textbf{else}\ (e_2\,\eta)
\end{aligned}$$

Lemma 2. *Seien $(e, \eta), (e', \eta') \in Cl$. Dann gilt:*

(a) *$e \in Val$ impliziert $(e\,\eta) \in Val$.*

(b) *$e\,(id : (e', \eta'); \eta) = (e[id'/id]\,\eta)[(e'\,\eta')/id']$ wenn $id' \notin dom(\eta)$*

Proof. (a) Fallunterscheidung über die Form von e:

- $e \in Const \rightarrow (e\,\eta) = e \in Val$
- $e = op\,z \rightarrow ((op\,z)\,\eta) = (op\,\eta)(z\,\eta) = op\,z \in Val$
- $e = (\lambda id.e)\eta = \lambda id'.(e[id'/id]\eta) \in Val$

(b) Induktion über die Größe des Ausdrucks und Fallunterscheidung über dessen Form:

- $e \in Const$. Dann gilt $c(id : (e', \eta'); \eta) = c = (c[id'/id]\eta)[(e', \eta')/id']$
- $e \in Id$. Dann gilt $id_0(id : (e', \eta'); \eta) = (e', \eta') = (id_0[id'/id]\eta)[(e', \eta')/id']$, falls $id_0 = id$. Ansonsten hat id keinen Effekt und die Bedingung gilt trivialerweise.
- $e = e_1 e_2$. Nach Definition und I.V. gilt:

$$\begin{aligned} & (e_1 e_2)(id : (e'\eta'); \eta) \\ = \quad & (e_1(id : (e'\eta'); \eta))(e_2(id : (e'\eta'); \eta)) \\ = \quad & ((e_1[id'/id]\eta)[(e', \eta')/id'])((e_2[id'/id]\eta)[(e', \eta')/id']) \\ = \quad & ((e_1 e_2)[id'/id]\eta)[(e', \eta')/id']) \end{aligned}$$

- $e = \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$ analog
- $e = (\lambda id.e)$:

$$\begin{aligned} & (\lambda id.e)(id'' : (e', \eta'); \eta) && (1) \\ = \quad & \lambda id'.((e[id'/id])(id'' : (e'\eta'); \eta)) && (2) \\ = \quad & \lambda id'.(e[id'/id])([id'''/id'']\eta)[(e'\eta')/id'''] && (3) \\ = \quad & (\lambda id'.(e[id'/id])([id'''/id'']\eta))[(e'\eta')/id'''] && (4) \\ = \quad & (\lambda id.e)([id'''/id''])\eta)[(e'\eta')/id'''] && (5) \end{aligned}$$

  a) (1) → (2) gilt nach Definition der Substitution

  b) (2) → (3) gilt nach I.V.

  c) (3) → (4) gilt nach Definition der Substitution

  d) (4) → (5) ebenso

□

4 Zusammenhang zwischen Substitutions- und Umgebungssemantik

4.1 Korrektheit

Satz 3 (Korrektheit der Umgebungssemantik)**.** *Seien* $(e, \eta) \in Cl$, $v' \in Val$ *und* $\eta' \in Env$. *Wenn* $(e, \eta) \Downarrow (v', \eta')$, *dann* $(e\,\eta) \Downarrow (v'\,\eta')$.

Proof. Induktion über die Länge der Herleitung des big-steps $(e, \eta) \Downarrow (v', \eta')$ mit Fallunterscheidung nach der zuletzt angewandten big-step-Regel:

- $(v, \eta) \Downarrow (v, \eta)$ mit (Val), dann ist $(v\,\eta) \in Val$ und somit folgt $(v\,\eta) \Downarrow (v\,\eta)$ mit (Val).
- $(id, \eta) \Downarrow (v', \eta')$ mit (Id) bedingt $\eta(id) = (\hat{e}, \hat{\eta})$ und $(\hat{e}, \hat{\eta}) \Downarrow (v', \eta')$. Nach I.V. muss also $(\hat{e}\,\hat{\eta}) \Downarrow (v'\,\eta')$ gelten, und somit folgt wegen $(id\,\eta) = (\hat{e}\,\hat{\eta})$ die Behauptung.

- Nach Voraussetzung gilt $(e_1 e_2, \eta) \Downarrow (v_0, \eta_0)$. Mit (Beta) kann dies nur aus den Prämissen der Form $(e_1, \eta) \Downarrow (\lambda id.e, \eta_1)$ und $(e, id : (e_2, \eta); \eta_1) \Downarrow (v_0, \eta_0)$ folgen. Setze $\eta_1' = id : (e_2, \eta); \eta_1$ zur Abkürzung. Nach I.V. gilt also für die erste Prämisse: $(e_1 \eta) \Downarrow ((\lambda id.e)\eta_1)$. Nach Definition der Substitution gilt: $((\lambda id.e)\eta_1) = \lambda id'.(e[id'/id]\eta_1)$. Setze $e' = e[id'/id]\eta_1$ zur Abkürzung des neuen Körpers des Lambdaausdrucks. Somit folgt ebenfalls nach I.V. für die zweite Prämisse: $(e'\eta_1') \Downarrow (v_0\eta_0)$, da nach Lemma 2 gilt: $(e'\eta_1') = (e[id'/id]\eta_1)[(e_2\eta)/id']$. Wegen $(e_1\eta) \Downarrow \lambda id'.e'$ und $(e'[(e_2\eta)/id']) \Downarrow (v_0\eta_0)$ gilt auch $(e_1\eta)(e_2\eta) = ((e_1 e_2)\eta) \Downarrow (v_0\eta_0)$

- Nach Voraussetzung gilt $(e_1 e_2, \eta) \Downarrow (v_0, [])$. Mit (Op-1) kann dies nur aus den Prämissen der Form $(e_1, \eta) \Downarrow (op, \eta')$ und $(e_2, \eta) \Downarrow (z, \eta'')$ folgen. Nach I.V. gilt: $(e_1\eta) \Downarrow ((op)\eta')$ und $(e_2\eta) \Downarrow (z\eta'')$. Mit (Op-1) aus der Substitutionssemantik gilt dann wegen $(op)\eta' = op$ und $(z\eta'') = z$ auch trivialerweise: $(e_1\eta)(e_2\eta) \Downarrow (op\ z)[] = op\ z$.

- (Op-2) analog

- Nach Voraussetzung gilt $(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2, \eta) \Downarrow (v_0, \eta')$. Mit (Cond-True) kann dies nur aus den Prämissen $(e_0, \eta) \Downarrow (true, \eta'')$ und $(e_1, \eta) \Downarrow (v_0, \eta')$ folgen. Nach I.V. gilt: $(e_0\eta) \Downarrow (true\ \eta'')$ und $(e_1\eta) \Downarrow (v_0\eta')$. Da $(true\ \eta'') = true$ folgt mit (Cond-True) aus der Substitutionssemantik, dass $(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\eta \Downarrow v_0\eta'$

- (Cond-False) analog

- Nach Voraussetzung gilt $(e_1 e_2, \eta) \Downarrow (v_0, \eta_0)$. Mit (Unfold) kann dies nur aus den Prämissen der Form $(e_1, \eta) \Downarrow (\textbf{fix}, \eta')$ und $(e_2, \eta) \Downarrow (e_2\ (\textbf{fix}\, e_2), \eta'')$ folgen. Nach I.V. gilt: $(e_1\eta) \Downarrow (\textbf{fix}\, \eta)$ und $(e_2\eta) \Downarrow ((e_2\, (\textbf{fix}\, e_2))\eta)$. Da $(\textbf{fix}\, \eta) = \textbf{fix}$ und $((e_2\, (\textbf{fix}\, e_2))\eta) = ((e_2\eta)(\textbf{fix}\, (e_2\eta)))$, folgt auch $((e_1 e_2)\eta) \Downarrow (v_0\eta_0)$.

□

4.2 Vollständigkeit

Die Umgebungssemantik ist bezüglich der Substitutionssemantik ggf. nicht vollständig. Folgende Vermutung sollte bewiesen werden: Seien $(e, \eta) \in Cl$ und $v \in Val$. Wenn $(e\, \eta) \Downarrow v$, dann ex. $(v', \eta') \in Cl$ mit $(v'\, \eta') = v$, sodass $(e, \eta) \Downarrow (v', \eta')$.

Es wird versucht, die Behauptung über die Länge der Herleitung des big-steps und Fallunterscheidung über die zuletzt angewandte Regel zu beweisen. Die Behauptung ist im Fall, dass e ein Identifier ist, trivialerweise wegen der I.V. erfüllt. Somit ist dieser Fall erledigt und es wird nur noch $e \notin Id$ betrachtet.

- (Val): Nach Voraussetzung gilt: $(v\eta) \Downarrow v$. Dann wähle $(v'\eta') = v$ und es gilt $(v, \eta) \Downarrow (v', \eta')$.

- (Beta): Nach Voraussetzung gilt: $((e_1 e_2)\eta) \Downarrow v$. Dies kann nur aus Prämissen der Form $(e_1\eta) \Downarrow \lambda id.e$ und $(e[e_2\eta/id]) \Downarrow v$ folgen. Nach I.V. existiert $(\lambda id.e, \eta_1') \in Cl$,

sodass auch gilt: $(e_1, \eta) \Downarrow (\lambda id.e, \eta_1')$. Damit dieser Fall beweisbar wäre, müsste zunächst folgender Zusammenhang bewiesen werden.

Existiere $(e[e_2\eta/id], \hat{\eta}) \Downarrow (v', \eta')$, dann existiert η'', sodass $(e, id : (e_2, \eta); \hat{\eta}) \Downarrow (v', \eta'')$.
Der Beweis wird über die Länge der Herleitung des big-steps und Fallunterscheidung der zuletzt angewandten Regel geführt.

- (Val): $(v[e_2\eta/id], \hat{\eta}) \Downarrow (v, \eta') \Rightarrow \exists \eta'' = [id : (e_2, \eta)] \wedge (v, \eta''; []) \Downarrow (v, \eta'')$
- (Id): Nach Voraussetzung gilt $(id[e_2\eta/id'], \hat{\eta}) \Downarrow (v', \eta')$. Falls $id = id'$, foglt: $(e_2\eta, \hat{\eta}) \Downarrow (v', \eta')$. Wähle $\eta^* = (id' : (e_2, \eta); \hat{\eta})$. Dann gilt $(id, \eta^*) \Downarrow (v, \eta'')$ nach I.V. Ansonsten wähle $\eta^* = \hat{\eta}$ und es gilt nach I.V.: $(id, \eta^*) \Downarrow (v', \eta'')$.
- (Beta): Nach Voraussetzung gilt: $(e_1 e_2[e_3\eta/id], \hat{\eta}) \Downarrow (v', \eta')$. Dies kann nur aus Prämissen der Form: $(e_1[e_3\eta/id]; \hat{\eta}) \Downarrow (\lambda id'.e, \eta_1)$ und $(e, id' : (e_2[e_3\eta/id]); \eta_1) \Downarrow (v', \eta')$ folgen. Nach I.V. existiert η_1', sodass $(e_1, id : (e_3, \eta); \eta_0) \Downarrow (\lambda id.e, \eta_1')$. Problem: I.V. ist für die zweite Prämisse nicht anwendbar, da die Substitution innen ist.

Somit kann die Behauptung der Vollständigkeit der Umgebungssemantik in dieser Form nicht gezeigt werden.

5 (Let) und (Rec) als abgeleitete Konzepte

5.1 (Let)

(Let) kann als syntaktischer Zucker eingeführt werden, um eine Deklaration zu vereinfachen.

Definition: 7. $\textbf{let}\ id = e_1\ \textbf{in}\ e_2 \equiv (\lambda id.e_2)e_1$.

Damit lässt sich folgende abgeleitete Regel formulieren.

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

Satz 4 (Korrektheit von (Let))**.** (Let) *ist eine abgeleitete Regel.*

Proof. Gelte: $(\textbf{let}\ id = e_1\ \textbf{in}\ e_2, \eta) \Downarrow cl$. Dies folgt aus der Prämisse $(e_2, id : (e_1, \eta), \eta) \Downarrow cl$. Wegen $(\textbf{let}\ id = e_1\ \textbf{in}\ e_2, \eta) \equiv ((\lambda id.e_2)e_1, \eta)$ kann auch (Beta) angewendet werden. Dies kann nur korrekt sein, falls die Prämissen: $(e_1, \eta) \Downarrow (\lambda id.e, \eta_1)$ und $(e, id : (e_2, \eta); \eta_1) \Downarrow cl$ herleitbar sind. Die erste Prämisse folgt trivialerweise mit (Val) und es gilt: $\eta = \eta_1$. Die zweite Prämisse gilt, da sie wegen (Let) vorausgesetzt wurde. □

5.2 (Rec)

(Rec) kann in der call-by-name-Semantik als syntaktischer Zucker über eine Lambda-Abstraktion eingeführt werden.

Definition: 8. $\textbf{rec}\ id.\ e \equiv \textbf{fix}(\lambda id.e)$

Damit lässt sich folgende abgeleitete Regeln formulieren.

$$\text{(Rec)} \quad \frac{(e, id : (\textbf{rec}\ id.\ e, \eta), \eta) \Downarrow cl}{(\textbf{rec}\ id.\ e, \eta) \Downarrow cl}$$

Satz 5 (Korrektheit von (Rec))**.** (Rec) *ist eine abgeleitete Regel.*

Proof. Gelte: $(\textbf{rec}\ id.\ e, \eta) \Downarrow cl$. Dies folgt aus der Prämisse: $(e, id : (\textbf{rec}\ id.\ e, \eta), \eta) \Downarrow cl$. Wegen $(\textbf{rec}\ id.\ e, \eta) \equiv (\textbf{fix}(\lambda id.e), \eta)$, kann zunächst Regel (Unfold) angewendet werden. Dies bedingt die Prämissen: $(\textbf{fix}, \eta) \Downarrow (\textbf{fix}, \eta')$ und $((\lambda id.e)\,(\textbf{fix}\,\lambda id.e), \eta) \Downarrow cl$. Die erste Prämisse folgt trivialerweise mit (Val). Die zweite Prämisse kann nur mit (Beta) hergeleitet sein. Dies bedingt wiederum, dass $(\lambda id.e, \eta) \Downarrow (\lambda id.e, \eta)$, was direkt aus (Val) folgt, und dass $(\textbf{fix}\,\lambda id.e, \eta) \Downarrow cl$. Ersetzt man nun $\textbf{fix}\,\lambda id.e$ durch den syntaktischen Zucker für **rec**, folgt mit der Prämisse von (Rec) die Herleitung. □

6 Vermeidung von Umgebungsbildung und Auffalten

Die abgeleiteten Regeln (Let) und (Rec) können noch optimiert werden.

(1) $((\textbf{let}\ id = e_1\ \textbf{in}\ e_2, [x : 5, y : 7, \ldots])$

↑ (Let)

(2) $(e_2, [id : (e_1, [x : 5, y : 7, \ldots]), x : 5, y : 7, \ldots])$

(...)

Die Regel (Let) kopiert die Umgebung der Closure zweimal, was in diesem Fall $[x : 5, y : 7, \ldots]$ ist. Es geht nun darum, dies zu vermeiden, indem für das innere Vorkommen der Umgebung ein Umgebungssymbol eingesetzt wird, was beim Nachschlagen eine besondere Bedeutung hat, sodass das äußere Vorkommen der Umgebung stattdessen erhalten wird.

Die Menge *Env* wird zunächst um eine neue Umgebung erweitert, die mit $\nearrow$ bezeichnet wird.

Um das Kopieren zu vermeiden, kann zunächst eine neue Regel (Let-$\nearrow$) eingeführt werden.

$$\text{(Let-}\nearrow\text{)} \quad \frac{(e_2, id : (e_1, \nearrow); \eta) \Downarrow cl}{(\textbf{let}\ id = e_1\ \textbf{in}\ e_2, \eta) \Downarrow cl}$$

Da $\nearrow$ nun eine besondere Umgebung ist, die nicht mehr mit (Id) behandelt werden kann, wird eine neue Regel (Id-$\nearrow$) definiert:

$$\text{(Id-}\nearrow\text{)} \quad \frac{(e, \eta) \Downarrow cl}{(id, id_0 : cl_0; id_1 : cl_1; \ldots; id_n : cl_n; id : (e, \nearrow); \eta \Downarrow cl)}$$

Da beim Erzeugen von $\nearrow$ das Muster $id : (e, \nearrow); \eta$ entstand, muss dieses bei (Id-$\nearrow$) in der Umgebung gesucht werden. Trifft man auf einen Identifier für dessen Eintrag die Closure als Umgebung $\nearrow$ hat, schlägt man die restliche Umgebung η hinter id nach. Diese Umgebung wäre die gewesen, die mit (Let) statt $\nearrow$ eingetragen worden wäre.

Beispiel:

(1) $(\textbf{let } x = 5 \textbf{ in let } y = 10 \textbf{ in } (*)\, x\, y, []) \Downarrow (50, [])$
(Let-$\nearrow$)
(2) $(\textbf{let } y = 10 \textbf{ in } (*)\, x\, y, [x : (5, \nearrow)]) \Downarrow (50, [])$
(Let-$\nearrow$)
(3) $((*)\, x\, y, [y : (10, \nearrow), x : (5, \nearrow)]) \Downarrow (50, [])$
(Op-2)
(4) $(* \, x, [y : (10, \nearrow), x : (5, \nearrow)]) \Downarrow ((*)\, 5, [])$
(Op-1)
(5) $((*), [y : (10, \nearrow), x : (5, \nearrow)]) \Downarrow ((*), [y : (10, \nearrow), x : (5, \nearrow)])$
(Val)
(6) $(x, [y : (10, \nearrow), x : (5, \nearrow)]) \Downarrow (5, [])$
(Id-$\nearrow$)
(7) $(5, []) \Downarrow (5, [])$
(Val)
(8) $(y : [y : (10, \nearrow), x : (5, \nearrow)]) \Downarrow (10, [x : (5, \nearrow)])$
(Id-$\nearrow$)
(9) $(10, [x : (5, \nearrow)]) \Downarrow (10, [x : (5, \nearrow)])$
(Val)

(Rec) kopiert ebenfalls unnötigerweise η zweimal. Darüber hinaus muss (Rec) immer dann angewendet werden, nachdem das zuvor von (Rec) eingetragene id per (Id) extrahiert wurde.

(1) $(id, id : (\textbf{rec } id.\ e, \eta); \eta) \Downarrow cl$
(Id)
(2) $(\textbf{rec } id.\ e, \eta) \Downarrow cl$
(Rec)
(3) $(e, id : (\textbf{rec } id.\ e, \eta)) \Downarrow cl$
...

Die Menge *Env*wird des Weiteren um $\nwarrow\!\!\searrow$ erweitert.
Mit der neuen Regel (Rec-$\nwarrow\!\!\searrow$) wird dann $(e, \nwarrow\!\!\searrow)$ statt (**rec** $id.\ e, \eta$) eingetragen.

$$\text{(Rec-}\nwarrow\!\!\searrow\text{)} \quad \frac{(e, id : (e, \nwarrow\!\!\searrow); \eta) \Downarrow cl}{(\textbf{rec}\ id.\ e, \eta) \Downarrow cl}$$

$\nwarrow\!\!\searrow$ kann ebenfalls nicht mehr mit (Id) behandelt werden. Es wird eine neue Regel (Id-$\nwarrow\!\!\searrow$) definiert, die nun (Unfold) und (Id) in einem Schritt macht.

$$\text{(Id-}\nwarrow\!\!\searrow\text{)} \quad \frac{(e, id : (e, \nwarrow\!\!\searrow); \eta) \Downarrow cl}{(id, id_0 : cl_0; id_1 : cl_1; \ldots; id_n : cl_n; id : (e, \nwarrow\!\!\searrow); \eta \Downarrow cl)}$$

Ähnlich wie bei (Id-$\nearrow$) enstand beim Anwenden von (Rec-$\nwarrow\!\!\searrow$) das Muster $id : (e, \nwarrow\!\!\searrow); \eta$. Falls also die Umgebung der nachzuschlagenden Closure $\nwarrow\!\!\searrow$ ist, wird die restliche Umgebung hinter *id* stattdessen nachgeschlagen. Das Auffalten ist ebenso in diesem Schritt enthalten, da **rec** *id.* *e* bereits zu *e* verkürzt wurde.

In Abbildung 1 soll die neue Semantik an der Fakultätsfunktion verdeutlicht werden. (Let) wird benutzt, um nicht beide neue Konzepte auf einmal zu präsentieren.

Im Folgenden soll die Korrektheit der neuen Semantik bewiesen werden. Da zwei neue Umgebungen hinzugekommen sind, wird zunächst die Menge der neuen Umgebungen, und dann eine Relation zwischen der Menge der alten und der Menge der neuen Umgebungen definiert.

Definition: 9. $Env_{\nwarrow\!\!\searrow\nearrow} := Env \cup \{\nwarrow\!\!\searrow, \nearrow\}$

Definition: 10. $R \subseteq Env \times Env_{\nwarrow\!\!\searrow\nearrow}$

- $([], []) \in R$
- $(\eta_1, \eta_1'), (\eta_2, \eta_2') \in R \Rightarrow (id : (e, \eta_1); \eta_2, id : (e, \eta_1'); \eta_2') \in R$
- $(\eta, \eta') \in R \Rightarrow (id : (\textbf{rec}\ id.\ e, \eta); \eta, id : (e, \nwarrow\!\!\searrow); \eta) \in R$
- $(\eta, \eta') \in R \Rightarrow (id : (e, \eta); \eta, id : (e, \nearrow); \eta) \in R$

Um eine Umgebung aus *Env* aus einer Umgebung aus $Env_{\nwarrow\!\!\searrow\nearrow}$ zu erzeugen, wird folgende Funktion definiert:

Definition: 11. $replace : Env_{\nwarrow\!\!\searrow\nearrow} \to Env$

- $replace([]) = []$
- $replace(id : (e, \eta); \eta_r) = id : (e, \eta); replace(\eta_r)$
- $replace(id : (e, \nearrow); \eta_r) = id : (e, replace(\eta_r)); replace(\eta_r)$
- $replace(id : (e, \nwarrow\!\!\searrow); \eta_r) = id : (\textbf{rec}\ id.\ e, replace(\eta_r)); replace(\eta_r)$

Satz 6 (Korrektheit von (Let-$\nearrow$), (Id-$\nearrow$) (Rec-$\nwarrow\!\!\searrow$) und (Id-$\nwarrow\!\!\searrow$))**.** *Sei* $(e, \eta') \in Cl_{\nwarrow\!\!\searrow\nearrow}, (e, \eta) \in Cl$ *mit* $(\eta, \eta') \in R$ *und* $(e, \eta') \Downarrow (v, \hat{\eta}')$*, dann existiert* $\hat{\eta}$ *mit* $(\hat{\eta}, \hat{\eta}') \in R$*, sodass* $(e, \eta) \Downarrow (v, \hat{\eta})$

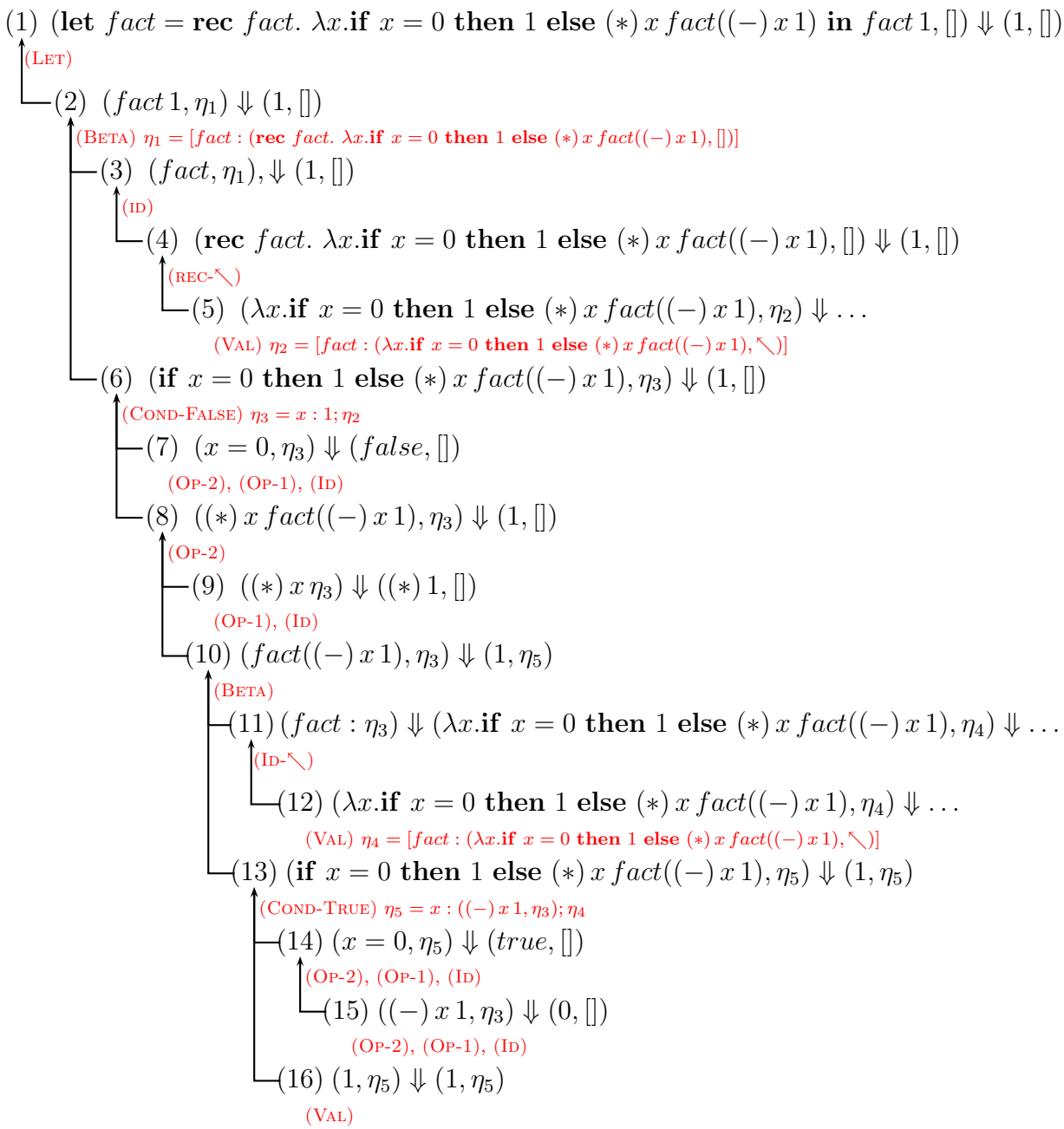


Abbildung 1: Fakultätsfunktion mit ↗-Regeln

Proof. Induktion über die Länge der Herleitung und Fallunterscheidung der zuletzt angewandten Regel.

- (Val) Es gilt $(v, \eta') \Downarrow (v, \eta')$. Dann wähle $\eta = replace(\eta')$. Alle Vorkommen von $id : (e, \nearrow); \eta_r$ werden also durch $id : (e, \eta'_r); \eta'_r$ und alle $id : (e, \nwarrow); \eta_r$ durch $id : (\textbf{rec}\ id.\ e, \eta'_r); \eta'_r$ ersetzt. Somit gilt $(\eta', \eta) \in R$ und es folgt $(v, \eta) \Downarrow (v, \eta)$.

- (Id) Es gilt $(id, \eta') \Downarrow (v, \hat{\eta}')$. Dies kann nur gelten, wenn $\eta' = id_0 : cl_0; \ldots; id_n : cl_n, id : (e, \eta''); \eta_r$ und $(e, \eta'') \Downarrow (v, \hat{\eta}')$. Wähle $\eta = replace(\eta')$. Dann existiert nach I.V. η''' mit $(\eta''', \eta'') \in R$ und $\hat{\eta}$ mit $(\hat{\eta}, \hat{\eta}') \in R$. Mit (Id) folgt dann $(id, \eta) \Downarrow (v, \hat{\eta})$.

- (Id-$\nearrow$) Sei $\eta' = id_0 : cl_0; \ldots; id_n : cl_n; id : (e, \nearrow); \eta_r$ und es gelte $(id, \eta') \Downarrow (v, \hat{\eta}')$. Dies kann nur mit Regel (Id-$\nearrow$) und einer Prämisse der Form $(e, \eta_r) \Downarrow (v, \hat{\eta}')$ gefolgt sein. Dann wähle $\eta = id_0 : cl_0; \ldots; id_n : cl_n; id : (e, \eta_r); \eta_r,$. Somit gilt $(\eta, \eta') \in R$ nach Definition. Nach I.V. existiert $\hat{\eta}$, sodass $(\hat{\eta}, \eta_r) \in R$ und $(e, \eta_r) \Downarrow (v, \hat{\eta})$. Mit (Id) folgt dann $(id, \eta) \Downarrow (v, \hat{\eta})$.

- (Id-$\nwarrow$) Sei $\eta' = id_0 : cl_0; \ldots; id_n : cl_n; id : (e, \nwarrow); \eta_r$ und es gelte $(id, \eta') \Downarrow (v, \hat{\eta}')$. Dies kann nur mit Regel (Id-$\nwarrow$) und einer Prämisse der Form $(e, id : (e, \nwarrow); \eta_r) \Downarrow (v, \hat{\eta}')$ gefolgt sein. Dann wähle $\eta = id_0 : cl_0; \ldots; id_n : cl_n, id : (\textbf{rec}\ id.\ e, \eta_r); \eta_r$. Somit gilt $(\eta, \eta') \in R$ nach Definition. $(id, \eta) \Downarrow (v, \hat{\eta})$ für ein bestimmtes $\hat{\eta}$ kann nur mit (Id) gelten. (Id) bedingt, dass $\eta(id) = (\textbf{rec}\ id.\ e, \eta_r) \Downarrow (v, \hat{\eta})$. Dies kann nur mit (Rec) gelten, was bedingt, dass $(e, id : (\textbf{rec}\ id.\ e, \eta_r); \eta_r) \Downarrow (v, \hat{\eta})$. Da $(id : (\textbf{rec}\ id.\ e, \eta_r); \eta_r, id : (e, \nwarrow); \eta_r) \in R$, existiert nach I.V. $\hat{\eta}$ mit $(\hat{\eta}, \hat{\eta}') \in R$ und es gilt $(id, \eta) \Downarrow (v, \hat{\eta})$.

- (Rec-$\nwarrow$) Es gilt $(\textbf{rec}\ id.\ e, \eta') \Downarrow (v, \hat{\eta}')$ mit (Rec-$\nwarrow$). Dies kann nur aus der Prämisse $(e, id : (e, \nwarrow); \eta') \Downarrow (v, \hat{\eta}')$ folgen. Da $(id : (\textbf{rec}\ id.\ e, \eta'); \eta', id : (e, \nwarrow); \eta') \in R$, existiert nach I.V. $\hat{\eta}$ mit $(\hat{\eta}, \hat{\eta}') \in R$ und $(e, id : (\textbf{rec}\ id.\ e, \eta'); \eta') \Downarrow (v, \hat{\eta})$. Wähle $\eta = replace(\eta')$ und mit (Rec) folgt $(\textbf{rec}\ id.\ e, \eta) \Downarrow (v, \hat{\eta})$.

- (Let-$\nearrow$) Es gilt $(\textbf{let}\ id = e_1\ \textbf{in}\ e_2, \eta') \Downarrow (v, \hat{\eta}')$ mit (Let-$\nearrow$). Dies kann nur aus der Prämisse $(e_2, id : (e_1, \nearrow); \eta') \Downarrow (v, \hat{\eta}')$ folgen. Da $(id : (e_1, \eta'); \eta', (id : (e_1, \nearrow); \eta') \in R$, existiert nach I.V. $\hat{\eta}$ mit $(\hat{\eta}, \hat{\eta}') \in R$ und $(e_2, id : (e_1, \eta'); \eta') \Downarrow (v, \hat{\eta})$. Wähle $\eta = replace(\eta')$ und mit (Let) folgt $(\textbf{let}\ id = e_1\ \textbf{in}\ e_2, \eta) \Downarrow (v, \hat{\eta})$.

- Andere Regeln trivial

□

Vollständigkeit folgt ähnlich, ist aber weniger interessant, da es sich um eine Übersetzung handelt.

7 Infixoperatoren

Eine Semantik, die nur eine Infixoperatorsyntax zulässt, ist zur Codegenerierung wünschenswert, wie in [Uhr09] beschrieben. Um dies zu erreichen, müssen zunächst die Regeln (Op-1) und (Op-2) entfernt werden. Ebenso ist $op\, z$ kein Wert mehr, da dies für die neue Regel nicht mehr nötig ist.

$$v \quad ::= \quad c \mid \lambda id.\, e$$

Folgende neue Regel wird definiert, um (Op-1) und (Op-2) abzulösen.

$$(\text{Op}) \quad \frac{(e_1, \eta) \Downarrow (z_1, \eta') \quad (e_2, \eta) \Downarrow (z_2, \eta'') \quad op^I(z_1, z_2) = z}{(e_1\ op\ e_2, \eta) \Downarrow (z, [])}$$

Es folgt ein Beispiel zur Anwendung von (Op).

(1) $((\lambda x.x * x)\, 5, [])$
(Beta)
(2) $(\lambda x.x * x, []) \Downarrow (\lambda x.x * x, [])$
(Val)
(3) $(x * x, [x : 5]) \Downarrow (25, [])$
(Op)
(4) $(x, [x : 5]) \Downarrow (5, [])$
(Id)
(5) $(5, []) \Downarrow (5, [])$
(Val)
(6) $(x, [x : 5]) \Downarrow (5, [])$
(Id)
(7) $(5, []) \Downarrow (5, [])$
(Val)

Satz 7 (Korrektheit der Infixnotation)**.** *Wenn* $(e_1\ op\ e_2, \eta) \Downarrow (v, [])$*, dann gilt* $(op\ e_1 e_2, \eta) \Downarrow (v, [])$.

Proof. Der Beweis wird per Induktion über die Länge der Herleitung geführt. Es gelte $(e_1\ op\ e_2, \eta) \Downarrow (v, [])$. Dies kann nur aus den Prämissen $(e_1, \eta) \Downarrow (z_1, \eta')$ und $(e_2, \eta) \Downarrow (z_2, \eta'')$ gefolgt sein. Nach I.V. existieren bereits Herleitungen für die Prämissen in der Präfixoperatorsemantik. Betrachte $(op\ e_1\ e_2, \eta)$. Dies kann nur aus der Regel (Op-2) mit folgenden Prämissen folgen: $(op\ e_1, \eta) \Downarrow (op\ z_1, \eta')$ und $(e_2, \eta) \Downarrow (z_2, \eta'')$. Nach Voraussetzung existiert eine Herleitung der zweiten Prämisse. Die Herleitung der ersten Prämisse kann nur aus (Op-1) gefolgt sein. Dies bedingt die Prämissen $(op, \eta) \Downarrow (op, \eta)$,

was gültig ist mit (VAL), und $(e_1, \eta) \Downarrow (z_1, \eta')$, was ebenfalls nach Voraussetzung gültig ist. Somit folgt auch, dass $(op\ e_1\ e_2) \Downarrow (v, [])$. □

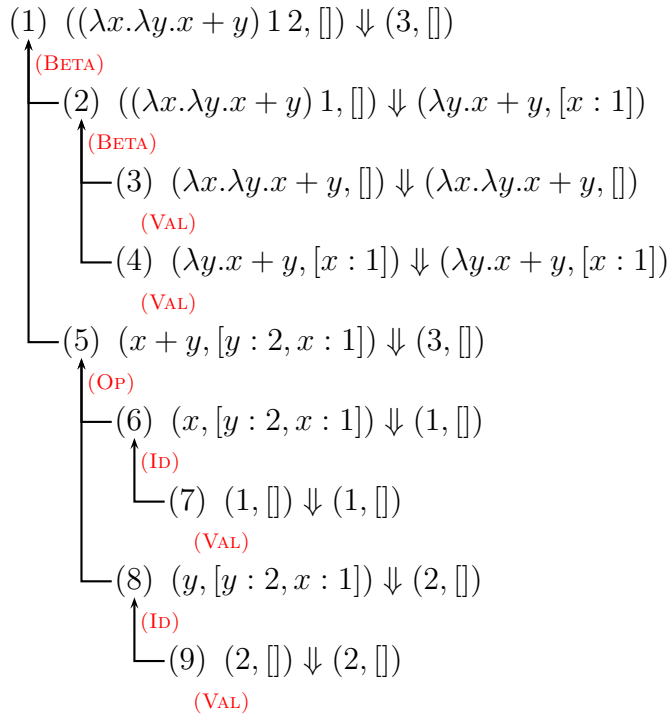
Zum Beweis der Vollständigkeit wird folgende Übersetzung der Präfixnotation in die Infixnotation definiert:

Definition: 12. $(op) \equiv \lambda x.\lambda y.x\ op\ y$

Satz 8 (Vollständigkeit der Infixnotation)**.** *Wenn* $(op\, e_1\, e_2, \eta) \Downarrow (v, [])$*, dann gilt* $((op)\, e_1\, e_2) \Downarrow (v, [])$ *mit der definierten Übersetzung.*

Proof. Gelte $(op\, e_1\, e_2, \eta) \Downarrow (v, [])$ in der Präfixnotation. Zu zeigen ist nun, dass $((\lambda x.\lambda y.x\ op\ y)\, e_1\, e_2, \eta) \Downarrow (v, [])$ herleitbar ist. Nach Anwenden von (BETA), (OP) und zweimal (ID), Hinzunahme der Voraussetzungen für (OP-2) und (OP-1) erhält man letztendlich die Herleitung. □

Es folgt ein Beispiel zur Umwandlung von Präfixnotation in Infixnotation.



8 Call-by-value

Bisher wurde eine call-by-name-Semantik betrachtet, d.h. Argumente werden nicht erst zu Werten ausgewertet, sondern direkt in eine Umgebung eingetragen. Insgesamt müssen

zwei Regeln angepasst werden, um eine call-by-value-Semantik zu erhalten. Die erste von ihnen ist (Beta), die zu (Beta-v) wird. Es wird eine weitere Prämisse gefordert, sodass das Argument zuerst ausgewertet wird:

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

(Unfold) muss ebenfalls geändert werden. Eine naive Änderung ist aber nicht möglich, denn (Unfold) würde mit einer Call-by-value-Semantik einfach divergieren, da gefordert werden müsste, dass der rechte Teil der Applikation $e(\mathbf{fix}\, e)$ zunächst ausgewertet wird.

(1) $(\mathbf{fix}\ e, \eta) \Downarrow cl$
(Unfold)
(2) $(e\ (\mathbf{fix}\ e), \eta') \Downarrow cl$
(Unfold)
(3) $(e\ (e\ (\mathbf{fix}\ e)), \eta'') \Downarrow cl$
(Unfold)

Eine Möglichkeit, dieses Problem zu umgehen, ist, nur noch (Rec) als weniger mächtiges Konzept zuzulassen. In diesem Fall würde man **fix** aus der Menge der Konstanten entfernen, und nur noch rec-Ausdrücke zulassen.

$$\begin{array}{lcl} c & ::= & b \mid z \mid op \\ e & ::= & c \mid id \mid \lambda id.\, e \mid e_1\, e_2 \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{rec}\ id.\ e \end{array}$$

(Rec) wäre in diesem Fall keine abgeleitete Regel mehr.

Eine weitere Möglichkeit besteht darin, die Regel (Unfold) so abzuändern, dass eine Lambda-Abstraktion auf der rechten Seite einer Applikation mit **fix** gefordert wird.

$$\text{(Unfold-v)} \quad \frac{(e_1, \eta) \Downarrow (\mathbf{fix}, \eta') \quad (e_2, \eta) \Downarrow (\lambda id.e, \eta'') \quad (e, id : (\mathbf{fix}\, e_2, \eta); \eta'') \Downarrow cl}{(e_1 e_2, \eta) \Downarrow cl}$$

Satz 9 (Korrektheit von (Unfold-V))**.** *Gelte* $(e_1 e_2, \eta) \Downarrow cl$ *wegen* (Unfold-v) *in der call-by-value-Semantik, so auch* $(e_1 e_2, \eta) \Downarrow cl$ *wegen* (Unfold) *in der call-by-name-Semantik.*

Proof. Der Beweis wird per Induktion über die Länge der Herleitung geführt. Da $(e_1 e_2, \eta) \Downarrow cl$ mit (Unfold-v) gilt, kann dies nur aus den Prämissen der Form $(e_1, \eta) \Downarrow (\mathbf{fix}, \eta')$, $(e_2, \eta) \Downarrow (\lambda id.e, \eta'')$ und $(e, id : (\mathbf{fix}\, e_2, \eta); \eta'') \Downarrow cl$ gefolgt sein.

(Unfold) bedingt, dass $(e_1, \eta) \Downarrow (\mathbf{fix}, \eta')$, was bereits nach Voraussetzung gilt, und $(e_2(\mathbf{fix}\, e_2)) \Downarrow cl$. Nach Voraussetzung gilt ebenso, dass $(e_2, \eta) \Downarrow (\lambda id.e, \eta'')$. Somit kann nur Regel (Beta) angewandt worden sein, die des Weiteren bedingt, dass $(e, id : (\mathbf{fix}\, e_2, \eta); \eta'') \Downarrow cl$, was ebenfalls nach Voraussetzung gültig ist.

□

9 Weitere Optimierungsmöglichkeit

Die hier definierte Umgebungssematik ist, wie bereits beschrieben, schon besser geeignet, um einen Interpreter zu implementieren. Sie birgt allerdings noch ineffizientes Verhalten, das vermieden werden kann: Wird ein Identifier in einer Umgebung nachgeschlagen, ist eine lineare Suche nötig. $(d, [a : cl_0, b : cl_1, c : cl_2, d : cl_3])$. In diesem Beispiel wird nach d in der Umgebung gesucht, was bedeutet, dass zunächst d mit a verglichen wird, dann mit b, usw, was in der Komplextitätsklasse $O(n)$ liegt, falls die Länge der Umgebung n ist. Darüberhinaus liegt der Vergleich zweier Identifier id_0, id_1 in $O(m)$, falls $m = min\{len(id_0), len(id_1)\}$. Dies ergibt eine Gesamtkomplexität von $O(n * m)$. Die lineare Suche und das Vergleichen können beide in konstanter Komplexität erreicht werden, indem Identifier durch so genannte De'Bruijn-Indizes ersetzt werden [Sie09]. Beispiel: $(\lambda x.\lambda y.(+)\, x\, y, [])$ wird zu $(\lambda.\lambda.(+)\, \underline{1}\, \underline{0}, [])$. Die Intuition ist, dass x zuerst eingetragen wird und y danach. Somit ergibt sich eine Umgebung, in der x auf Index 1 und y auf Index 0 steht. Zur Laufzeit kann dann per Index in konstanter Zeit darauf zugegriffen werden. Die Ausarbeitung dessen ist allerdings Thema von [Meu09].

10 Zusammenfassung

Im Rückblick wurden folgende Themen behandelt:

- Definition einer Substitutionssemantik mit call-by-value
- Definition einer Substitutionssemantik mit call-by-name
- Definition einer Umgebungssemantik: Die Umgebungssemantik ist als Implementierung besser geeignet, da sie das ineffiziente Substituieren der Substitutionssemantik beseitigt
- Die Umgebungssemantik ist korrekt bezüglich der Substitutionssemantik
- Die Umgebungssemantik ist vermutlich nicht vollständig bezüglich der Substitutionssemantik. Gegebenenfalls müssen die Sätze oder Lemmata anders formuliert werden, um den Zusammenhang herzustellen
- Die Umgebungssemantik mit call-by-name nutzt den Fixpunktoperator. **rec** *id*. *e* kann als syntaktischer Zucker und (Rec) als abgeleitete Regel eingeführt werden
- Ebenso können **let** $id = e_1$ **in** e_2 als syntaktischer Zucker und (Let) als abgeleitete Regel eingeführt werden
- Optimierung von (Let). Da es eine Umgebung zweimal produziert, wird eine von ihnen durch ein Umgebungssymbol ersetzt, was mit (Let-↗) geschieht. Mit (Id-↗) kann der neue Fall für *id* behandelt werden.

- Optimierung (Rec). Es produziert eine Umgebung ebenfalls zweimal, und fordert ein wiederholtes Auffalten. Mit einem weiteren Umgebungssymbol für eine Umgebung, (Rec-↖) und (Id-↖) wurde dieses ineffiziente Verhalten ebenfalls beseitigt.
- Definition einer Semantik mit Infixnotation. Diese kann zur leichteren Implementierung benutzt werden.
- Definition einer call-by-value-Semantik. (Beta) wurde durch (Beta-v) und (Unfold) durch (Unfold-v) ersetzt.
- Ausblick: Weitere Optimierung durch De'Bruijn-Indizes, sodass Identifier in konstanter Zeit nachgeschlagen werden können.

Literatur

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

[Sie09] Sieber, Kurt: *Theorie der Programmierung I*. 2009

[Uhr09] Uhrhan, Christian: *Codegenerierung*. 2009. – Seminar Umgebungssemantiken