Universität Siegen
Fachbereich 12 - Informatik und Elektrotechnik
Institut für Programmiersprachen

WS 2008/2009

Seminar: Umgebungssemantiken

De Bruijn Indizes

Simon Meurer

Mat.Nr.: 731120

17. April 2009

Betreuer: Benedikt Meurer

Inhaltsverzeichnis

Einleitung

In [Reh09] wird gezeigt, wie man von einer Substitutionssemantik zu einer Umgebungssemantik gelangt. Der Vorteil einer Umgebungssemantik liegt darin, dass man sich Substitutionen zur Laufzeit sparen kann. Eine Substitution hat lineare Laufzeit in der Größe des Ausdrucks. Da diese nun wegfällt ist die Implementierung viel schneller geworden.

Nun ist die Umgebungssemantik, so wie sie bislang definiert wurde immer noch ineffizient: Variablen *id* werden über ihren Namen (Zeichenketten) identifiziert. Daraus ergeben sich zwei Probleme.

Erstens: *Namen sind nicht eindeutig.*

Es gibt mehrere Möglichkeiten dieses Problem zu lösen. Eine mögliche Lösung wäre doppelte Variablennamen bei der Substitution umzubenennen.(Gebundene Umbennenung, [Sie08]) Andere Möglichkeiten werden in [Pie02] vorgestellt.

Das zweite Problem ist, dass in der Umgebungssemantik die Closures (siehe Kapitel 1) in den Umgebungen über ebendiese Namen identifiziert werden. Dies ist in zweierlei Hinsicht ineffizient. Zum einen muss ein Stringvergleich vorgenommen werden, was schon von Natur aus relativ ineffizient ist, und zum anderen hat man wieder lineare Laufzeit bis man überhaupt die richtige Closure gefunden hat. Um dieses Problem und damit implizit das erste Problem zu lösen, wäre es also nötig die Namen in irgendeiner Art ganz weg zu bekommen.

Damit kommen wir zum niederländischen Mathematiker Nicolaas Govert de Bruijn. Dieser hatte die Idee die Namen durch Zahlen, sog. *De Bruijn-Indizes*, zu ersetzen. Diese Indizes stehen dann für die „Entfernung“ zum entsprechenden Punkt, wo die Variable gebunden wurde.

Um dieses Prinzip in eine formal korrekte Form zu bringen soll in dieser Arbeit zunächst die Umgebungssemantik aus [Reh09] vorgestellt werden. Danach soll eine Semantik mit De Bruijn-Indizes, im weiteren De Bruijn-Semantik genannt, definiert werden. Anschließend soll ein Zusammenhang zwischen den beiden Semantiken in Form einer Übersetzungsfunktion definiert werden. Zum Schluss wird eine alternative Umsetzung einer De Bruijn-Semantik, wie in [Pie02] vorgeschlagen, vorgestellt und mit Unserer verglichen.

1 Umgebungssemantik

In diesem Abschnitt soll die Umgebungssemantik vorgestellt werden, von welcher ausgegangen wird, um zur De Bruijn-Semantik zu gelangen. Bei dieser handelt es sich um eine Call-By-Name Semantik mit Konstanten, wie in [Sie08] und [Reh09] definiert.

Zunächst wird die Syntax der Sprache vorgestellt.

Definition 1.1 (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:

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

Da es sich um eine Umgebungssemantik handelt, werden nun noch Closures und Umgebungen hinzugefügt. In einer Umgebung wird der Ausdruck mit der zugehörigen Umgebung eingetragen. Der Ausdruck mit der Umgebung ist eine Closure. Die Closures werden in der Umgebung über Namen id identifiziert und links an die Umgebung angehängt.

Definition 1.2 (Closures und Umgebungen)
Die Mengen *Env* aller *(Laufzeit-)Umgebungen* η und *Cl* aller *Closures* cl sind durch die folgende kontextfreie Grammatik definiert:

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

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

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

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 $(id_1 : cl_1; \ldots; id_n : cl_n; [\,]) \in \mathit{Env}$ schreiben wir

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

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

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

Nun folgen die Regeln zur Auswertung. Es handelt sich um big step Regeln, welche auf Closures definiert werden, da wir eine Umgebungssemantik betrachten.

Definition 1.3 (Big step Regeln) Ein *big step* in der Umgebungssemantik ist eine Formel der Gestalt $(e, \eta) \Downarrow (e', \eta')$, wobei $e, e' \in \mathit{Exp}$ und $\eta, \eta' \in \mathit{Env}$. 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) $$\frac{(e_0, \eta) \Downarrow (\mathit{true}, \eta_0) \quad (e_1, \eta) \Downarrow cl}{(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \eta) \Downarrow cl}$$

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

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

2 De Bruijn Indizes

Unser Ziel ist, wie am Anfang bereits angedeutet, die Namen aus der Sprache zu entfernen. Dazu werden sogenannte De Bruijn Indizes eingeführt.

Die grundsätzliche Idee ist, dass anstelle des Namens id, die Entfernung zur jeweiligen λ-Abstraktion verwendet wird.

Beispiel 2.1 $\lambda x.\, \lambda y.\, +\, x\, y \mathrel{\hat{=}} \lambda.\, \lambda.\, +\, \underline{2}\, \underline{1}$

Hier wurde für x der Index $\underline{2}$ eingesetzt, da die Entfernung zum definierenden λ 2 beträgt. (Wir beginnen bei 1 zu zählen um nicht in die Gefahr zu kommen negative Indizes zu bekommen.)

Man kann auch anstatt von innen nach außen zu gehen, von außen nach innen gehen.

Beispiel 2.2 $\lambda x.\, \lambda y.\, +\, x\, y \mathrel{\hat{=}} \lambda.\, \lambda.\, +\, \underline{1}\, \underline{2}$

Hier wurde für x der Index $\underline{1}$ eingesetzt, da x zuerst definiert wurde und danach erst y, das den Index $\underline{2}$ bekommt. Diese Möglichkeit wollen wir hier nicht weiter betrachten. Nähere Informationen dazu finden sich in [Pie02].

2.1 Die De Bruijn Semantik

Um nun aus der Umgebungssemantik, die im vorherigen Kapitel vorgestellt wurde eine Semantik ohne Namen zu machen, müssen wir zunächst eine neue Sprache einführen, die De Bruijn Indizes beinhaltet.

Definition 2.1 (Expressions) Vorgeben sei eine (unendliche) Menge $\{\underline{1}, \underline{2}, \ldots\}$ von *De Bruijn-Indizes* $\underline{i}$. Die Mengen *dbExp* aller *De Bruijn-Ausdrücke* e und *dbVal* aller *De Bruijn-Werte* sind durch die kontextfreie Grammatik

$$\begin{array}{lcl} e & ::= & c \mid \underline{i} \mid \lambda.\, e \mid e_1\, e_2 \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \\ v & ::= & c \mid op\, z \mid \lambda.\, e \end{array}$$

definiert. Der *Rang* $rank\,(e) \in \mathbb{N}$ eines De Bruijn-Ausdrucks e ist wie folgt induktiv definiert:

$$\begin{array}{rcl} rank\,(c) & = & 0 \\ rank\,(\underline{i}) & = & \underline{i} \\ rank\,(\lambda.\, e) & = & \max(rank\,(e), 1) - 1 \\ rank\,(e_1\, e_2) & = & \max(rank\,(e_1), rank\,(e_2)) \\ rank\,(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) & = & \max(rank\,(e_0), rank\,(e_1), rank\,(e_2)) \end{array}$$

Außerdem müssen die Closures und Umgebungen etwas anders, als bei der Umgebungssemantik definiert werden, da nun keine Namen mehr vorkommen.

Definition 2.2 (Closures und Umgebungen) Die Menge *dbEnv* aller *De Bruijn-(Laufzeit-)Umgebungen* η und die Menge *dbCl* aller *De Bruijn-Closures* cl sind durch die folgende kontextfreie Grammatik definiert:

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

Die *Länge* $len\,(\eta)$ einer De Bruijn-Umgebung η ist wie folgt induktiv definiert:

$$\begin{aligned} len\,([\,]) &= 0 \\ len\,(cl; \eta) &= 1 + len\,(\eta) \end{aligned}$$

Eine De Bruijn-Closure $cl = (e, \eta)$ heißt *gültig*, wenn $rank\,(e) \leq len\,(\eta)$ und η gültig ist. Eine De Bruijn-Umgebung η heißt *gültig*, wenn alle eingetragenen Closures gültig sind.

Darauf aufbauen werden nun big step Regeln aufgestellt, die ohne Namen auskommen.

Die im Folgenden vorgestellte Definition der big step Regeln ähnelt den big step Regeln der Umgebungssemantik.

Der Unterschied wird sich in der (Beta)- und der (Id)- bzw. (Index)-Regel zeigen. Bei (Beta) wird jetzt nicht mehr der Name an die Closure angehängt und bei (Index) wird jetzt anstelle des *id*bei (Id) ein De Bruijn Index erwartet und die Closure an der Stelle i ausgewertet.

Definition 2.3 (Big step Regeln) Ein *big step* in der De Bruijn-Umgebungssemantik ist eine Formel der Gestalt $(e, \eta) \Downarrow (e', \eta')$, wobei $e, e' \in dbExp$ und $\eta, \eta' \in dbEnv$. Ein derartiger big step heißt *gültig*, wenn er sich mit den folgenden Regeln herleiten lässt:

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

(Index) $$\frac{\eta(\underline{i}) \Downarrow cl}{(\underline{i}, \eta) \Downarrow cl}$$

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

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

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

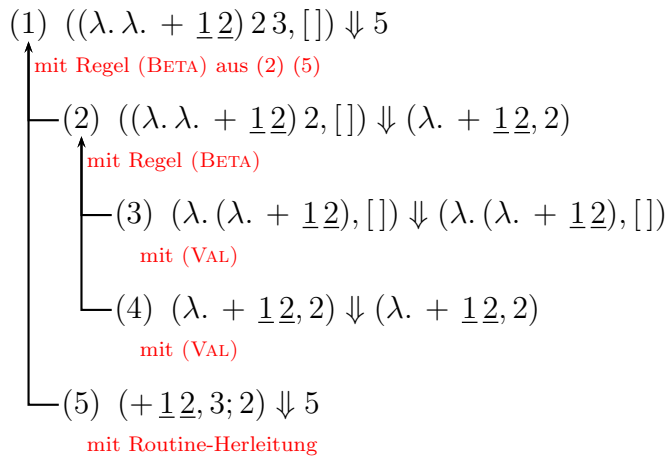

Abbildung 1: Beispiel für einen big step-Beweisbaum

Als erstes ist für diese neue De Bruijn-Umgebungssemantik zu zeigen, dass sie wohldefiniert ist, das heißt, dass wir aus einer gültigen Closure mit den big step Regeln keine nicht gültigen Closures bilden können.

Lemma 2.1 (Wohldefiniertheit der De Bruijn-Umgebungssemantik)
Seien $(e,\eta),(\hat{e},\hat{\eta}) \in dbCl$. *Wenn* (e,η) *gültig und* $(e,\eta) \Downarrow (\hat{e},\hat{\eta})$, *dann ist auch* $(\hat{e},\hat{\eta})$ *gültig und es gilt* $\hat{e} \in dbVal$.

Beweis: Induktion über die Länge der Herleitung von $(e,\eta) \Downarrow (\hat{e},\hat{\eta})$ und Fallunterscheidung nach der zuletzt angewendeten Big Step Regel.

- (VAL) (v,η) ist nach Voraussetzung gültig und $v \in dbVal$.
- (INDEX) $(\underline{i},\eta)$ ist nach Voraussetzung gültig und damit auch $\eta(\underline{i})$ dann ist mit Induktionsvoraussetzung $cl = (e',\eta')$ gültig und $e' \in dbVal$.
- (BETA) $(e_1 e_2,\eta)$ ist nach Voraussetzung gültig, ebenso (e_1,η). Dann ist wegen Induktionsvoraussetzung $(\lambda.e,\eta_1)$ gültig. Da (e_2,η) nach Voraussetzung gültig ist, ist auch $(e,(e_2,\eta);\eta_1)$ gültig, da $rank\,(e) <= rank\,(\lambda.e) + 1 \leq len\,(\eta_1) + 1 = len\,((e_2,\eta);\eta_1)$ dann ist nach Induktionsvoraussetzung $cl = (e',\eta')$ gültig und $e' \in dbVal$.
- (OP-1) $op\,z \in dbVal$ und $(op\,z,[\,])$ gültig, da $rank\,(op\,z) = len\,([\,]) = 0$.
- (OP-2) $z \in dbVal$ und $(z,[\,])$ gültig, da $rank\,(z) = len\,([\,]) = 0$.
- (COND-TRUE) (**if** e_0 **then** e_1 **else** e_2,η) ist nach Voraussetzung gültig, damit ist auch (e_1,η) gültig. Nach Induktionsvoraussetzung ist dann $cl = (e',\eta')$ gültig und $e' \in dbVal$.

- (Cond-False) analog zu (Cond-True).
- (Unfold) $(e_1\, e_2, \eta)$ ist gültig nach Voraussetzung, dadurch ist auch (e_2, η) gültig, da $rank\,(fix\, e_2) = rank\,(e_2)$. Insgesamt ist $(e_2(fix\, e_2), \eta)$ gültig und damit folgt mit Induktionsvoraussetzung $cl = (e', \eta')$ gültig und $e' \in dbVal$.

Wie zuvor betrachten wir nun nur noch gültige De Bruijn-Closures und -Umgebungen, d.h. wir nehmen an, *dbCl* und *dbEnv* enthalten nur noch gültige Elemente.

2.2 Call-By-Value

Bislang haben wir eine Call-By-Name Semantik betrachtet. Jetzt zeigen wir, dass eine äquivalente Call-By-Value Semantik existiert und dass dafür nicht viele Änderungen gemacht werden müssen.

Zunächst muss die (Beta)-Regel in eine (Beta-V)-Regel umgebaut werden. Dies erfolgt relativ einfach, indem man das Argument ändert.

$$\text{(Beta-V)} \quad \frac{(e_1, \eta) \Downarrow (\lambda.\, e, \eta_1) \quad (e_2, \eta) \Downarrow (v, \eta_2) \quad (e, (v, \eta_2); \eta_1) \Downarrow cl}{(e_1\, e_2, \eta) \Downarrow cl}$$

Als nächstes muss die (Fix)-Regel geändert werden, da Ausdrücke der Form $\mathbf{fix}\, \lambda.\, e$ wegen der neuen (Beta-V) divergieren würden, da immer zuerst ausgewertet wird. Also muss eine neue (Fix)-Regel eingeführt werden, die trotz Call-By-Value, einen unausgewerteten Ausdruck in die Umgebung einsetzt.

$$\text{(Fix-V)} \quad \frac{(e_1, \eta) \Downarrow (\mathbf{fix}, \eta_1) \quad (e_2, \eta) \Downarrow (\lambda.\, e, \eta_2) \quad (e, (\mathbf{fix}\, \lambda.\, e, \eta_2); \eta_2) \Downarrow cl}{(e_1\, e_2, \eta) \Downarrow cl}$$

Im weiteren Verlauf betrachten wir aber wieder die Call-By-Name Semantik, da sie etwas einfacher ist.

2.3 Syntaktischer Zucker

Bislang haben wir eine relativ kleine Syntax betrachtet. Nun sollen einige Konstrukte als syntaktischer Zucker eingeführt werden.

2.3.1 let

Zunächst muss die Menge der De Bruijn-Ausrücke *dbExp* erweitert werden um:

$$e \;::=\; \mathbf{let}\; e_1\; \mathbf{in}\; e_2$$

Zusätzlich muss eine neue big step Regel eingeführt werden:

$$\text{(Let)} \quad \frac{(e_2, (e_1, \eta); \eta) \Downarrow cl}{(\mathbf{let}\; e_1\; \mathbf{in}\; e_2, \eta) \Downarrow cl}$$

Damit kann **let** e_1 **in** e_2 als syntaktischer Zucker für $\lambda.\, e_2\, e_1$ aufgefasst werden. Wie der Beweisbaum in Abbildung 2 zeigt.

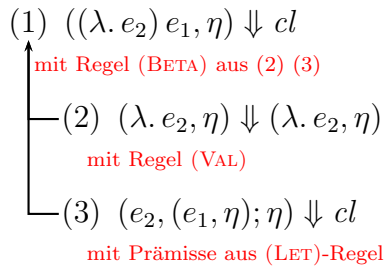

Abbildung 2: Beweisbaum für $((\lambda.\, e_2)\, e_1, \eta)$

2.3.2 rec

Die Menge der De Bruijn-Ausdrücke *dbExp* muss erweitert werden.

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

Zusätzlich muss eine neue big step Regel eingeführt werden:

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

rec. e kann als syntaktischer Zucker für $\mathbf{fix}\, \lambda.\, e$.(Siehe Abbildung 3)

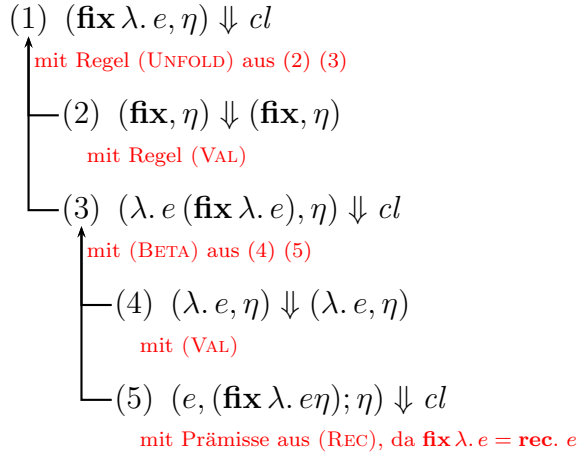

Abbildung 3: Beweisbaum für $(\mathbf{fix}\, \lambda.\, e, \eta)$

2.3.3 And, Or, ...

Als weiteres Beispiel für syntaktischen Zucker führen wir logische Operatoren ein. Um den Rahmen aber nicht zu überspannen, soll hier nur der AND-Operator gezeigt werden.

Zunächst muss hierzu wieder die Menge der De Bruijn-Ausrücke *dbExp* erweitert werden.

$$e \quad ::= \quad e_1 \,\&\&\, e_2$$

Dann werden zwei zusätzliche big step Regeln eingeführt.

$$\text{(And-True)} \quad \frac{(e_1, \eta) \Downarrow (\mathit{true}, \eta_1) \quad (e_2, \eta) \Downarrow \mathit{cl}}{(e_1 \,\&\&\, e_2, \eta) \Downarrow \mathit{cl}}$$

$$\text{(And-False)} \quad \frac{(e_1, \eta) \Downarrow (\mathit{false}, \eta_1)}{(e_1 \,\&\&\, e_2, \eta) \Downarrow \mathit{false}}$$

Insgesamt kann man $e_1 \,\&\&\, e_2$ als syntaktischen Zucker für **if** e_1 **then** e_2 **else** *false* auffassen. Die Beweise laufen wie bei **let** und **rec**.

3 Die Übersetzungsfunktion

Die nächste Frage, die sich stellt ist, wie kommt man von der Umgebungssemantik zur De Bruijn-Semantik?

3.1 Definition der Übersetzungsfunktion

Hierzu wird eine Übersetzungsfunktion definiert. Dazu wird zunächst ein Namenskontext eingeführt, in den man die Namen, die im Ausdruck in der Umgebungssemantik verwendet werden, einträgt.

Definition 3.1 (Namenskontext) Die Menge *Ncx* aller *Namenskontexte* Γ ist induktiv definiert durch:

$$\Gamma \quad ::= \quad [\,] \mid id; \Gamma$$

Der *Definitionsbereich* $dom(\Gamma)$ eines Namenskontext Γ ist wie folgt induktiv definiert:

$$\begin{array}{rcl} dom([\,]) & = & \emptyset \\ dom(id; \Gamma) & = & \{id\} \cup dom(\Gamma) \end{array}$$

Schreibweisen

(a) Für $\Gamma = (id_1; \ldots; id_n; [\,]) \in Ncx$ schreiben wir $[id_1; \ldots; id_n]$.

(b) Für $\Gamma = [id_1; \ldots; id_n]$ und $id \in dom(\Gamma)$ sei

$$\Gamma(id) = \min\{j \in \{1, \ldots, n\} \mid id_j = id\}$$

Jetzt kann man die Übersetzungsfunktion definieren.

Definition 3.2 (Übersetzungsfunktion) Gegeben $e \in Exp$, $\Gamma \in Ncx$ mit $free\,(e) \subseteq dom(\Gamma)$. Der *De Bruijn-Ausdruck* $tr(\Gamma, e)$, welcher durch Entfernen der Bezeichner aus e entsteht, ist wie folgt induktiv definiert:

$$\begin{array}{rcl} tr(\Gamma, c) & = & c \\ tr(\Gamma, id) & = & \underline{i} \quad \text{falls } i = \Gamma(id) \\ tr(\Gamma, \lambda id.\, e) & = & \lambda.\, tr(id; \Gamma, e) \\ tr(\Gamma, e_1\, e_2) & = & (tr(\Gamma, e_1))\,(tr(\Gamma, e_2)) \\ tr(\Gamma, \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2) & = & \textbf{if } tr(\Gamma, e_0) \textbf{ then } tr(\Gamma, e_1) \textbf{ else } tr(\Gamma, e_2) \end{array}$$

Sei weiter $\eta = [id_1 : cl_1; \ldots; id_n : cl_n] \in Env$ mit $free\,(e) \subseteq dom(\eta)$. Die *De Bruijn-Closure* $tr(e, \eta)$, welche durch Entfernen der Bezeichner aus e und η entsteht, ist wie folgt induktiv definiert:

$$\begin{array}{rcl} tr(\eta) & = & [tr(cl_1); \ldots; tr(cl_n)] \\ tr(e, \eta) & = & (tr([id_1; \ldots; id_n], e), tr(\eta)) \end{array}$$

3.2 Eigenschaften der Übersetzungsfunktion

Als erstes muss nun gezeigt werden, dass die Übersetzungsfunktion wohldefiniert ist. Um dies zu zeigen, benötigt man zunächst ein Lemma, welches besagt, dass bei der Übersetzung keine Indizes für nicht vorkommende Variablen erzeugt werden.

Lemma 3.1 *Sei* $\Gamma = [id_1, \ldots, id_n]$ *und* $free\,(e) \subseteq \{id_1, \ldots, id_n\}$. *Dann* $rank\,(tr(\Gamma, e)) \leq n$

Beweis: Induktion über e

- $e = c$ klar.
- $e = id$, dann gilt $rank\,(tr(\Gamma, id)) = \Gamma(id) \leq n$
- $e = \lambda id.e'$, dann gilt $rank\,(tr(\Gamma, \lambda id.e')) = rank\,(\lambda.tr(id; \Gamma, e'))$
  $= max(rank\,(tr(id; \Gamma, e')), 1) - 1 \leq_{I.V.} max(n+1, 1) - 1 \leq max(n, 0) \leq n$
- $e = e_1\, e_2$, dann gilt $rank\,(tr(\Gamma, e_1\, e_2)) = rank\,((tr(\Gamma, e_1))(tr(\Gamma, e_2)))$
  $= \max(rank\,(tr(\Gamma, e_1)), rank\,(tr(\Gamma, e_2)))$. Es folgt mit Induktionsvoraussetzung $l_1 = rank\,(tr(\Gamma, e_1)) \leq n$ und $l_2 = rank\,(tr(\Gamma, e_2)) \leq n$, also insgesamt $\max(l_1, l_2) \leq n$
- $e =$ **if** e_0 **then** e_1 **else** e_2, dann gilt $rank\,(tr(\Gamma,$ **if** e_0 **then** e_1 **else** $e_2))$
  $= rank\,($**if** $tr(\Gamma, e_0)$ **then** $tr(\Gamma, e_1)$ **else** $tr(\Gamma, e_2))$
  $= \max(rank\,(tr(\Gamma, e_0)), rank\,(tr(\Gamma, e_1)), rank\,(tr(\Gamma, e_2)))$ und es folgt mit Induktionsvoraussetzung $l_0 = rank\,(tr(\Gamma, e_0)) \leq n$, $l_1 = rank\,(tr(\Gamma, e_1))$ und $l_2 = rank\,(tr(\Gamma, e_2)) \leq n$, insgesamt also $\max(l_0, l_1, l_2) \leq n$

Nun lässt sich die Wohldefiniertheit der Übersetzung zeigen.

Lemma 3.2 (Wohldefiniertheit der Übersetzung)

(a) *Seien* $e \in Exp$ *und* $\Gamma \in Ncx$ *mit* $free\,(e) \subseteq dom(\Gamma)$,
dann gilt $(tr(\Gamma, e)) \in dbExp$.

(b) *Sei* $cl \in Cl$. *Dann gilt* $(tr(cl)) \in dbCl$.

Beweis:

(a) Induktion über die Größe von e und Fallunterscheidung nach der Form von e.

- $e = c : c \in dbExp$.
- $e = id : free\,(id) \subseteq dom(\Gamma)$ gilt nach Voraussetzung, d.h. $\Gamma(id)$ existiert. Daraus folgt, wenn $i = \Gamma(id)$ dann $\underline{i} \in dbExp$.
- $e = \lambda id.e$: Nach Voraussetzung gilt $free\,(\lambda id.e) \subseteq dom(\Gamma)$. Nach Definition ist $free\,(()\lambda id.e) = free\,(e) \setminus \{id\}$, d.h. $free\,(e) \subseteq free\,(()\lambda id.e) \cup \{id\} \subseteq dom(\Gamma) \cup \{id\} = dom(id; \Gamma)$. Dann folgt nach Induktionsvoraussetzung $tr(id; \Gamma, e) \in dbExp$. Nach Definition gilt $(\lambda.tr(id; \Gamma, e)) \in dbExp$

- $e = e_1\, e_2 : e_1\, e_2, \Gamma \in \mathit{Ncx}$ gilt nach Voraussetzung, sowie $\mathit{free}\,(e_1\, e_2) \subseteq \mathit{dom}(\Gamma)$. Daraus folgt $\mathit{free}\,(e_1) \subseteq \mathit{dom}(\Gamma)$ und $\mathit{free}\,(e_2) \subseteq \mathit{dom}(\Gamma)$. Dann folgt mit Induktionsvoraussetzung $tr(\Gamma, e_1) \in \mathit{dbExp}$ und $tr(\Gamma, e_2) \in \mathit{dbExp}$, also insgesamt $(tr(\Gamma, e_1)\, tr(\Gamma, e_2)) \in \mathit{dbExp}$
- $e = \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2$: analog zu $e = e_1\, e_2$

(b) Dazu sei die Gesamtgröße $|\eta|$ der Umgebung η wie folgt induktiv definiert:

$$\begin{aligned} |[\,]| &= 0 \\ |[id_1 : (e_1, \eta_1); \eta_2]| &= |\eta_1| + |\eta_2| + 1 \end{aligned}$$

Induktion über Gesamtgröße von η und Fallunterscheidung nach der Form von η

- $\eta = [\,]$, also ist $\mathit{free}\,(e) = \emptyset$. Dann folgt $(tr([\,], e)) \in \mathit{dbExp}$ wegen (a) gilt $tr(e, [\,]) = (tr([\,], e), [\,]) \in \mathit{dbCl}$.
- $\eta = [id_1 : (e_1, \eta_1); \ldots; id_n : (e_n, \eta_n)]$. Sei $\Gamma = [id_1, \ldots, id_n]$. Es ist zu zeigen $(tr(\Gamma, e), tr(\eta)) \in \mathit{dbCl}$
  a) $tr(\Gamma, e) \in \mathit{dbExp}$ klar wegen (a).
  b) $tr(\eta) \in \mathit{dbEnv}$
     d.h.: $[tr(cl_1); \ldots tr(cl_n)] \in \mathit{dbEnv}$
     dazu: $tr(cl_i) \in \mathit{dbCl}\ \forall\ i = 1, \ldots, n$ folgt sofort mit Induktionsvoraussetzung
  c) $\mathit{rank}\,(tr(\Gamma, e)) \leq \mathit{len}\,(tr(\eta))$
     dazu: $\mathit{rank}\,(tr(\Gamma, e)) \leq n$, dies folgt mit Lemma 3.1.

Als nächstes ist die Korrektheit der Übersetzungsfunktion zu zeigen.

Wirkliche Korrektheit, also $tr(e, \eta) \Downarrow tr(e', \eta') \Rightarrow (e, \eta) \Downarrow (e', \eta')$ kann es nicht geben, da die Übersetzungsfunktion nur in die andere Richtung eindeutig ist.

Zunächst benötigt man dazu ein Lemma, das die Strukturerhaltung der Übersetzungsfunktion zeigt.

Lemma 3.3 *Die Übersetzungsfunktion tr ist* strukturerhaltend.

Insbesondere gilt $(tr(\Gamma, v)) \in \mathit{dbVal}$ für alle $v \in \mathit{Val}$ und $\Gamma \in \mathit{Ncx}$ mit $\mathit{free}\,(v) \subseteq \mathit{dom}(\Gamma)$.

Beweis: Klar.

Nun kann die eingeschränkte Korrektheit zunächst formuliert und dann bewiesen werden.

Satz 3.1 (Korrektheit) *Sei* $(e, \eta), (\hat{v}, \hat{\eta}) \in \mathit{Cl}$. *Dann gilt wenn* $tr(e, \eta) \Downarrow (\hat{v}, \hat{\eta})$, *existiert* $(v', \eta') \in \mathit{Cl}$, *so dass* $(e, \eta) \Downarrow (v', \eta')$ *und* $(\hat{v}, \hat{\eta}) = tr(v', \eta')$

Beweis: Induktion über Länge des Bigsteps und Fallunterscheidung, nach der zuletzt angewendeten Regel.

- (VAL) dann gilt $tr(e,\eta) = (\hat{v},\hat{\eta})$. Also gilt $e \in \mathit{Val}$. Daher existiert der big step $(e,\eta) \Downarrow (e,\eta)$ mit Regel (VAL) aus der Umgebungssemantik.

- (INDEX) dann gilt $tr(e,\eta) = (\underline{i},\hat{\eta}')$ mit $(\underline{i},\hat{\eta}') \Downarrow (\hat{v},\hat{\eta})$. Das heißt $i \leq \mathit{len}\,(\hat{\eta}')$ und $\hat{\eta}'(\underline{i}) \Downarrow (\hat{v},\hat{\eta})$.
  Sei $\eta = [id_1 : cl_1; \ldots; id_n : cl_n]$. Dann ist $\hat{\eta}' = [tr(cl_1); \ldots; tr(cl_n)]$ und $1 \leq i \leq n$. Das heißt $\hat{\eta}'(i) = tr(cl_i)$.
  Nach Induktionsvoraussetzung existiert ein $(v,\eta) \in \mathit{Cl}$, so dass $cl_i \Downarrow (v,\eta)$ und $tr(v,\eta) = (\hat{v},\hat{\eta})$.
  Das heißt $\eta(id_i) = cl_i$, also existiert $(id_i,\eta) \Downarrow (v,\eta)$ mit Regel (ID) aus der Umgebungssemantik.

- (BETA), dann gilt $tr(e,\eta) = (\hat{e_1}\,\hat{e_2},\hat{\eta}')$ mit $(\hat{e_1}\,\hat{e_2},\hat{\eta}') \Downarrow (\hat{v},\hat{\eta})$ und $e = e_1\,e_2$.
  Nach Voraussetzung existieren die folgenden Big steps:
  - $(\hat{e_1},\hat{\eta}') \Downarrow (\lambda.\,\hat{e_1}',\hat{\eta_1})$
    Nach Induktionsvoraussetzung existiert (v_1',η_1'), so dass $(e_1,\eta) \Downarrow (v_1',\eta_1')$ und $\underbrace{(\lambda.\,\hat{e_1}',\hat{\eta_1}) = tr(v_1',\eta_1')}_{(1)}$
    Sei $\eta_1' = [id_1' : cl_1'; \ldots; id_n' : cl_n']$. Dann existiert $id \in \mathit{Id}$ und $e_1' \in \mathit{Exp}$, so dass $v_1 = \lambda id.\,e_1'$, d. h. $\underbrace{tr([id; id_1'; \ldots; id_n'], e_1') = \hat{e_1}'}_{(2)}$ und $\underbrace{(e_1,\eta) \Downarrow (\lambda id.\,e_1',\eta_1')}_{(*)}$
  - $(\hat{e_1}', (\hat{e_2},\hat{\eta}');\hat{\eta_1}) \Downarrow (\hat{v},\hat{\eta})$
    $tr(id : (e_2,\eta);\eta_1') \overset{(1)}{=} ((\hat{e_2},\hat{\eta}');\hat{\eta_1})$ Mit (2) folgt dann
    $tr(e_1', id : (e_2,\eta);\eta_1') = (\hat{e_1}', (\hat{e_2},\hat{\eta}');\hat{\eta_1})$
    Nach Induktionsvoraussetzung folgt:
    Es existiert $(v',\eta') \in \mathit{Cl}$, so dass
    $\underbrace{(e_1', id : (e_2,\eta);\eta_1')}_{(**)} \Downarrow (v',\eta')$ und $tr(v',\eta') = (\hat{v},\hat{\eta})$

  Insgesamt folgt mit Regel (BETA) aus Umgebungssemantik wegen (*) und (**), dass der big step $(e_1,e_2,\eta) \Downarrow (v',\eta')$ existiert.

- (OP-1), dann gilt $tr(e,\eta) = (\hat{e_1}\,\hat{e_2},\hat{\eta}')$ mit $(\hat{e_1}\,\hat{e_2},\hat{\eta}') \Downarrow (op\,z,[\,])$ und $e = e_1\,e_2$.
  Nach Voraussetzung ex. die folgenden big steps:
  - $(\hat{e_1},\hat{\eta}') \Downarrow (op,\hat{\eta_1})$
    Nach Induktionsvoraussetzung existiert $(v_1,\eta_1) \in \mathit{Cl}, (e_1,\eta) \Downarrow (v_1,\eta_1)$ und $(op,\hat{\eta_1}) = tr(v_1,\eta_1) \Rightarrow (e_1,\eta) \Downarrow (op,\eta_1)$ (*)
  - $(\hat{e_2},\hat{\eta}') \Downarrow (z,\hat{\eta_2})$
    Nach Induktionsvoraussetzung existiert $(v_2,\eta_2) \in \mathit{Cl}, (e_2,\eta) \Downarrow (v_2,\eta_2)$ und $(z,\hat{\eta_2}) = tr(v_2,\eta_2) \Rightarrow (e_2,\eta) \Downarrow (z,\eta_2)$ (**)

Wegen (*) und (**) existiert $(op\, z, [\,]) \in Cl$, so dass $tr(op\, z, [\,]) = (op\, z, [\,])$ und mit (Op-1) aus der Umgebungssemantik folgt $(e, \eta) \Downarrow (op\, z, [\,])$.

- (Cond-True), dann gilt $tr(e, \eta) = (\textbf{if } \hat{e_0} \textbf{ then } \hat{e_1} \textbf{ else } \hat{e_2}, \hat{\eta})$, mit
  $(\textbf{if } \hat{e_0} \textbf{ then } \hat{e_1} \textbf{ else } \hat{e_2}, \hat{\eta'}) \Downarrow (\hat{v}, \hat{\eta})$ und $e = \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2$
  Nach Voraussetzung existieren folgende big steps:
  - $(\hat{e_0}, \eta') \Downarrow (true, \hat{\eta_0})$
    Nach Induktionsvoraussetzung existiert $\eta_0 \in Env$, so dass $(e_0, \eta) \Downarrow (true, \eta_0)$ und $(true, \hat{\eta_0}) = tr(true, \eta_0)$
  - $(\hat{e_1}, \eta') \Downarrow (\hat{v}, \hat{\eta})$
    Nach Induktionsvoraussetzung existiert $(v', \eta') \in Cl$, so dass $(e_1, \eta) \Downarrow (v', \eta')$ und $(\hat{v}, \hat{\eta}) = tr(v', \eta')$

  Insgesamt existiert mit (Cond-True) aus der Umgebungssemantik der big step $(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2, \eta) \Downarrow (v', \eta')$.

Die restlichen Fälle laufen analog.

Nachdem die Korrektheit gezeigt wurde, fehlt jetzt nur noch die Vollständigkeit. Diese ist ziemlich einfach zu formulieren, wie der folgende Satz zeigt.

Satz 3.2 (Vollständigkeit) *Sei $(e, \eta), (e', \eta') \in Cl$. Dann gilt wenn $(e, \eta) \Downarrow (v, \eta')$ existiert $tr(e, \eta) \Downarrow tr(v, \eta')$.*

Beweis: Induktion über Länge des Bigsteps und Fallunterscheidung, nach der zuletzt angewendeten Regel.

- (Val) dann gilt $e = v \in db\,Val$. Daher existiert der big step $tr(v, \eta) \Downarrow tr(v, \eta)$ mit Regel (Val) aus der De Bruijn Semantik.

- (Id) dann gilt $(e, \eta) = (id, \eta)$ also $\eta = [id_1, cl_1; \ldots; id_n, cl_n]$ und es existiert ein i mit $1 \leq i \leq n$ und $id = id_i$. Außerdem ist $tr(id, \eta) = tr(\underline{i}, tr(\eta))$ und $tr(\eta)(i) = tr(cl_i)$ mit $\eta(id) = cl_i$. Aus der Pämisse ergibt sich, dass $cl_i \Downarrow (v, \eta')$ existiert. Mit Induktionsvoraussetzung folgt $tr(cl_i) \Downarrow tr(v, \eta')$, d.h. es existiert der big step $tr(id, \eta) = (\underline{i}, tr(\eta)) \Downarrow tr(v, \eta')$ mit Regel (Index) aus der De Bruijn Semantik.

- (Beta) dann gilt $(e_1\, e_2, \eta) \Downarrow (v, \eta')$ aus den Prämissen ergibt sich, dass folgende big steps existieren:
  - $(e_1, \eta) \Downarrow (\lambda \hat{id}.\, \hat{e}, \hat{\eta})$, wobei $\hat{\eta} = [\hat{id_1} : \hat{cl_1}; \ldots; \hat{id_n} \hat{cl_n}]$
    Mit Induktionsvoraussetzung folgt $tr(e_1, \eta) \Downarrow tr(\lambda \hat{id}.\, \hat{e}, \hat{\eta}) =$
    $\underbrace{(\lambda.tr(\hat{\Gamma}, \hat{e}), tr(\hat{\eta}))}_{(1)}$, wobei $\hat{\Gamma} = [\hat{id}, \hat{id_1}, \ldots, \hat{id_n}]$

  - $(\hat{e}, \hat{id} : (e_2, \eta); \hat{\eta}) \Downarrow (v, \eta')$
    Das heißt $tr(\hat{e}, \hat{id} : (e_2, \eta); \hat{\eta}) \Downarrow tr(v, \eta') = \underbrace{(tr(\hat{\Gamma}, \hat{e}), tr(e_2, \eta); tr(\hat{\eta}))}_{(2)}$

  Wegen (1) und (2) folgt mit (Beta) aus der De Bruijn Semantik, dass der big step $tr(e_1\, e_2) \Downarrow tr(v, \eta')$ existiert.

- (Cond-True) dann gilt (**if** e_0 **then** e_1 **else** $e_2, \eta) \Downarrow (v, \eta')$ mit $\eta = [id_1 : cl_1; \ldots; id_n : cl_n]$. Sei $\Gamma = [id_1, \ldots, id_n]$, dann gilt $tr($**if** e_0 **then** e_1 **else** $e_2, \eta) =$ (**if** $tr(\Gamma, e_0)$ **then** $tr(\Gamma, e_1)$ **else** $tr(\Gamma, e_2), tr(\eta))$ Außerdem existieren nach Voraussetzung folgende big steps:
  - $(e_0, \eta) \Downarrow (true, \eta_0)$
    Mit Induktionsvoraussetzung folgt $tr(e_0, \eta) \Downarrow tr(true, \eta_0)$
  - $(e_1, \eta) \Downarrow (v, \eta')$
    Mit Induktionsvoraussetzung folgt $tr(e_1, \eta) \Downarrow tr(v, \eta')$

  Da die Prämissen für die Regel (Cond-True), aus der De Bruijn Semantik, erfüllt sind, existiert der big step $tr($**if** e_0 **then** e_1 **else** $e_2, \eta) \Downarrow tr(v, \eta')$ mit (Cond-True).

Die restlichen Fälle laufen analog.

Insgesamt ergibt sich durch die Sätze 3.1 und 3.2 eine Äquivalenz. Diese könnte auch zu einem Satz zusammengefasst werden, was aber, aufgrund der etwas komplizierteren Formulierung der Korrektheit, ziemlich schwierig und auch nicht viel mehr ergeben würde.

4 Vergleich mit Pierce

In diesem Kapitel soll die vorher eingeführte De Bruijn Semantik mit der in [Pie02, S.75 ff.] verglichen werden. Dazu wird zunächst die Semantik, die Pierce vorschlägt, vorgestellt und danach verglichen.

Die Beispiele in diesem Kapitel sind zum größten Teil aus [Pie02] entnommen.

4.1 Ungetypter λ-Kalkül

Als Sprache wird der reine, ungetypte λ-Kalkül verwendet. Dieser soll hier zur Erinnerung kurz gezeigt werden.

Definition 4.1 (Syntax der Sprache) Die Mengen *Exp* aller *Ausdrücke* e und *Val* aller *Werte* v sind durch folgende kontextfreie Grammatik definiert:

$$\begin{aligned} e &::= id \mid \lambda id.\, e \mid e_1\, e_2 \\ v &::= \lambda id.\, e \end{aligned}$$

Wobei $id \in Id$ ist.

Auf dieser Basis werden jetzt die small step Regeln definiert. Im Unterschied zu Pierce betrachten wir hier wieder eine Call-By-Name Semantik.

Definition 4.2 (Small step Regeln) Ein *small step* im ungetypten λ-Kalkül ist eine Formel der Gestalt $e \to v$ mit $e \in Exp$

$$\text{(App-Left)} \quad \frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}$$

$$\text{(Beta)} \quad (\lambda id.\, e_1)\, e_2 \to e_1[e_2/id]$$

4.2 Namenskontexte

Um nun die namenlose Darstellung zu erhalten, benötigt man zunächst einen sogenannten „naming context" Γ, der wie bei der de Bruijn-Semantik definiert ist (siehe Definition 3.1).

Für das Eintragen und Auslesen von Γ werden die Funktionen *removenames* und *restorenames* definiert.

Definition 4.3 Gegeben sei $e \in Exp$ und $\Gamma \in Ncx$, dann ist die Funktion $removenames_\Gamma(e)$ wie folgt definiert.

$$\begin{aligned} removenames_\Gamma(id) &= \Gamma(id), \text{ wobei das rechteste } id \in \Gamma \text{ gewählt wird.} \\ removenames_\Gamma(\lambda id.\, e) &= \lambda.\, removenames_{\Gamma,id}(e) \\ removenames_\Gamma(e_1\, e_2) &= removenames_\Gamma(e_1)\, removenames_\Gamma(e_2) \end{aligned}$$

Definition 4.4 Gegeben sei $e \in Exp$ und $\Gamma \in Ncx$, dann ist die Funktion $restorenames_\Gamma(e)$ wie folgt definiert.

$$\begin{aligned} restorenames_\Gamma(\underline{n}) &= id_n \in \Gamma \\ restorenames_\Gamma(\lambda.\, e) &= \lambda id.\, restorenames_{\Gamma,id}(e), \text{ wobei } id \notin dom(\Gamma) \\ restorenames_\Gamma(e_1\, e_2) &= restorenames_\Gamma(e_1)\, restorenames_\Gamma(e_2) \end{aligned}$$

4.3 Shifting und Substitution

Als nächstes muss die Substitution auf namenlosen Ausdrücken definiert werden. Dazu wird die Operation „Shifting“ eingeführt, welche die Indizes der freien Variablen neu nummeriert.

Dies ist notwendig, wenn eine Substitution auf eine Abstraktion angewendet wird. Ein Beispiel wäre $\lambda.\, \underline{2}[s/\underline{1}]$, also mit Namen zum Beispiel $\lambda w.\, x[s/x]$. Dabei müssen die Indizes der freien Variablen in s angepasst werden. Aber es dürfen nicht einfach alle Indizes um 1 erhöht werden, sondern die Erhöhung muss „vorsichtig“ gemacht werden. Dies wäre zum Beispiel wenn $s = \underline{2}\,(\lambda.\, \underline{0})$ (also mit Namen zum Beispiel $s = a\,(\lambda b.\, b)$), dabei muss die $\underline{2}$ erhöht werden, die $\underline{0}$ muss allerdings erhalten bleiben.

Um dies sicherzustellen wird das Shifting wie folgt definiert.

Definition 4.5 (Shifting) Gegeben sei $e \in Exp$, dann ist der „d-malige shift“ für alle Indizes oberhalb von c, $\uparrow_c^d (e)$, definiert wie folgt.

$$\begin{aligned} \uparrow_c^d (\underline{k}) &= \begin{cases} \underline{k} & wenn\ k < c \\ \underline{k+d} & sonst \end{cases} \\ \uparrow_c^d (\lambda.\, e) &= \lambda.\ \uparrow_{c+1}^d (e) \\ \uparrow_c^d (e_1\, e_2) &= \uparrow_c^d (e_1)\ \uparrow_c^d (e_2) \end{aligned}$$

$\uparrow^d (e)$ steht im weiteren Verlauf für $\uparrow_0^d (e)$.

Beispiel 4.1 $\uparrow^2 (\lambda.\, \lambda.\, \underline{1}\,(\underline{0}\,\underline{2})) = \lambda.\, \lambda.\, \underline{1}\,(\underline{0}\,\underline{4})$

Beispiel 4.2 $\uparrow^2 (\lambda.\, \underline{0}\,\underline{1}\,(\lambda.\, \underline{0}\,\underline{1}\,\underline{2})) = \lambda.\, \underline{0}\,\underline{3}\,(\lambda.\, \underline{0}\,\underline{1}\,\underline{4})$

Man sieht hier, dass zunächst die erste $\underline{0}$ „geschützt“ ist durch das λ, die $\underline{1}$ hingegen nicht. In der zweiten Abstraktion ist dann die $\underline{0}$ durch die innere Abstraktion und die $\underline{1}$ durch die Äußere geschützt, die $\underline{2}$ ist dagegen nicht geschützt und wird erhöht.

Daraus bildet sich dann die neue Definition der Substitution:

Definition 4.6 (Substitution) Die Substitution einer Zahl i durch einen Ausdruck s in dem Ausdruck e, also $e[s/i]$, ist definiert durch:

- $\underline{k}[s/\underline{i}] = \begin{cases} s & wenn\ k = i \\ \underline{k} & sonst \end{cases}$
- $(\lambda.\, e)[s/\underline{i}] = \lambda.\, e[\uparrow^1 (s)/\underline{i+1}]$
- $(e_1\, e_2)[s/\underline{i}] = e_1[s/\underline{i}]\, e_2[s/\underline{i}]$

Beispiel 4.3 Sei $\Gamma = a, b$ der vorgegebene Namenskontext und die folgende Substitution anzuwenden $(b\,(\lambda x.\,(\lambda y.\, b)))[a/b]$.

- Zunächst wird removenames angewendet und es ergibt sich Folgendes:
  $(\underline{0}\,(\lambda.\,(\lambda.\,\underline{2})))[\underline{1}/\underline{0}]$
- Nun kann die Substitution angewendet werden:

  $(\underline{0}\,(\lambda.\,(\lambda.\,\underline{2})))[\underline{1}/\underline{0}]$
  $= \underline{1}\,(\lambda.\,(\lambda.\,\underline{2})[\underline{2}/\underline{1}])$
  $= \underline{1}\,(\lambda.\,(\lambda.\,\underline{2}[\underline{3}/\underline{2}]))$
  $= \underline{1}\,(\lambda.\,(\lambda.\,\underline{3}))$
- Zurückübersetzt lautet der Ausdruck $a\,(\lambda x.\,\lambda y.\, a)$

4.4 Regeländerungen

Es muss nur die Regel (Beta) verändert werden, denn sie ist die einzige in der Variablen vorkommen. Diese muss jetzt unsere neue namenlose Substitution nutzen.

Zunächst müssen die Indizes in e_2 alle um eins hochgezählt werden, bevor dieser in e_1 eingesetzt wird, da sonst die ungebundenen Indizes in e_2 nicht mehr richtig wären.

Dann muss der Index, der durch die λ-Abstraktion gebunden wird, „rausgenommen“ werden, daher werden die Indizes alle um 1 reduziert.

Beispiel 4.4 $(\lambda.\,\underline{1}\,\underline{0}\,\underline{2})\,(\lambda.\,\underline{0}) \rightarrow \underline{0}\,(\lambda.\,\underline{0})\,\underline{1}$ nicht $\underline{1}\,(\lambda.\,\underline{0})\,\underline{2}$

Man könnte also sagen, zuerst müssen die Indizes in e_2 geschützt werden und danach muss die Variable, die durch die Abstraktion hinzukommt, wieder herausgenommen werden.

Insgesamt ergibt sich die folgende Regel:
(Beta') $(\lambda.\, e_1)\, e_2 \rightarrow \lambda.\ \uparrow^{-1} (e_1[\uparrow^1 (e_2)/\underline{0}])$

4.5 Vergleich

In der Substitutionssemantik wird immer nur die aktuelle Substitution betrachtet. Daher muss man ggf. die Indizes neu nummerieren, was dazu führt, dass Substitution schwieriger wird, da man immer darauf achten muss, dass die Indizes richtig sind.(→ Shifting)

In der De Bruijn Semantik dagegen werden Closures betrachtet. Es wird also zu jedem Ausdruck die passende Umgebung betrachtet. Daher ist mit Umgebungen kein „shiften“ nötig.

5 Fazit

Wir haben gesehen, dass man Namen komplett aus der Umgebungssemantik entfernen kann. Damit haben wir die Implementierung noch schneller gemacht, da wir nun weder einen Stringvergleich bei den Namen machen, noch die gesamte Umgebung nach der passenden Closure durchsuchen müssen. Die Suche verläuft einfach über den Index, d.h. eine Umgebung kann man sich als eine Art Array vorstellen, so dass der Zugriff sehr schnell geht.

Nebenbei haben wir auch noch die Uneindeutigkeit der Namen aus der Sprache entfernt, da es ja jetzt keine Namen mehr gibt.

Zum wirklichen Programmieren eignet sich diese Sprach nur bedingt, da man relativ schnell mit den Indizes durcheinander kommen kann. In diesem Fall sind Namen dann doch etwas einfacher. Aber da wir eine Übersetzungsfunktion haben, müssen wir auch nicht in der Sprache unsere Programme schreiben, sondern können sie „bequem“ in der Umgebungssemantik schreiben und dann übersetzen.

Als Alternative wurde eine Substitutionssemantik mit De Bruijn-Indizes vorgestellt. Hier konnte man leicht sehen, dass die Nummerierung in den Substitutionen zu Problemen führt und man daher ein relativ aufwändiges shiften der Indizes vornehmen muss. Dieses Problem haben wir mit Umgebungen nicht.

Literatur

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

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

[Sie08] Sieber, Kurt: *Theorie der Programmierung I*. 2008. – Vorlesungsmitschrift