UNIVERSITÄT SIEGEN
FACHBEREICH MATHEMATIK

Diplomarbeit

Theoretische Grundlagen der Objektorientierung

Benedikt Meurer
6. März 2007

INTERNE BERICHTE
INTERNAL REPORTS

Diplomarbeit am
Fachbereich Mathematik
der Universität Siegen

Gutachter:
Privatdozent Dr. Kurt Sieber
Prof. Dr. Dieter Spreen

Inhaltsverzeichnis

Einleitung **4**

1 Grundlagen **5**

1.1 Syntax der Sprache $\mathcal{L}_f$ 5

1.2 Operationelle Semantik der Sprache $\mathcal{L}_f$ 6

1.2.1 Frei vorkommende Namen und Substitution 7

1.2.2 Small step Semantik 8

2 Funktionale Objekte **10**

2.1 Syntax der Sprache $\mathcal{L}_o$ 10

2.2 Operationelle Semantik der Sprache $\mathcal{L}_o$ 11

2.2.1 Frei vorkommende Namen und Substitution 12

2.2.2 Small step Semantik 15

2.2.3 Big step Semantik 18

2.2.4 Äquivalenz der Semantiken 19

2.3 Ein einfaches Typsystem 20

2.3.1 Syntax der Sprache $\mathcal{L}_o^t$ 20

2.3.2 Typsystem der Sprache $\mathcal{L}_o^t$ 22

Index **30**

Literaturverzeichnis **30**

Erklärung **31**

Einleitung

Narf, [Rém02, 50f].

Kapitel 1

Grundlagen

Einführend soll kurz die funktionale Programmiersprache $\mathcal{L}_f$ vorgestellt werden, die als Grundlage für die weiteren Betrachtungen in diesem Dokument dient. Hierbei handelt es sich um einen ungetypten λ-Kalkül mit Konstanten, der im wesentlichen mit dem in der Vorlesung „Theorie der Programmierung I“ ([Sie04], [Sie06]) vorgestellten übereinstimmt.

1.1 Syntax der Sprache $\mathcal{L}_f$

TODO: Prosa.

Definition 1.1 Vorgegeben sei

(a) eine unendliche Menge *Id* von *Namen id*,

(b) eine Menge *Int* (von Darstellungen) ganzer Zahlen n und

(c) die Menge $Bool = \{false, true\}$ der boolschen Werte b.

Während reale Programmiersprachen die Menge *Id* bestimmten Beschränkungen unterwerfen, die zumeist auf Einschränkungen der konkreten Syntax zurückzuführen sind, wie zum Beispiel, dass *Id* keine Schlüsselwörter der Sprache enthalten darf, fordern wir lediglich, dass *Id* mindestens abzählbar unendlich ist.

Auch müsste in einer realen Programmiersprache spezifiziert werden, was genau unter der Darstellung einer ganzen Zahl $n \in Int$ verstanden werden soll. Für die Betrachtungen in diesem Dokument sind diese Aspekte allerdings irrelevant, und wir nehmen deshalb an, dass die beiden Mengen *Int* und $\mathbb{Z}$ rekursiv isomorph sind und unterscheiden nicht weiter zwischen einer ganzen Zahl und ihrer Darstellung.

Definition 1.2 (Abstrakte Syntax von $\mathcal{L}_f$) Die Menge *Op* aller Operatoren *op* ist definiert durch die kontextfreie Grammatik

$$
\begin{array}{lll}
op & ::= & + \mid - \mid * \\
 & \mid & < \mid > \mid \leq \mid \geq \mid =
\end{array}
\qquad
\begin{array}{l}
\text{arithmetische Operatoren} \\
\text{Vergleichsoperatoren,}
\end{array}
$$

die Menge *Const* aller Konstanten c durch

$$
\begin{array}{rcll}
c & ::= & () & \textbf{unit}\text{-Element} \\
 & | & b & \text{boolscher Wert} \\
 & | & n & \text{Ganzzahl} \\
 & | & op & \text{Operator}
\end{array}
$$

und die Menge *Exp* aller *Ausdrücke* e von $\mathcal{L}_f$ durch

$$
\begin{array}{rcll}
e & ::= & c & \text{Konstante} \\
 & | & id & \text{Name} \\
 & | & e_1\, e_2 & \text{Applikation} \\
 & | & \lambda id.e_1 & \lambda\text{-Abstraktion} \\
 & | & \mathbf{rec}\, id.e_1 & \text{rekursiver Ausdruck} \\
 & | & \mathbf{let}\, id = e_1\, \mathbf{in}\, e_2 & \textbf{let}\text{-Ausdruck} \\
 & | & \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 & \text{bedingter Ausdruck.}
\end{array}
$$

Dem aufmerksamen Leser wird sicherlich nicht entgangen sein, dass keine Operatoren für Ganzzahldivision und Divisionsrest existieren. Diese wurden bewusst weggelassen, da eine Hinzunahme dieser Operatoren zur Kernsprache eine Laufzeitfehlerbehandlung notwendig macht, welche unsere Betrachtungen der Sprache unnötig kompliziert machen würde. Wir werden Laufzeitfehlerbehandlung später als Erweiterung der Kernsprache betrachten.

Nichtsdestrotrotz ist die Programmiersprache $\mathcal{L}_f$ hinreichend mächtig. Der nächste Abschnitt führt die Funktionen *div* und *mod* als syntaktischen Zucker in die Sprache ein.

Syntaktischer Zucker

$$
\begin{array}{rcll}
e_1\, op\, e_2 & \text{für} & op\, e_1\, e_2 & \text{Infixnotation} \\
-e & \text{für} & 0 - e & \text{unäres Minus}
\end{array}
$$

TODO: Die bereits angesprochenen *div* und *mod*, Division durch 0 ist nicht definiert.

$$
\begin{array}{rcl}
div & = & \mathbf{rec}\, div.\lambda x.\lambda y. \\
 & & \quad \mathbf{if}\, x < y\, \mathbf{then}\, 0 \\
 & & \quad \mathbf{else}\, (div\ (x - y)\ y) + 1 \\
 & & \\
mod & = & \lambda x.\lambda y.x - y * (div\, x\, y)
\end{array}
$$

Der Leser mag sich selbst davon überzeugen, dass diese Implementationen gemäss der im nächsten Abschnitt vorgestellten Semantik korrekt sind.

1.2 Operationelle Semantik der Sprache $\mathcal{L}_f$

Definition 1.3 Für jeden Operator $op \in Op$ sei eine Funktion op^I vorgegeben mit

$$
\begin{array}{ll}
op^I : Int \times Int \to Int & \text{falls } op \in \{+, -, *\} \\
op^I : Int \times Int \to Bool & \text{sonst.}
\end{array}
$$

Die Funktion op^I heisst *Interpretation* des Operators op.

Auf eine exakte Definition der Interpretationen der verschiedenen Operatoren wird hier verzichtet. Wir nehmen stattdessen an, dass diese Funktionen gemäß ihrer intuitiven Bedeutung definiert seien.

Definition 1.4 Die Menge $Val_e \subseteq Exp$ aller Werte v ist durch die kontextfreie Grammatik

$$\begin{array}{lcll} v & ::= & c & \text{Konstante} \\ & | & id & \text{Name} \\ & | & op\, v_1 & \text{partielle Applikation} \\ & | & \lambda id.e_1 & \lambda\text{-Abstraktion} \end{array}$$

definiert.

1.2.1 Frei vorkommende Namen und Substitution

TODO: Blah blub

Definition 1.5 (Frei vorkommende Namen) Die Menge $\mathit{free}\,(e)$ aller im Ausdruck $e \in Exp$ *frei vorkommenden Namen* wird durch Induktion über die Grösse von e wie folgt definiert.

$$\begin{array}{rcl} \mathit{free}\,(c) & = & \emptyset \\ \mathit{free}\,(id) & = & \{id\} \\ \mathit{free}\,(e_1\, e_2) & = & \mathit{free}\,(e_1) \cup \mathit{free}\,(e_2) \\ \mathit{free}\,(\lambda id.e_1) & = & \mathit{free}\,(e_1) \setminus \{id\} \\ \mathit{free}\,(\mathbf{rec}\, id.e_1) & = & \mathit{free}\,(e_1) \setminus \{id\} \\ \mathit{free}\,(\mathbf{let}\, id = e_1\, \mathbf{in}\, e_2) & = & \mathit{free}\,(e_1) \cup (\mathit{free}\,(e_2) \setminus \{id\}) \\ \mathit{free}\,(\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) & = & \mathit{free}\,(e_0) \cup \mathit{free}\,(e_1) \cup \mathit{free}\,(e_2) \end{array}$$

TODO: Blub blah

Definition 1.6 Ein Ausdruck $e \in Exp$ heisst *abgeschlossen*, wenn $\mathit{free}\,(e) = \emptyset$.

TODO: Beispiel

Es ist leicht zu sehen, dass gemäß dieser Definition, ein Ausdruck genau dann abgeschlossen ist, wenn er keine freien Vorkommen von Namen enthält. Zum Beispiel sind im Ausdruck

$$f\,(x+1)$$

die beiden Vorkommen der Namen f und x frei, der Ausdruck also nicht abgeschlossen. Erweitert man hingegen den Ausdruck zu

$$\mathbf{let}\, f = \lambda y.y * y\, \mathbf{in}\, \mathbf{let}\, x = 2\, \mathbf{in}\, f\,(x+1)$$

so werden nun beide zuvor freien Vorkommen von Namen durch **let**-Ausdrücke gebunden und der Gesamtausdruck ist abgeschlossen.

Definition 1.7 (Substitution) Für $e, e' \in Exp$, $id \in Id$ ist der Ausdruck $e'[e/id]$, der aus e' durch *Substitution* von e für id entsteht, durch Induktion über die Grösse von e wie folgt definiert.

$$
\begin{array}{rcl}
c[e/id] & = & c \\
id'[e/id] & = & \begin{cases} e & \text{falls } id = id' \\ id' & \text{sonst} \end{cases} \\
(e_1\, e_2)[e/id] & = & e_1[e/id]\, e_2[e/id] \\
(\lambda id'.e')[e/id] & = & \begin{cases} \lambda id'.e' & \text{falls } id = id' \\ \lambda \widetilde{id}.e'[\widetilde{id}/id'][e/id] & \text{sonst} \end{cases} \\
& & \text{mit } \widetilde{id} \notin \mathit{free}\,(\lambda id'.e') \cup \{id\} \cup \mathit{free}\,(e) \\
(\mathbf{rec}\, id'.e')[e/id] & = & \begin{cases} \mathbf{rec}\, id'.e' & \text{falls } id = id' \\ \mathbf{rec}\, \widetilde{id}.e'[\widetilde{id}/id'][e/id] & \text{sonst} \end{cases} \\
& & \text{mit } \widetilde{id} \notin \mathit{free}\,(\mathbf{rec}\, id'.e') \cup \{id\} \cup \mathit{free}\,(e) \\
(\mathbf{let}\, id' = e_1\, \mathbf{in}\, e_2)[e/id] & = & \begin{cases} \mathbf{let}\, id' = e_1\, \mathbf{in}\, e_2 & \text{falls } id = id' \\ \mathbf{let}\, \widetilde{id} = e_1[e/id]\, \mathbf{in}\, e_2[\widetilde{id}/id'][e/id] & \text{sonst} \end{cases} \\
& & \text{mit } \widetilde{id} \notin \mathit{free}\,(\mathbf{let}\, id' = e_1\, \mathbf{in}\, e_2) \cup \{id\} \cup \mathit{free}\,(e) \\
(\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2)[e/id] & = & \mathbf{if}\, e_0[e/id]\, \mathbf{then}\, e_1[e/id]\, \mathbf{else}\, e_2[e/id]
\end{array}
$$

TODO: α-Konversion (Quelle: Church) bzw. gebundene Umbenennung. Vielleicht noch ein Beispiel. Wahl des $\widetilde{id}$ in TPML als Beispiel für algorithmisches Verfahren.

1.2.2 Small step Semantik

TODO: Prosa

Definition 1.8 Ein *small step* ist eine Formel der Gestalt $e \rightarrow_e e'$ mit $e, e' \in Exp$.

TODO: Lala

Definition 1.9 (Gültige small steps für $\mathcal{L}_f$) Ein small $e \rightarrow_e e'$ mit $e, e' \in Exp$ heisst gültig für $\mathcal{L}_f$, wenn er sich mit den folgenden Regeln herleiten lässt.

(OP)	$op\, n_1\, n_2 \rightarrow_e op^I(n_1, n_2)$
(BETA-V)	$(\lambda id.e)\, v \rightarrow_e e[v/id]$
(APP-LEFT)	$\dfrac{e_1 \rightarrow_e e_1'}{e_1\, e_2 \rightarrow_e e_1'\, e_2}$
(APP-RIGHT)	$\dfrac{e_2 \rightarrow_e e_2'}{v_1\, e_2 \rightarrow_e v_1\, e_2'}$
(UNFOLD)	$\mathbf{rec}\, id.e \rightarrow_e e[\mathbf{rec}\, id.e/id]$
(LET-EVAL)	$\dfrac{e_1 \rightarrow_e e_1'}{\mathbf{let}\, id = e_1\, \mathbf{in}\, e_2 \rightarrow_e \mathbf{let}\, id = e_1'\, \mathbf{in}\, e_2}$
(LET-EXEC)	$\mathbf{let}\, id = v_1\, \mathbf{in}\, e_2 \rightarrow_e e_2[v_1/id]$
(COND-EVAL)	$\dfrac{e_0 \rightarrow_e e_0'}{\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \rightarrow_e \mathbf{if}\, e_0'\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2}$
(COND-TRUE)	$\mathbf{if}\, true\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \rightarrow_e e_1$
(COND-FALSE)	$\mathbf{if}\, false\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \rightarrow_e e_2$

TODO: Erläuterung der Regeln

Kapitel 2

Funktionale Objekte

Basierend auf der einfachen funktionalen Programmiersprache $\mathcal{L}_f$, die im vorangegangenen Kapitel vorgestellt wurde, soll nun die objektorientierte Programmiersprache $\mathcal{L}_o$ entwickelt werden, indem $\mathcal{L}_f$ durch objektorientierte Konzepte erweitert wird. In diesem Kapitel werden dazu zunächst einfache, funktionale Objekte in die Sprache eingeführt.

2.1 Syntax der Sprache $\mathcal{L}_o$

TODO: Prosa.

Definition 2.1 Vorgegeben sei eine unendliche Menge *Method* von *Methodennamen* m.

Ähnlich wie bei *Id* gelten für *Method* keinerlei Einschränkungen, außer dass die Menge mindestens abzählbar unendlich sein muss. Eine Implementierung wird üblicherweise $\mathit{Method} = \mathit{Id}$ wählen.

TODO: Noch ein bisschen einführende Erläuterung zur Idee von Reihen.

Definition 2.2 (Abstrakte Syntax von $\mathcal{L}_o$) Die Menge *Row* aller *Reihen* r von $\mathcal{L}_o$ ist durch definiert durch die kontextfreie Grammatik

$$
\begin{array}{rcll}
r & ::= & \epsilon & \text{leere Reihe} \\
  & | & \mathbf{val}\ id = e;\ r_1 & \text{Attribut} \\
  & | & \mathbf{method}\ m = e;\ r_1 & \text{Methode}
\end{array}
$$

und die Menge *Exp* aller *Ausdrücke* e von $\mathcal{L}_o$ erhält man, indem man die Produktionen

$$
\begin{array}{rcll}
e & ::= & r\#m & \text{Hilfsausdruck} \\
  & | & e_1\#m & \text{Methodenaufruf} \\
  & | & \mathbf{object}\ (id)\ r\ \mathbf{end} & \text{Objekt} \\
  & | & \{\langle id_1 = e_1; \ldots; id_n = e_n \rangle\} & \text{Duplikation}
\end{array}
$$

zur Grammatik von $\mathcal{L}_f$ hinzunimmt, wobei die Attributnamen $id_1, \ldots, id_n$ bei der Duplikation paarweise verschieden sein müssen.

Bei $r\#m$ handelt es sich um einen Hilfsausdruck, der benötigt wird, um die operationelle Semantik der Sprache zu definieren. Dieser Ausdruck hat keine Entsprechung in der konkreten Syntax der Programmiersprache, kann also nicht direkt durch den Programmierer verwendet werden.

Die Forderung, dass die Attributnamen bei der Duplikationen verschieden sein müssen, kann in der konkreten Syntax aufgeweicht werden. Zum Beispiel könnten bei der Syntaxanalyse mehrfache Vorkommen des gleichen Namens erkannt werden und alle bis auf das letzte Vorkommen verworfen werden.

Definition 2.3 Sei $r \in \mathit{Row}$ eine Reihe.

(a) Der *Definitionsbereich* $\mathit{dom}\,(r)$ der Reihe r wird wie folgt induktiv über die Grösse von r definiert.

$$\begin{array}{rcl} \mathit{dom}\,(\epsilon) & = & \emptyset \\ \mathit{dom}\,(\mathbf{val}\ id = e;\, r_1) & = & \mathit{dom}\,(r_1) \cup \{(\mathbf{attr}, id)\} \\ \mathit{dom}\,(\mathbf{method}\ m = e;\, r_1) & = & \mathit{dom}\,(r_1) \cup \{(\mathbf{method}, m)\} \end{array}$$

(b) Die Menge der Attributnamen $\mathit{dom}_a\,(r)$ der Reihe r ist wie folgt definiert.

$$\mathit{dom}_a\,(r) = \big\{ id \,|\, (\mathbf{attr}, id) \in \mathit{dom}\,(r) \big\}$$

(c) Die Menge der Methodennamen $\mathit{dom}_m\,(r)$ der Reihe r ist wie folgt definiert.

$$\mathit{dom}_m\,(r) = \big\{ m \,|\, (\mathbf{method}, m) \in \mathit{dom}\,(r) \big\}$$

Syntaktischer Zucker

$$\mathbf{method}\ m\ id_1 \ldots id_n = e;\, r \quad \text{für} \quad \mathbf{method}\ m = \lambda id_1 . \ldots \lambda id_n . e;\, r$$

2.2 Operationelle Semantik der Sprache $\mathcal{L}_o$

TODO: Prosa.

Definition 2.4

(a) Sei $\mathit{Val}_r \subseteq \mathit{Row}$ die Menge aller *Reihenwerte* ω von $\mathcal{L}_o$ mit den folgenden Eigenschaften.

- Jede Attributdeklaration in ω ist von der Form $\mathbf{val}\ id = v;\, \omega'$ mit $v \in \mathit{Val}_e, \omega' \in \mathit{Val}_r$ und
- die Namen aller Attribute in ω sind paarweise verschieden.

(b) Die Menge $\mathit{Val}_e \subseteq \mathit{Exp}$ aller *Werte* v von $\mathcal{L}_o$ erhält man, indem man die Produktion

$$v \ ::= \ \mathbf{object}\,(id)\ \omega\ \mathbf{end}$$

zur kontextfreien Grammatik von $\mathcal{L}_f$ hinzunimmt.

TODO: Prosa, z.B. warum Attributnamen paarweise verschieden (in anderen Sprachen gilt das generell für die kf. Grammatik, bei uns nur für Werte, $\rightarrow$ Vererbung).

2.2.1 Frei vorkommende Namen und Substitution

TODO: Einleitende Worte mit Bezug auf $\mathcal{L}_f$

Definition 2.5 (Frei vorkommende Namen)

(a) Die Menge $\mathit{free}\,(e)$ aller im Ausdruck $e \in \mathit{Exp}$ *frei vorkommenden Namen* erhält man durch folgende Erweiterung von Definition 1.5.

$$\begin{aligned}
\mathit{free}\,(r\#m) &= \mathit{free}\,(r) \\
\mathit{free}\,(e_1\#m) &= \mathit{free}\,(e_1) \\
\mathit{free}\,(\mathbf{object}\,(\mathit{id})\ r\ \mathbf{end}) &= \mathit{free}\,(r) \setminus \{\mathit{id}\} \\
\mathit{free}\,(\{\langle \mathit{id}_1 = e_1; \ldots; \mathit{id}_n = e_n\rangle\}) &= \mathit{free}\,(e_1) \cup \ldots \cup \mathit{free}\,(e_n)
\end{aligned}$$

(b) Die Menge $\mathit{free}\,(r)$ aller in der Reihe $r \in \mathit{Row}$ *frei vorkommenden Namen* wird wie folgt induktiv definiert.

$$\begin{aligned}
\mathit{free}\,(\epsilon) &= \emptyset \\
\mathit{free}\,(\mathbf{val}\ \mathit{id} = e;\ r_1) &= \mathit{free}\,(e) \cup \big(\mathit{free}\,(r_1) \setminus \{\mathit{id}\}\big) \\
\mathit{free}\,(\mathbf{method}\ m = e;\ r_1) &= \mathit{free}\,(e) \cup \mathit{free}\,(r_1)
\end{aligned}$$

TODO: Prosa, zum Beispiel, dass Methodennamen nicht in $\mathit{free}\,(e)$ auftauchen

Definition 2.6 (Substitution) Sei $e \in \mathit{Exp}$ und $\mathit{id} \in \mathit{Id}$.

(a) Für $e' \in \mathit{Exp}$ erhält man den Ausdruck $e'[e/\mathit{id}]$, der aus e' durch *Substitution* von e für id entsteht, durch folgende Erweiterung von Definition 1.7.

$$\begin{aligned}
(r\#m)[e/\mathit{id}] &= (r[e/\mathit{id}])\#m \\
(e_1\#m)[e/\mathit{id}] &= (e_1[e/\mathit{id}])\#m \\
(\mathbf{object}\,(\mathit{id}')\ r\ \mathbf{end})[e/\mathit{id}] &= \begin{cases} \mathbf{object}\,(\mathit{id}')\ r\ \mathbf{end} & \text{falls } \mathit{id} = \mathit{id}' \\ \mathbf{object}\,(\widetilde{\mathit{id}})\ r[\widetilde{\mathit{id}}/\mathit{id}'][e/\mathit{id}]\ \mathbf{end} & \text{sonst} \end{cases} \\
&\quad\ \text{mit } \widetilde{\mathit{id}} \notin (\mathit{free}\,(r) \setminus \{\mathit{id}'\}) \cup \{\mathit{id}\} \cup \mathit{free}\,(e) \\
\{\langle \mathit{id}_i = e_i{}^{1\leq i\leq n}\rangle\}[e/\mathit{id}] &= \begin{cases} \mathbf{object}\,(\mathit{id})\ \omega\langle \mathit{id}_i = e_i[e/\mathit{id}]^{1\leq i\leq n}\rangle\ \mathbf{end} \\ \quad \text{falls } e = \mathbf{object}\,(\mathit{id})\ \omega\ \mathbf{end} \\ \{\langle \mathit{id}_i = e_i[e/\mathit{id}]^{1\leq i\leq n}\rangle\} \\ \quad \text{sonst} \end{cases}
\end{aligned}$$

(b) Für $r \in \mathit{Row}$ ist die Reihe $r[e/\mathit{id}]$, die aus r durch *Substitution* von e für id entsteht, durch Induktion über die Grösse von r wie folgt definiert.

$$\begin{aligned}
\epsilon[e/\mathit{id}] &= \epsilon \\
(\mathbf{val}\ \mathit{id}' = e';\ r_1)[e/\mathit{id}] &= \begin{cases} \mathbf{val}\ \mathit{id}' = e'[e/\mathit{id}];\ r_1 & \text{falls } \mathit{id} = \mathit{id}' \\ \mathbf{val}\ \widetilde{\mathit{id}} = e'[e/\mathit{id}];\ r_1\{\widetilde{\mathit{id}}/\mathit{id}'\}[e/\mathit{id}] & \text{sonst} \end{cases} \\
&\quad\ \text{mit } \widetilde{\mathit{id}} \notin (\mathit{free}\,(r_1) \setminus \{\mathit{id}'\}) \cup \{\mathit{id}\} \cup \mathit{free}\,(e) \\
(\mathbf{method}\ m = e';\ r_1)[e/\mathit{id}] &= \mathbf{method}\ m = e'[e/\mathit{id}];\ r_1[e/\mathit{id}]
\end{aligned}$$

Die Reihe $\omega\langle id_1 = e_1; \ldots; id_n = e_n\rangle$ im ersten Fall der Anwendung einer Substitution auf eine Duplikation entsteht aus ω durch Einsetzung der Ausdrücke $e_1, \ldots, e_n$ auf den rechten Seiten der Attribute $id_1, \ldots, id_n$. Die Reihe $r\{\widetilde{id}/id'\}$ im zweiten Fall der Anwendung einer Substitution auf eine Attributdeklaration ensteht aus r durch Attributumbenennung von id' nach $\widetilde{id}$, einer speziellen Form der Substitution von $\widetilde{id}$ für id', wobei auch die Attributnamen auf den linken Seiten von Duplikationen substituiert werden. Die exakten Definition der Reiheneinsetzung und Attributumbenennung folgen später.

Gebundene Umbenennung oder *alpha@α-Konversion* (**TODO:** guter Literaturverweis), Idee beim ersten Fall der Anwendung einer Substitution auf eine Duplikation u.a. Zusammenhang zwischen id und **object** (id) ω **end** bei der Substitution.

Definition 2.7 (Attributumbenennung) Der Ausdruck $e\{id'/id\}$, der aus e durch *Attributumbenennung* von id nach id' entsteht, und die Reihe $r\{id'/id\}$, die aus r durch *Attributumbenennung* von id nach id' entsteht, sind durch Induktion über die Grössen von e und r wie folgt definiert.

$$
\begin{array}{rcl}
c\{id'/id\} & = & c \\
id''\{id'/id\} & = & id''[id'/id] \\
(e_1\, e_2)\{id'/id\} & = & (e_1\{id'/id\})\,(e_2\{id'/id\}) \\
(\lambda id''.e)\{id'/id\} & = & \begin{cases} \lambda id''.e, \\ \text{ falls } id = id'' \\ \lambda id'''.e[id'''/id'']\{id'/id\}, \\ \text{ sonst, mit } id''' \notin \mathit{free}\,(\lambda id''.e) \cup \{id, id'\} \end{cases} \\
(\mathbf{rec}\, id''.e)\{id'/id\} & = & \begin{cases} \mathbf{rec}\, id''.e, \\ \text{ falls } id = id'' \\ \mathbf{rec}\, id'''.e[id'''/id']\{id'/id\}, \\ \text{ sonst, mit } id''' \notin \mathit{free}\,(\mathbf{rec}\, id''.e) \cup \{id, id'\} \end{cases} \\
(\mathbf{let}\, id'' = e_1\, \mathbf{in}\, e_2)\{id'/id\} & = & \begin{cases} \mathbf{let}\, id'' = e_1\{id'/id\}\, \mathbf{in}\, e_2, \\ \text{ falls } id = id'' \\ \mathbf{let}\, id''' = e_1\{id'/id\}\, \mathbf{in}\, e_2[id'''/id'']\{id'/id\}, \\ \text{ sonst, mit } id''' \notin \mathit{free}\,(\lambda id''.e_2) \cup \{id, id'\} \end{cases} \\
(\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2)\{id'/id\} & = & \mathbf{if}\, e_0\{id'/id\}\, \mathbf{then}\, e_1\{id'/id\}\, \mathbf{else}\, e_2\{id'/id\}
\end{array}
$$

$$
\begin{array}{rcl}
(r\#m)\{id'/id\} & = & (r\{id'/id\})\#m \\
(e\#m)\{id'/id\} & = & (e\{id'/id\})\#m \\
(\mathbf{object}\,(id'')\ r\ \mathbf{end})\{id'/id\} & = & \begin{cases} \mathbf{object}\,(id'')\ r\ \mathbf{end}, \\ \text{ falls } id = id'' \\ \mathbf{object}\,(id''')\ r[id'''/id''][id'/id]\ \mathbf{end}, \\ \text{ sonst, mit } id''' \notin (\mathit{free}\,(r) \setminus \{id''\}) \cup \{id, id'\} \end{cases} \\
\{\langle id_i = e_i{}^{1 \le i \le n} \rangle\}\{id'/id\} & = & \{\langle id_i[id'/id] = e_i\{id'/id\}^{1 \le i \le n} \rangle\}
\end{array}
$$

$$
\begin{array}{rcl}
\epsilon\{id'/id\} & = & \epsilon \\
(\mathbf{val}\ id'' = e;\ r)\{id'/id\} & = & \begin{cases} \mathbf{val}\ id'' = e\{id'/id\};\ r, \\ \text{ falls } id = id'' \\ \mathbf{val}\ id''' = e\{id'/id\};\ r\{id'''/id''\}\{id'/id\}, \\ \text{ sonst, mit } id''' \notin (\mathit{free}\,(r) \setminus \{id'\}) \cup \{id, id'\} \end{cases} \\
(\mathbf{method}\ m = e;\ r)\{id'/id\} & = & \mathbf{method}\ m = e\{id'/id\};\ r\{id'/id\}
\end{array}
$$

Die Attributumbenennung ähnelt der Substitution in vielen Punkten, allerdings gibt es einige wichtige Unterschiede. Zum einen bezieht sich die Attributumbenennung ausschliesslich auf Attribute, d.h. es wird stets ein Attributname durch einen neuen Attributnamen ersetzt und gebundene Umbenennung von Namen, die mittels λ, **rec**, **let** oder **object** gebunden sind, erfolgt stets durch Substitution. Desweiteren endet die Umbenennung von Attributen sobald ein neues Objekt angetroffen wird, da Umbenennungen in einem umgebenden Objekt keinen Einfluss auf die Attribute des inneren Objekts haben, und im Falle der Duplikationen sind auch die linken Seiten, also die Namen der Attribute, die beim Duplizieren mit neuen Werten versehen werden sollen, von der Umbenennung betroffen.

Definition 2.8 (Reiheneinsetzung) Die Reihe $r\langle id_1 = e_1; \ldots; id_n = e_n\rangle$ entsteht aus $r \in \mathit{Row}$ durch *Reiheneinsetzung*, indem die rechten Seiten der Attribute $id_1, \ldots, id_n \in \mathit{Id}$ durch die Ausdrücke $e_1, \ldots, e_n \in \mathit{Exp}$ ersetzt werden, ist durch Induktion über die Größe von r wie folgt definiert.

$$
\begin{array}{rcl}
\epsilon\langle id_i = {e_i}^{1\le i\le n}\rangle & = & \epsilon \\
(\textbf{method}\ m = e;\ r)\langle id_i = {e_i}^{1\le i\le n}\rangle & = & \textbf{method}\ m = e;\ r\langle id_i = {e_i}^{1\le i\le n}\rangle \\
(\textbf{val}\ id = e;\ r)\langle id_i = {e_i}^{1\le i\le n}\rangle & = & \begin{cases} \textbf{val}\ id' = e;\ r'\langle id_i = {e_i}^{1\le i\le n}\rangle, \\ \ \text{falls } id \notin \{id_1, \ldots, id_n\} \\ \textbf{val}\ id' = e_j;\ r'\langle id_i = {e_i}^{1\le i\le n \wedge i\neq j}\rangle, \\ \ \text{falls } \exists j \in \{1, \ldots, n\} : id = id_j \end{cases}
\end{array}
$$

$$\text{mit } id' \notin (\mathit{free}\,(r) \setminus \{id\}) \cup \bigcup_{i=1}^{n} \mathit{free}\,(e_i) \text{ und } r' = r\{id'/id\}$$

Bei der Einsetzung in Attributdeklarationen muss gegebenenfalls eine gebundene Umbenennung durchgeführt werden, da die Ausdrücke $e_1, \ldots, e_n$ nicht unbedingt abgeschlossen sind und durch naive Einsetzung dieser Ausdrücke, frei vorkommende Namen durch zuvor in der Reihe deklarierte Attribute neu gebunden werden könnten. Hierbei ist zu beachten, dass, falls id nicht frei in $e_1, \ldots, e_n$ vorkommt, keine Umbenennung durchgeführt werden muss.

Beispiel 2.1 Betrachten wir die Reiheneinsetzung

$$(\textbf{val}\ x = 1;\ \textbf{val}\ y = 2;\ \epsilon)\langle y = x\rangle$$

so würde bei einer naiven Ersetzung von 2 durch x die Reihe

$$\textbf{val}\ x = 1;\ \textbf{val}\ y = x;\ \epsilon$$

entstehen. Das zuvor freie Vorkommen von x wäre nun an das Attribut x gebunden, was in der Regel zu einer Änderung der Semantik des Gesamtausdrucks führen würde. Gebundene Umbenennung des Attributs x liefert das gewünschte Ergebnis

$$\textbf{val}\ x' = 1;\ \textbf{val}\ y = x;\ \epsilon.$$

2.2.2 Small step Semantik

TODO: Einleitende Worte, mit Verweis auf $\mathcal{L}_f$

Definition 2.9 Ein *small step* für $\mathcal{L}_o$ist eine Formel der Gestalt $e \rightarrow_e e'$ mit $e, e' \in \mathit{Exp}$ oder $r \rightarrow_r r'$ mit $r, r' \in \mathit{Row}$.

TODO: Ein bisschen Erläuterung

Definition 2.10 (Gültige small steps für $\mathcal{L}_o$) Ein small step, $e \rightarrow_e e'$ mit $e, e' \in \mathit{Exp}$ oder $r \rightarrow_r r'$ mit $r, r' \in \mathit{Row}$, heißt *gültig* für $\mathcal{L}_o$, wenn er sich mit den small step Regeln von $\mathcal{L}_f$ (vgl. Definition 1.9), sowie den folgenden small step Regeln für Objekte

$$\text{(OBJECT-EVAL)} \quad \frac{r \rightarrow_r r'}{\textbf{object}\,(id)\; r\; \textbf{end} \rightarrow_e \textbf{object}\,(id)\; r'\; \textbf{end}}$$

$$\text{(SEND-EVAL)} \quad \frac{e \rightarrow_e e'}{e\#m \rightarrow_e e'\#m}$$

$$\text{(SEND-UNFOLD)} \quad (\textbf{object}\,(id)\; \omega\; \textbf{end})\#m \rightarrow_e (\omega[\textbf{object}\,(id)\; \omega\; \textbf{end}/id])\#m$$

$$\text{(SEND-ATTR)} \quad (\textbf{val}\; id = v;\, \omega)\#m \rightarrow_e (\omega[v/id])\#m$$

$$\text{(SEND-SKIP)} \quad (\textbf{method}\; m' = e;\, \omega)\#m \rightarrow_e \omega\#m \quad \text{falls } m' \neq m \vee m \in dom_m(\omega)$$

$$\text{(SEND-EXEC)} \quad (\textbf{method}\; m' = e;\, \omega)\#m \rightarrow_e e \quad \text{falls } m' = m \wedge m \notin dom_m(\omega)$$

und den small step Regeln für Reihen

$$\text{(ATTR-LEFT)} \quad \frac{e \rightarrow_e e'}{\textbf{val}\; id = e;\, r \rightarrow_r \textbf{val}\; id = e';\, r}$$

$$\text{(ATTR-RIGHT)} \quad \frac{r \rightarrow_r r'}{\textbf{val}\; id = v;\, r \rightarrow_r \textbf{val}\; id = v;\, r'} \quad \text{falls } id \notin dom_a(r)$$

$$\text{(ATTR-RENAME)} \quad \textbf{val}\; id = v;\, r \rightarrow_r \textbf{val}\; id' = v;\, r\{id'/id\} \quad \text{falls } id \in dom_a(r),\ \text{wobei } id' \notin \{id\} \cup \mathit{free}(r) \cup dom_a(r)$$

$$\text{(METHOD-RIGHT)} \quad \frac{r \rightarrow_r r'}{\textbf{method}\; m = e;\, r \rightarrow_r \textbf{method}\; m = e;\, r'}$$

herleiten lässt.

TODO: Erläuterung der Regeln

Im Folgenden betrachten wir $\rightarrow_e$ als eine Relation

$$\rightarrow_e \subseteq \mathit{Exp} \times \mathit{Exp}$$

und $\rightarrow_r$ als eine Relation

$$\rightarrow_r \subseteq \mathit{Row} \times \mathit{Row}$$

und benutzen die Infixnotation $e \rightarrow_e e'$ anstelle von $(e, e') \in \rightarrow_e$ und $r \rightarrow_r r'$ anstelle von $(r, r') \in \rightarrow_r$. Statt $\rightarrow_e$ und $\rightarrow_r$ schreiben wir auch $\rightarrow$, wenn wir uns auf beide Relationen beziehen oder es aus dem Zusammenhang hervorgeht, welche Relation gemeint ist.

Wir schreiben $\xrightarrow{+}$ für den transitiven und $\xrightarrow{*}$ für den reflexiven transitiven Abschluss der Reltation $\rightarrow$, d.h. $\xrightarrow{+}$ bildet eine endliche und $\xrightarrow{*}$ bildet eine nicht-leere endliche Folge von small steps.

Ferner schreiben wir $e \not\to_e$, wenn kein e' existiert, so dass $(e, e') \in\to_e$, und $r \not\to_r$, wenn kein r' existiert, so dass $(r, r') \in\to_r$. Damit sind wir nun in der Lage die erste einfache Eigenschaft der small step Semantik zu formulieren.

Lemma 2.1

(a) $v \not\to$ *für alle* $v \in Val_e$.

(b) $\omega \not\to$ *für alle* $\omega \in Val_r$.

Beweis: Der Beweis ergibt sich durch simultane vollständige Induktion über die Grösse von Werten v und Reihenwerten ω: Für Konstanten, Namen, Abstraktionen und die leere Reihe ist die Behauptung unmittelbar klar, da diese weder in den Axiomen noch in den Konklusionen der Regeln links von $\to$ stehen.

Für $v = op\, v'$ kommt nur ein small step mit (APP-LEFT) oder (APP-RIGHT) in Frage. Dann müsste aber in der Prämisse dieser Regel ein small step für op oder v' stehen, was nach Induktionsvorraussetzung nicht möglich ist.

Im Fall von $v = \textbf{object}\ (id)\ \omega'\ \textbf{end}$ kommt nur ein small step mit (OBJECT-EVAL) in Frage. Nach Induktionsvorraussetzung existiert aber kein small step für ω.

Für $\omega = \textbf{val}\ id = v';\ \omega'$ kommt nur ein small step mit (ATTR-LEFT), (ATTR-RIGHT) oder (ATTR-RENAME) in Frage. Nach Induktionsvoraussetzung existiert weder für ω' noch für v' ein small step, die Regeln (ATTR-LEFT) und (ATTR-RIGHT) können folglich nicht angewandt werden. Nach Definition müssen die Attributnamen in ω paarweise verschieden sein, also $id \notin dom_a\,(\omega')$ gelten, was die Anwendung von (ATTR-RENAME) ausschliesst.

Es bleibt der Fall $\omega = \textbf{method}\ m = e;\ \omega'$, in dem nur ein small step mit (METHOD-RIGHT) in Frage kommt. Nach Induktionsvoraussetzung existiert aber kein small step für ω'. □

Die Auswertung eines Ausdrucks beschreibt man als eine Aneinanderreihung gültiger small steps, die sich mit den small step Regeln herleiten lassen. Die folgende Definition beschreibt diesen Zusammenhang.

Definition 2.11 (Berechnung)

(a) Eine *Berechnungsfolge* ist eine endliche oder unendliche Folge von small steps $e_1 \to e_1 \to \ldots$.

(b) Eine *Berechnung* des Ausdrucks e ist eine *maximale*, mit e beginnende Berechnungsfolge, d.h., eine Berechnungsfolge, die sich nicht weiter fortsetzen lässt.

TODO: Überleitung.

Satz 2.2 (Eindeutigkeit des Übergangsschritts) *Für jeden Ausdruck e existiert höchstens ein $e' \in Exp$ mit $e \to e'$ und für jede Reihe r existiert höchstens ein $r' \in Row$ mit $r \to r'$.*

Beweis: Simultane vollständige Induktion über die Grösse von Ausdrücken e und Reihen r. □

Korollar 2.3

(a) *Für jeden Ausdruck e existiert genau eine Berechnung.*

(b) *Für jede endliche Berechnung $e_1 \to \ldots \to e_n$ ist das Resultat e_n eindeutig durch e_1 bestimmt.*

Beweis: Folgt trivialerweise aus der Eindeutigkeit des Übergangsschritts. □

TODO: Was bringt das nun?

2.2.3 Big step Semantik

TODO: Einführung.

Definition 2.12 Ein *big step* ist eine Formel der Form $e \Downarrow_e v$ mit $e \in \mathit{Exp}, v \in \mathit{Val}_e$ oder $r \Downarrow_r \omega$ mit $r \in \mathit{Row}, \omega \in \mathit{Val}_r$.

Definition 2.13 (Gültige big steps) Ein big step, $e \Downarrow_e v$ mit $e \in \mathit{Exp}, v \in \mathit{Val}_e$ oder $r \Downarrow_r \omega$ mit $r \in \mathit{Row}, \omega \in \mathit{Val}_r$, heisst *gültig*, wenn er sich mit den *big step Standard-Regeln*

(VAL) $\quad v \Downarrow_e v$

(OP) $\quad op\, n_1\, n_2 \Downarrow_e op^I(n_1, n_2)$

(BETA-V) $\quad \dfrac{e[v/id] \Downarrow_e v'}{(\lambda id.e)\, v \Downarrow_e v'}$

(APP) $\quad \dfrac{e_1 \Downarrow_e v_1 \quad e_2 \Downarrow_e v_2 \quad v_1\, v_2 \Downarrow_e v}{e_1\, e_2 \Downarrow_e v}$

(UNFOLD) $\quad \dfrac{e[\mathbf{rec}\; id.e/id] \Downarrow_e v}{\mathbf{rec}\; id.e \Downarrow_e v}$

(LET) $\quad \dfrac{e_1 \Downarrow_e v_1 \quad e_2[v_1/id] \Downarrow_e v_2}{\mathbf{let}\; id = e_1\; \mathbf{in}\; e_2 \Downarrow_e v_2}$

(COND-TRUE) $\quad \dfrac{e_0 \Downarrow_e \mathit{true} \quad e_1 \Downarrow_e v}{\mathbf{if}\; e_0\; \mathbf{then}\; e_1\; \mathbf{else}\; e_2 \Downarrow_e v}$

(COND-FALSE) $\quad \dfrac{e_0 \Downarrow_e \mathit{false} \quad e_2 \Downarrow_e v}{\mathbf{if}\; e_0\; \mathbf{then}\; e_1\; \mathbf{else}\; e_2 \Downarrow_e v}$

sowie den folgenden *big step Regeln für Objekte*

(OBJECT) $$\frac{r \Downarrow_r \omega}{\textbf{object}\,(id)\ r\ \textbf{end} \Downarrow_e \textbf{object}\,(id)\ \omega\ \textbf{end}}$$

(SEND) $$\frac{e \Downarrow_e \textbf{object}\,(id)\ \omega\ \textbf{end} \quad \omega[\textbf{object}\,(id)\ \omega\ \textbf{end}/id]\#m \Downarrow_e v}{e\#m \Downarrow_e v}$$

(SEND-ATTR) $$\frac{\omega[v/id]\#m \Downarrow_e v'}{(\textbf{val}\ id = v;\ \omega)\#m \Downarrow_e v'}$$

(SEND-SKIP) $$\frac{m' \neq m \vee m \in dom_m(\omega) \quad \omega\#m \Downarrow_e v}{(\textbf{method}\ m' = e;\ \omega)\#m \Downarrow_e v}$$

(SEND-EXEC) $$\frac{m' = m \wedge m \notin dom_m(\omega) \quad e \Downarrow_e v}{(\textbf{method}\ m' = e;\ \omega)\#m \Downarrow_e v}$$

und den *big step Regeln für Reihen*

(OMEGA) $$\omega \Downarrow_r \omega$$

(ATTR) $$\frac{id \notin dom_a(r) \quad e \Downarrow_e v \quad r \Downarrow_r \omega}{\textbf{val}\ id = e;\ r \Downarrow_r \textbf{val}\ id = v;\ \omega}$$

(RENAME) $$\frac{id \in dom_a(r) \quad e \Downarrow_e v \quad r\{id'/id\} \Downarrow_r \omega}{\textbf{val}\ id = e;\ r \Downarrow_r \textbf{val}\ id' = v;\ \omega}$$
wobei $id' \notin \{id\} \cup free(r) \cup dom_a(r)$

(METHOD) $$\frac{r \Downarrow_r \omega}{\textbf{method}\ m = e;\ r \Downarrow_r \textbf{method}\ m = e;\ \omega}$$

herleiten lässt.

Die zweite Prämisse der Regel (RENAME) könnte eigentlich auch entfallen, denn der so berechnete Wert v wird im weiteren Verlauf des Programms nicht mehr benötigt. Allerdings sollen die big und small step Semantiken bezüglich des Ergebnisses einer Berechnung äquivalent sein, und in der small step Semantik kann (ATTR-RENAME) erst angewendet werden, wenn hinter dem Attribut ein Wert steht. Mit imperativen Konzepten spielt es später durchaus eine Rolle ob und wann ein Ausdruck ausgewertet wird.

Analog zur small step Semantik betrachten wir $\Downarrow_e \subseteq Exp \times Val_e$ und $\Downarrow_r \subseteq Row \times Val_r$ als Relationen und schreiben lediglich $\Downarrow$, wenn wir uns auf beide Relationen beziehen oder wenn aus dem Zusammenhang hervorgeht welche Relation gemeint ist.

Weiter schreiben wir $e \not\Downarrow$, falls kein big step für e existiert, und $r \not\Downarrow$, falls kein big step für r existiert.

2.2.4 Äquivalenz der Semantiken

Der nun folgende Satz wird es uns erlauben die meisten Eigenschaften der small step Semantik auf die big step Semantik zu übertragen.

Satz 2.4 (Äquivalenzsatz)

(a) $\forall e \in Exp : \forall v \in Val_e : e \Downarrow v \Leftrightarrow e \xrightarrow{*} v$

(b) $\forall r \in Row : \forall \omega \in Val_r : r \Downarrow \omega \Leftrightarrow r \xrightarrow{*} \omega$

Beweis: Der Beweis erfolgt in zwei Schritten. Zunächst zeigen wir, dass für jeden big step eine äquivalente[1] endliche Berechnung, d.h. eine maximale, endliche Berechnungsfolge von small steps, existiert. Anschliessend zeigen wir, dass für jede endliche Berechnung ein äquivalenter big step existiert.

„$\Rightarrow$" Diese Richtung beweisen wir mittels simultaner Induktion über die Längen der Herleitungen der big steps $e \Downarrow v$ und $r \Downarrow \omega$, und Fallunterscheidung nach der zuletzt angewandten big step Regel.

TODO

„$\Leftarrow$" Der zweite Teil des Beweises erfolgt mittels simultaner Induktion über die Länge der Berechnungen $e \xrightarrow{*} v$ und $r \xrightarrow{*} \omega$, und Fallunterscheidung nach der Form von e und r.

TODO □

Wie bereits angedeutet lassen sich mit Hilfe des Äquivalenzsatzes bereits bewiesene Eigenschaften der small step Semantik auf die big step Semantik übertragen.

Korollar 2.5

(a) Existiert ein big step $e \Downarrow v$ mit $e \in Exp, v \in Val_e$, so ist v durch e eindeutig bestimmt.

(b) Existiert ein big step $r \Downarrow \omega$ mit $r \in Row, \omega \in Val_r$, so ist ω durch r eindeutig bestimmt.

Beweis: Folgt mit dem Äquivalenzsatz unmittelbar aus der Eindeutigkeit des Übergangsschritts. □

2.3 Ein einfaches Typsystem

Einfach – entspricht dem *einfach getypten λ-Kalkül* – explizit getypt, keine komplizierten Mechanismen wie rekursive Typen, Subtyping oder Polymorphie.

2.3.1 Syntax der Sprache $\mathcal{L}_o^t$

Definition 2.14 (Typen) Die Menge $Type_e$ aller *Typen* τ von $\mathcal{L}_o^t$ ist durch die kontextfreie Grammatik

$$
\begin{array}{rcll}
\tau & ::= & \textbf{bool} \mid \textbf{int} \mid \textbf{unit} & \text{Basistypen} \\
 & \mid & \tau_1 \to \tau_2 & \text{Funktionstypen} \\
 & \mid & \langle \phi \rangle & \text{Objekttypen}
\end{array}
$$

[1] Der Begriff der Äquivalenz beudetet in diesem Zusammenhang, dass die Ausführung eines Programms mit den unterschiedlichen Semantiken zum gleichen Ergebnis führen.

und die Menge $Type_r$ aller *Reihentypen* ϕ von $\mathcal{L}_o^t$ ist durch

$$\begin{array}{rcl}\phi & ::= & \emptyset \\ & | & m : \tau;\, \phi_1\end{array}$$

definiert, wobei die Methodennamen in einem Reihentyp ϕ paarweise verschieden sein müssen.

Explizit getypt, d.h. Typen müssen in die Syntax der Programmiersprache aufgenommen werden, indem die Produktionen für Abstraktion, Rekursion und Objekte durch die folgenden „getypten Versionen“ ersetzt werden.

$$\begin{array}{rcl}e & ::= & \lambda id : \tau.e_1 \\ & | & \mathbf{rec}\, id : \tau.e_1 \\ & | & \mathbf{object}\, (id : \tau)\; r\; \mathbf{end}\end{array}$$

Die Reihenfolge in der die Methodentypen in einem Reihentypen aufgelistet werden ist irrelevant, d.h. es gilt

$$m_1 : \tau_1;\; m_2 : \tau_2;\; \phi = m_2 : \tau_2;\; m_1 : \tau_1;\; \phi$$

für alle $m_1, m_2 \in \mathit{Method}$, $\tau_1, \tau_2 \in \mathit{Type}_e$ und $\phi \in \mathit{Type}_r$.

Definition 2.15 Die *Vereinigung* $\phi_1 \oplus \phi_2$ der beiden Reihentypen

$$\phi_1 = m_1 : \tau_1;\; \ldots m_n : \tau_n;\; \emptyset$$

und

$$\phi_2 = m'_1 : \tau'_1;\; \ldots m'_l : \tau'_l;\; \emptyset$$

ist definiert als

$$\begin{array}{rcll}\phi_1 \oplus \phi_2 & = & m_{i_1} : \tau_{i_1};\; \ldots m_{i_x} : \tau_{i_x}; & \text{(gemeinsame Methodentypen)} \\ & & m_{j_1} : \tau_{j_1};\; \ldots m_{j_y} : \tau_{j_y}; & \text{(Methodentypen exklusiv in } \phi_1\text{)} \\ & & m'_{k_1} : \tau'_{k_1};\; \ldots m'_{k_z} : \tau'_{k_z}; & \text{(Methodentypen exklusiv in } \phi_2\text{)} \\ & & \emptyset & \end{array}$$

mit

$$\begin{array}{rcl}\{m_{i_1}, \ldots, m_{i_x}\} & = & \{m_1, \ldots, m_n\} \cap \{m'_1, \ldots, m'_l\}, \\ \{m_{j_1}, \ldots, m_{j_y}\} & = & \{m_1, \ldots, m_n\} \setminus \{m_{i_1}, \ldots, m_{i_x}\} \text{ und} \\ \{m'_{k_1}, \ldots, m'_{k_z}\} & = & \{m'_1, \ldots, m'_l\} \setminus \{m_{i_1}, \ldots, m_{i_x}\},\end{array}$$

falls $\forall i \in \{i_1, \ldots, i_x\} : \tau_i = \tau'_i$.

Insbesondere führt die Vereinigung zweier inkompatibler Reihentypen, die also auf dem Schnitt ihrer Methoden nicht übereinstimmen, während der Typüberprüfung zu einem Typfehler, da in diesem Fall kein Vereinigungstyp definiert ist. Wir werden auch im Rahmen der Beweise der Sätze über die Typsicherheit auf diese Definition zurückgreifen.

Definition 2.16 Zwei Reihentypen

$$\phi_1 = m_1 : \tau_1; \ldots m_n : \tau_n; \emptyset$$

und

$$\phi_2 = m'_1 : \tau'_1; \ldots m'_l : \tau'_l; \emptyset$$

heissen *disjunkt*, wenn $\{m_1, \ldots, m_n\} \cap \{m'_1, \ldots, m'_l\} = \emptyset$.

Hiermit lässt sich nun die erste einfache Eigenschaft unserer getypten Programmiersprache $\mathcal{L}_o^t$ formulieren.

Proposition 2.6 *Die Vereinigung $\phi_1 \oplus \phi_2$ zweier disjunkter Reihentypen $\phi_1, \phi_2 \in Type_r$ ist stets definiert.*

Beweis: Folgt vermöge Definition 2.16 unmittelbar aus Definition 2.15, da

$$\forall i \in \emptyset : \tau_i = \tau'_i$$

allgemeingültig ist. □

Syntaktischer Zucker

TODO: Syntaktischer Zucker entsprechend mit Typen versehen.

$$\textbf{method}\ m\ (id_1 : \tau_1) \ldots (id_n : \tau_n) = e;\ r \quad \text{für} \quad \textbf{method}\ m = \lambda id_1 : \tau_1. \ldots \lambda id_n : \tau_n.e;\ r$$

2.3.2 Typsystem der Sprache $\mathcal{L}_o^t$

Definition 2.17 (Typurteile für Konstanten) Ein *Typurteil für Konstanten* ist eine Formel der Gestalt $c :: \tau$ mit $c \in Const, \tau \in Type_e$. Die *gültigen* Typurteile sind durch die folgenden Axiome festgelegt.

(BOOL)	$b :: \textbf{bool}$	
(INT)	$n :: \textbf{int}$	
(UNIT)	$() :: \textbf{unit}$	
(AOP)	$op :: \textbf{int} \to \textbf{int} \to \textbf{int}$	falls $op \in \{+, -, *\}$
(ROP)	$op :: \textbf{int} \to \textbf{int} \to \textbf{bool}$	falls $op \in \{<, >, \leq, \geq, =\}$

Das Typurteil $c :: \tau$ liest man als „c hat Typ τ“. Man beachte, dass gemäss obiger Definition jede Konstante $c \in Const$ einen eindeutigen Typ $\tau \in Type_e$ hat.

Definition 2.18 (Typumgebungen)

(a) Die Menge *Tag* aller *Typmarkierungen* κ ist definiert durch die folgende kontextfreie Grammatik.

$$\kappa \ ::= \ attr \mid name \mid self$$

(b) Eine *Typumgebung* ist eine partielle Funktion $\Gamma : \mathit{Id} \hookrightarrow \mathit{Tag} \times \mathit{Type}_e$ mit endlichem Definitionsbereich.

(c) Sei Γ eine Typumgebung, dann ist die partielle Funktion $\Gamma^\star$ wie folgt definiert.

$$\Gamma^\star(id) = \begin{cases} (name, \tau) & \text{falls } \exists \kappa \in \mathit{Tag} : \Gamma(id) = (\kappa, \tau) \\ \uparrow & \text{sonst} \end{cases}$$

(d) Sei Γ eine Typumgebung, dann ist die partielle Funktion Γ^- wie folgt definiert.

$$\Gamma^-(id) = \begin{cases} (name, \tau) & \text{falls } \Gamma(id) = (name, \tau) \\ \uparrow & \text{sonst} \end{cases}$$

Die Typmarkierungen werden benötigt, um Bindungen, die durch die Einträge einer Typumgebung repräsentiert werden, mit einem Kontext zu versehen. Mittels der Typmarkierungen lässt sich beispielweise bis zu einem gewissen Punkt unterscheiden, ob eine Bindung durch einen **let**-Ausdruck oder eine Attributdeklaration zu Stande gekommen ist. Dieser Zusammenhang wird im Folgenden anhand der Typregeln deutlich.

Die partiellen Funktionen $\Gamma^\star$ und Γ^- stellen Einschränkungen der Typumgebung Γ dar. Die Funktion $\Gamma^\star$ ersetzt alle *attr*- und $self$-Markierungen, die in Γ vorhanden sind, durch $name$-Markierungen, während die Funktion Γ^- alle derart markierten Einträge entfernt. Die Anwendung dieser eingeschränkten Typumgebungen wird ebenfalls im Rahmen des Regelwerks für das Typsystem deutlich.

Bemerkung 2.1 Wir verwenden die Listenschreibweise $[id_1 : (\kappa_1, \tau_1), \ldots, id_n : (\kappa_n, \tau_n)]$ mit $n \geq 0$ für die Typumgebung Γ mit den Eigenschaften

(a) $\mathit{dom}\,(\Gamma) = \{id_1, \ldots, id_n\}$ und

(b) $\forall i \in \{1, \ldots, n\} : \Gamma(id_i) = (\kappa_i, \tau_i)$.

Die Listenschreibweise verdeutlicht, dass eine Typumgebung letztlich nichts anderes ist als eine „Tabelle“, in der die Typen der bereits bekannten Namen eingetragen sind[2]. Das „Nachschlagen“ in der Tabelle Γ können wir durch die Funktionsanwendung zum Ausdruck bringen: Falls $id \in \mathit{dom}\,(\Gamma)$, dann liefert $\Gamma(id)$ die Markierung κ und den Typ τ, die für den Namen id in der Tabelle Γ eingetragen wurden, sonst ist $\Gamma(id)$ undefiniert.

Definition 2.19 Seien $\kappa \in \mathit{Tag}$, $\tau \in \mathit{Type}_e$, $id \in \mathit{Id}$ und $\Gamma : \mathit{Id} \hookrightarrow \mathit{Tag} \times \mathit{Type}_e$ eine Typumgebung. Dann bezeichnet $\Gamma[(\kappa, \tau)/id]$ die Typumgebung $\Gamma' : \mathit{Id} \hookrightarrow \mathit{Tag} \times \mathit{Type}_e$ mit den folgenden Eigenschaften.

- $\mathit{dom}\,(\Gamma') = \mathit{dom}\,(\Gamma) \cup \{id\}$
- $\Gamma'(id) = (\kappa, \tau)$
- $\forall id' \in \mathit{dom}\,(\Gamma') \setminus \{id\} : \Gamma'(id') = \Gamma(id')$

[2] In der Compilerbau-Literatur findet man deshalb häufig die Bezeichnung „Symboltabelle“.

$\Gamma[(\kappa,\tau)/id]$ ist also die Typumgebung, die sich von Γ – wenn überhaupt – nur darin unterscheidet, dass an der Stelle id der Eintrag (κ,τ) steht. Insbesondere wird dadurch ein eventuell schon in Γ vorhandener Eintrag für id durch (κ,τ) überschrieben.

Nach diesen Vorbereitungen können wir nun die Wohlgetyptheit von Ausdrücken und Reihen der Programmiersprache $\mathcal{L}_o^t$ formulieren. Dazu definieren wir zunächst was wir unter einem Typurteil verstehen wollen und geben anschliessend ein Regelwerk an, mit dem sich gültige Typurteile für $\mathcal{L}_o^t$ herleiten lassen.

Definition 2.20 (Typurteile für Ausdrücke und Reihen)

(a) Ein *Typurteil für Ausdrücke* ist eine Formel der Gestalt $\Gamma \triangleright_e e :: \tau$ mit $\Gamma : Id \hookrightarrow Tag \times Type_e$, $e \in Exp$, $\tau \in Type_e$.

(b) Ein *Typurteil für Reihen* ist eine Formel der Gestalt $\Gamma \triangleright_r r :: \phi$ mit $\Gamma : Id \hookrightarrow Tag \times Type_e$, $r \in Row$, $\phi \in Type_r$.

TODO: Prosa

Definition 2.21 (Gültige Typurteile für $\mathcal{L}_o^t$) Ein Typurteil $\Gamma \triangleright_e e :: \tau$ oder $\Gamma \triangleright_r r :: \psi$ heisst *gültig* für $\mathcal{L}_o^t$, wenn es sich mit den Typregeln für die funktionale Kernsprache

(ID) $\quad \Gamma \triangleright_e id :: \tau \quad \text{falls } \exists \kappa \in Tag : \Gamma(id) = (\kappa,\tau)$

(CONST) $$\frac{c :: \tau}{\Gamma \triangleright_e c :: \tau}$$

(APP) $$\frac{\Gamma \triangleright_e e_1 :: \tau_2 \to \tau \quad \Gamma \triangleright e_2 :: \tau_2}{\Gamma \triangleright_e e_1\, e_2 :: \tau}$$

(ABSTR) $$\frac{\Gamma[(name,\tau)/id] \triangleright_e e_1 :: \tau_1}{\Gamma \triangleright_e \lambda id : \tau.e_1 :: \tau \to \tau_1}$$

(REC) $$\frac{\Gamma[(name,\tau)/id] \triangleright_e e_1 :: \tau}{\Gamma \triangleright_e \mathbf{rec}\, id : \tau.e_1 :: \tau}$$

(LET) $$\frac{\Gamma \triangleright_e e_1 :: \tau_1 \quad \Gamma[(name,\tau_1)/id] \triangleright_e e_2 :: \tau_2}{\Gamma \triangleright_e \mathbf{let}\, id = e_1\, \mathbf{in}\, e_2 :: \tau_2}$$

(COND) $$\frac{\Gamma \triangleright_e e_0 :: \mathbf{bool} \quad \Gamma \triangleright_e e_1 :: \tau \quad \Gamma \triangleright_e e_2 :: \tau}{\Gamma \triangleright_e \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau}$$

sowie den Typregeln für Objekte

(SEND-ROW) $$\frac{\Gamma \triangleright_e r :: (m : \tau; \phi)}{\Gamma \triangleright_e r\#m :: \tau}$$

(SEND) $$\frac{\Gamma \triangleright_e e_1 :: \langle m : \tau; \phi\rangle}{\Gamma \triangleright_e e_1\#m :: \tau}$$

(OBJECT) $$\frac{\Gamma^\star[(self,\tau)/id] \triangleright_r r :: \phi \quad \tau = \langle\phi\rangle}{\Gamma \triangleright_e \mathbf{object}\, (id : \tau) \;::\; \mathbf{end}\tau}$$

(DUPL) $$\frac{\exists id : \Gamma(id) = (self,\tau) \quad \forall 1 \leq i \leq n : \Gamma \triangleright_e e_i :: \tau_i \wedge \Gamma(id_i) = (attr,\tau_i)}{\Gamma \triangleright_e \{\langle id_1 = e_1; \ldots; id_n = e_n\rangle\} :: \tau}$$

und den Typregeln für Reihen

$$\text{(EMPTY)} \qquad \Gamma \rhd_r \epsilon :: \emptyset$$

$$\text{(ATTR)} \qquad \frac{\Gamma^- \rhd_e e :: \tau \quad \Gamma[(attr, \tau)/id] \rhd_r r_1 :: \phi}{\Gamma \rhd_r \textbf{val } id = e;\, r_1 :: \phi}$$

$$\text{(METHOD)} \qquad \frac{\Gamma \rhd_e e :: \tau \quad \Gamma \rhd_r r_1 :: \phi}{\Gamma \rhd_r \textbf{method } m = e;\, r_1 :: (m : \tau;\, \emptyset) \oplus \phi}$$

herleiten lässt.

Im Folgenden schreiben wir statt $\Gamma \rhd_e e :: \tau$ kurz $\Gamma \rhd e :: \tau$ und statt $\Gamma \rhd_r r :: \phi$ kurz $\Gamma \rhd r :: \phi$, da es aus dem Zusammenhang stets ersichtlich ist, ob es sich um ein Typurteil für Ausdrücke oder ein Typurteil für Reihen handelt.

TODO: Erläuterung des Regelwerks, Beispiel einer Typherleitung

Definition 2.22 (Wohlgetyptheit)

(a) Ein Ausdruck $e \in Exp$ heisst *wohlgetypt in* Γ, wenn es ein $\tau \in Type_e$ gibt, so dass gilt: $\Gamma \rhd e :: \tau$.

(b) Eine Reihe $r \in Row$ heisst *wohlgetypt in* Γ, wenn es ein $\phi \in Type_r$ gibt, so dass gilt: $\Gamma \rhd r :: \phi$.

TODO: Prosa

Satz 2.7 (Typeindeutigkeit)

(a) Für jede Typumgebung Γ *und jeden Ausdruck* $e \in Exp$ *existiert höchstens ein Typ* $\tau \in Type_e$ *mit* $\Gamma \rhd e :: \tau$.

(b) Für jede Typumgebung Γ *und jede Reihe* $r \in Row$ *existiert höchstens ein Reihentyp* $\phi \in Type_r$ *mit* $\Gamma \rhd r :: \phi$.

Beweis: Der Beweis erfolgt mittels simultaner Induktion über die Grössen von e und r und Fallunterscheidung nach der Form von e bzw. r. Dabei ist jeweils zu zeigen, dass für jede syntaktische Form nur höchstens eine Typregel anwendbar ist. Dies folgt trivialerweise aus der Definition der Typregeln, da jede syntaktische Form eines Ausdrucks oder einer Reihe nur jeweils in der Konklusion höchstens einer Typregel steht. □

Korollar 2.8 *Sei* Γ *eine Typumgebung.*

(a) Wenn $e \in Exp$ *wohlgetypt ist in* Γ, *dann existiert genau ein* $\tau \in Type_e$ *mit* $\Gamma \rhd e :: \tau$.

(b) Wenn $r \in Row$ *wohlgetypt ist in* Γ, *dann existiert genau ein* $\phi \in Type_r$ *mit* $\Gamma \rhd e :: \phi$.

Beweis: Folgt vermöge Definition 2.22 direkt aus Satz 2.7. □

Diese Erkenntnis erlaubt es uns einen Algorithmus zu formulieren, der überprüft, ob ein Ausdruck in einer gegebenen Typumgebung wohlgetypt ist, und falls ja, den eindeutigen Typ des Ausdrucks liefert.

TODO: Typalgorithmus

Typsicherheit

Nach diesen einleitenden Betrachtung wollen wir nun zeigen, dass die Programmiersprache $\mathcal{L}_o^t$ typsicher ist. Typsicherheit bedeutet in diesem Zusammenhang, dass die Berechnung eines wohlgetypten abgeschlossenen (**TODO:** err, nicht unbedingt, mal sehen!) Ausdrucks nicht steckenbleiben kann.

Die Berechnung eines Ausdrucks in $\mathcal{L}_o^t$ ist wie in $\mathcal{L}_o$ definiert. Die Definition der Menge Val_e muss an die neue Syntax angepasst werden, in dem die Produktionen durch die entsprechenden Produktionen mit Typen ersetzt werden.

$$\begin{array}{rcl} v & ::= & \lambda id : \tau.e \\ & \mid & \mathbf{object}\,(id : \tau)\ \omega\ \mathbf{end} \end{array}$$

Die small step Regeln aus $\mathcal{L}_o$ werden übernommen, wobei die Regeln (BETA-V), (UNFOLD), (OBJECT-EVAL) und (SEND-UNFOLD) an die neue Syntax angepasst werden müssen.

$$\begin{array}{ll} \text{(BETA-V)} & (\lambda id : \tau.e)v \to_e e[v/id] \\ \text{(UNFOLD)} & \mathbf{rec}\, id : \tau.e \to_e e[\mathbf{rec}\, id : \tau.e/id] \\ \text{(OBJECT-EVAL)} & \dfrac{r \to_r r'}{\mathbf{object}\,(id : \tau)\ r\ \mathbf{end} \to_e \mathbf{object}\,(id : \tau)\ r'\ \mathbf{end}} \\ \text{(SEND-UNFOLD)} & (\mathbf{object}\,(id : \tau)\ \omega\ \mathbf{end})\#m \to_e (\omega[\mathbf{object}\,(id : \tau)\ \omega\ \mathbf{end}/id])\#m \end{array}$$

Es gilt zu beachten, dass der Typ in diesen Ausdrücken keinerlei Einfluss auf den small step hat, d.h. Typen spielen zur Laufzeit keine Rolle. Sie werden lediglich zur Compilezeit während der statischen Typüberprüfung benötigt. Die folgende Definition verdeutlicht diese Sachverhalt.

Definition 2.23 Für jeden Ausdruck[3] $e \in Exp(\mathcal{L}_o^t)$ sei $erase\,(e) \in Exp(\mathcal{L}_o)$ der Ausdruck, der aus e durch Entfernen aller Typen entsteht, also

$$\begin{array}{rcl} erase\,(c) & = & c \\ erase\,(id) & = & id \\ erase\,(e_1\, e_2) & = & erase\,(e_1)\ erase\,(e_2) \\ erase\,(\lambda id : \tau.e_1) & = & \lambda id.erase\,(e_1) \\ erase\,(\mathbf{rec}\, id : \tau.e_1) & = & \mathbf{rec}\, id.erase\,(e_1) \\ erase\,(\mathbf{let}\, id = e_1\, \mathbf{in}\, e_2) & = & \mathbf{let}\, id = erase\,(e_1)\, \mathbf{in}\, erase\,(e_2) \\ erase\,(\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) & = & \mathbf{if}\, erase\,(e_0)\, \mathbf{then}\, erase\,(e_1)\, \mathbf{else}\, erase\,(e_2) \\ erase\,(r\#m) & = & erase\,(r)\#m \\ erase\,(e_1\#m) & = & erase\,(e_1)\#m \\ erase\,(\mathbf{object}\,(id : \tau)\ r\ \mathbf{end}) & = & \mathbf{object}\,(id)\ erase\,(r)\ \mathbf{end} \\ erase\,(\{\langle id_i = e_i{}^{1 \le i \le n}\rangle\}) & = & \{\langle id_i = erase\,(e_i)^{1 \le i \le n}\rangle\} \end{array}$$

[3] Wir benutzen im Folgenden die Schreibweise $M(\mathcal{L})$ zur Verdeutlichung, dass wir uns auf die Definition der Menge M in der Sprache $\mathcal{L}$ beziehen.

und für jede Reihe r in $\mathcal{L}_o^t$ sei $erase\,(r)$ die Reihe in $\mathcal{L}_o$, die aus r durch Entfernen aller Typen ensteht, also

$$\begin{array}{rcl} erase\,(\epsilon) & = & \epsilon \\ erase\,(\mathbf{val}\ id = e;\ r_1) & = & \mathbf{val}\ id = erase\,(e);\ erase\,(r_1) \\ erase\,(\mathbf{method}\ m = e;\ r_1) & = & \mathbf{method}\ m = erase\,(r);\ erase\,(r_1). \end{array}$$

Bei der Funktion $erase : Exp(\mathcal{L}_o^t) \to Exp(\mathcal{L}_o)$ handelt es sich also offensichtlich um einen Übersetzer zwischen der Sprache $\mathcal{L}_o^t$ und der Sprache $\mathcal{L}_o$. Bei Übersetzern zwischen Programmiersprachen interessieren wir uns im wesentlichen dafür, dass diese *semantikerhaltend* sind, d.h. dass ein Ausdruck durch eine Übersetzung in eine andere Programmiersprache keine neue Bedeutung erlangt. Der folgende Satz sichert diese Eigenschaft für die Funktion *erase*.

Satz 2.9

(a) $\forall e, e' \in Exp(\mathcal{L}_o^t) : e \to e' \Leftrightarrow erase\,(e) \to erase\,(e')$

(b) $\forall r, r' \in Row(\mathcal{L}_o^t) : r \to r' \Leftrightarrow erase\,(r) \to erase\,(r')$

Beweis: TODO: Nicht schön, könnte man allerdings wie folgt argumentieren:
Folgt unmittelbar aus der Tatsache, dass die small step Semantik für $\mathcal{L}_o^t$ mit der small step Semantik für $\mathcal{L}_o$ bis auf syntaktische Details – die Typen – übereinstimmt. □

Korollar 2.10 $\forall e \in Exp(\mathcal{L}_o^t), v \in Val_e(\mathcal{L}_o^t) : e \xrightarrow{*} v \Leftrightarrow erase\,(e) \xrightarrow{*} erase\,(v)$

Beweis: Trivial. □

TODO: Erklären warum macht man das?
TODO: Prosa
TODO: Lemmata

Satz 2.11 (Preservation)

(a) Wenn $\Gamma \triangleright e :: \tau$ *und* $e \to e'$, *dann gilt auch* $\Gamma \triangleright e' :: \tau$.

(b) Wenn $\Gamma \triangleright r :: \phi$ *und* $r \to r'$, *dann gilt auch* $\Gamma \triangleright r' :: \phi$.

Beweis: Nach Vorraussetzung gilt $\Gamma \triangleright e :: \tau$, $\Gamma \triangleright r :: \phi$, $e \to e'$ und $r \to r'$. Wir zeigen $\Gamma \triangleright e' :: \tau$ und $\Gamma \triangleright r' :: \phi$ durch simultane Induktion über die Länge der Herleitungen der small steps $e \to e'$ und $r \to r'$ und Fallunterscheidung nach der zuletzt angewandten small step Regel.

1. $op\, n_1\, n_2 \to op^I(n_1, n_2)$ mit (OP).

   Dann ist zu unterscheiden zwischen arithmetischen und relationalen Operatoren. Sei also $op \in \{+, -, *\}$, dann gilt nach Vorraussetzung $\Gamma \triangleright op\, n_1\, n_2 :: \mathbf{int}$ und insbesondere $op :: \mathbf{int} \to \mathbf{int} \to \mathbf{int}$, wegen (AOP). Folglich ist $op^I(n_1, n_2) \in Int$, und es gilt $\Gamma \triangleright op^I(n_1, n_2) :: \mathbf{int}$.

   Sei andererseits $op \in \{<, >, \leq, \geq, =\}$, dann gilt nach Voraussetzung $\Gamma \triangleright op\, n_1\, n_2 :: \mathbf{bool}$ und insbesondere $op :: \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$, wegen (ROP). Folglich ist $op^I(n_1, n_2) \in Bool$, also gilt $\Gamma \triangleright op^I(n_1, n_2) :: \mathbf{bool}$.

2. $(\lambda id : \tau.e)\, v \to e[v/id]$ mit (BETA-V).

   TODO:

3. $e_1\, e_2 \to e_1'\, e_2$ mit (APP-LEFT) aus $e_1 \to e_1'$.

   $\Gamma \triangleright e_1\, e_2 :: \tau$ kann nur mit Typregel (APP) aus $\Gamma \triangleright e_2 :: \tau'$ und $\Gamma \triangleright e_1 :: \tau' \to \tau$ folgen. Nach Induktionsvoraussetzung gilt dann $\Gamma \triangleright e_1' :: \tau' \to \tau$ und es folgt $\Gamma \triangleright e_1'\, e_2 :: \tau$ mit Typregel (APP).

4. $v_1\, e_2 \to v_1\, e_2'$ mit (APP-RIGHT) aus $e_2 \to e_2'$.

   Analog zum vorhergehenden Fall kann $\Gamma \triangleright v_1\, e_2 :: \tau$ nur mit Typregel (APP) aus $\Gamma \triangleright v_1 :: \tau' \to \tau$ und $\Gamma \triangleright e_2 :: \tau'$. Nach Induktionsvoraussetzung gilt dann $\Gamma \triangleright e_2' :: \tau'$ und mit Typregel (APP) folgt $\Gamma \triangleright v_1\, e_2' :: \tau$.

5. $\mathbf{rec}\, id : \tau.e \to e[\mathbf{rec}\, id : \tau.e/id]$ mit (UNFOLD).

   TODO:

6. $\mathbf{let}\, id = e_1\, \mathbf{in}\, e_2 \to \mathbf{let}\, id = e_1'\, \mathbf{in}\, e_2$ mit (LET-EVAL) aus $e_1 \to e_1'$.

   $\Gamma \triangleright \mathbf{let}\, id = e_1\, \mathbf{in}\, e_2 :: \tau$ kann nur mit Typregel (LET) aus $\Gamma \triangleright e_1 :: \tau_1$ und $\Gamma[(name, \tau_1)/id] \triangleright e_2 :: \tau$ folgen. Nach Induktionsvoraussetzung folgt dann $\Gamma \triangleright e_1' :: \tau_1$ und somit $\Gamma \triangleright \mathbf{let}\, id = e_1'\, \mathbf{in}\, e_2 :: \tau$ mit Typregel (LET).

7. $\mathbf{let}\, id = v_1\, \mathbf{in}\, e_2 \to e_2[v_1/id]$ mit (LET-EXEC).

   TODO:

8. $\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \to \mathbf{if}\, e_0'\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2$ mit (COND-EVAL) aus $e_0 \to e_0'$.

   $\Gamma \triangleright \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau$ kann nur mit Typregel (COND) aus $\Gamma \triangleright e_0 :: \mathbf{bool}$, $\Gamma \triangleright e_1 :: \tau$ und $\Gamma \triangleright e_2 :: \tau$ folgen. Nach Induktionsvoraussetzung gilt $\Gamma \triangleright e_0' :: \mathbf{bool}$ und es folgt $\Gamma \triangleright \mathbf{if}\, e_0'\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau$ mit Typregel (COND).

9. $\mathbf{if}\, true\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \to e_1$ mit (COND-TRUE).

   $\Gamma \triangleright \mathbf{if}\, true\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau$ kann nur mit Typregel (COND) aus $\Gamma \triangleright true :: \mathbf{bool}$, $\Gamma \triangleright e_1 :: \tau$ und $\Gamma \triangleright e_2 :: \tau$ folgen. Die zweite Prämisse liefert die Typerhaltung.

10. $\mathbf{if}\, false\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \to e_2$ mit (COND-FALSE).

    Analog zum vorhergehenden Fall.

11. $\mathbf{object}\, (id : \tau)\; r\; \mathbf{end} \to \mathbf{object}\, (id : \tau)\; r'\; \mathbf{end}$ mit (OBJECT-EVAL) aus $r \to r'$.

    $\Gamma \triangleright \mathbf{object}\, (id : \tau)\; r\; \mathbf{end} :: \tau$ kann nur aus $\Gamma^\star[(self, \tau)/id] \triangleright r :: \phi$ und $\tau = \langle\phi\rangle$ mit Typregel (OBJECT) folgen. $\Gamma^\star[(self, \tau)/id] \triangleright r' :: \phi$ gilt nach Induktionsvoraussetzung und mit Typregel (OBJECT) folgt $\Gamma \triangleright \mathbf{object}\, (id : \tau)\; r'\; \mathbf{end} :: \tau$.

12. $e\#m \to e'\#m$ mit (SEND-EVAL) aus $e \to e'$.

    $\Gamma \triangleright e\#m :: \tau$ kann nur mit Typregel (SEND) aus $\Gamma \triangleright e :: m : \tau; \phi$ folgen. Nach Induktionsvoraussetzung gilt $\Gamma \triangleright e' :: m : \tau; \phi$ und es folgt $\Gamma \triangleright e'\#m :: \tau$ mit Typregel (SEND).

13. $(\textbf{object}\,(id\ :\ \tau)\ \omega\ \textbf{end})\#m \rightarrow (\omega[\textbf{object}\,(id\ :\ \tau)\ \omega\ \textbf{end}/id])\#m$ mit (SEND-UNFOLD).

    TODO:

14. $(\textbf{val}\ id = v;\ \omega)\#m \rightarrow (\omega[v/id])\#m$ mit (SEND-ATTR).

    TODO:

15. $(\textbf{method}\ m' = e;\ \omega)\#m \rightarrow \omega\#m$ mit (SEND-SKIP).

    $\Gamma \triangleright (\textbf{method}\ m' = e;\ \omega)\#m :: \tau$ kann lediglich mit Typregel (SEND-ROW) aus $\Gamma \triangleright \textbf{method}\ m' = r;\ \omega :: (m\ :\ \tau;\ \phi)$ folgen. Gemäß small step Regel (SEND-SKIP) sind zwei Fälle zu unterscheiden: Für $m \neq m'$ ergibt sich direkt, dass $\Gamma \triangleright \omega :: (m\ :\ \tau;\ \phi)$ gelten muss, also $\Gamma \triangleright \omega\#m :: \tau$ mit Typregel (SEND). Für $m = m' \wedge m \in dom_m(\omega)$ muss gelten $\Gamma \triangleright \omega :: (m : \tau';\ \phi')$ und es bleibt zu zeigen, dass $\tau = \tau'$ gilt.

    Angenommen $\tau \neq \tau'$, dann wäre nach Typregel (METHOD)

    $$(m : \tau;\ \phi) = (m : \tau;\ \emptyset) \oplus (m : \tau';\ \phi')$$

    und somit gemäß Definition 2.15 nicht definiert, da die Methodentypen auf dem Schnitt nicht übereinstimmen. Dies steht im Widerspruch zur Voraussetzung, dass $m : \tau;\ \phi$ definiert ist.

16. $(\textbf{method}\ m = e;\ \omega)\#m \rightarrow e$ mit (SEND-EXEC).

    $\Gamma \triangleright (\textbf{method}\ m = e;\ \omega)\#m :: \tau$ kann ausschliesslich mit Typregel (SEND-ROW) aus $\Gamma \triangleright \textbf{method}\ m = e; \omega :: (m\ :\ \tau;\ \phi)$ folgen, und die Prämisse liefert die Typerhaltung.

17. $\textbf{val}\ id = e;\ r \rightarrow \textbf{val}\ id = e';\ r$ mit (ATTR-LEFT) aus $e \rightarrow e'$.

    $\Gamma \triangleright \textbf{val}\ id = e;\ r :: \phi$ kann nur mit Typregel (ATTR) aus $\Gamma[(attr, \tau)/id] \triangleright r :: \phi$ und $\Gamma^- \triangleright e :: \tau$ folgen. Nach Induktionsvoraussetzung gilt $\Gamma^- \triangleright e' :: \tau$ und mit Typregel (ATTR) folgt $\Gamma \triangleright \textbf{val}\ id = e';\ r :: \phi$.

18. $\textbf{val}\ id = v;\ r \rightarrow \textbf{val}\ id = v;\ r'$ mit (ATTR-RIGHT) aus $r \rightarrow r'$.

    $\Gamma \triangleright \textbf{val}\ id = v;\ r :: \phi$ kann nur aus $\Gamma^- \triangleright v :: \tau$ und $\Gamma[(attr, \tau)/id] \triangleright r :: \phi$ mit Typregel (ATTR) folgen. Nach Induktionsvoraussetzung gilt $\Gamma[(attr, \tau)/id] \triangleright r' :: \phi$ und mit Typregel (ATTR) folgt $\Gamma \triangleright \textbf{val}\ id = v;\ r' :: \phi$.

19. $\textbf{val}\ id = v;\ r \rightarrow \textbf{val}\ id' = v;\ r\{id'/id\}$ mit (ATTR-RENAME).

    TODO:

20. $\textbf{method}\ m = e;\ r \rightarrow \textbf{method}\ m = e;\ r'$ mit (METHOD-RIGHT) aus $r \rightarrow r'$.

    $\Gamma \triangleright \textbf{method}\ m = e;\ r :: \phi$ kann nur mit Typregel (METHOD) aus $\Gamma \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi'$ folgen. Nach Induktionsvoraussetzung gilt dann $\Gamma \triangleright r' :: \phi'$ und somit folgt $\Gamma \triangleright \textbf{method}\ m = e;\ r' :: \phi$ mit Typregel (METHOD). $\square$

Index

A

Abgeschlossenheit 7
Äquivalenzsatz . . . . . 19 f.
α-Konversion . . . . . *siehe* Gebundene Umbenennung
Attributumbenennung . . . . . 13 f.
Ausdruck 6, 10

B

Berechnung 17
Berechnungsfolge 17
big step 18

D

Definitionsbereich 11

F

Frei vorkommende Namen 7, 12

G

Gebundene Umbenennung . . . . . 13 f.

M

Methodenname 10

N

Name 5

P

Preservation 27

R

Reihe 10
Reiheneinsetzung 13, 15
Reihentyp 21
Reihenwert 11

S

small step . . . . . 8, 15 f.
Substitution 8, 12, 14

T

Typ 20
Typmarkierung 22
Typregeln . . . . . 24 f.
Typsicherheit 26
Typumgebung 23
Typurteil für Ausdrücke 24
Typurteil für Konstanten 22
Typurteil für Reihen 24

V

Vereinigung 21

W

Wert 11

Literaturverzeichnis

[Rém02] Rémy, Didier: Using, Understanding, and Unraveling the OCaml Language. From Practice to Theory and Vice Versa. In: *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures.* London, UK : Springer-Verlag, 2002. – ISBN 3–540–44044–5, S. 413–536

[Sie04] Sieber, Kurt: *Theorie der Programmierung I.* Universität Siegen, 2004. – Vorlesungsskript

[Sie06] Sieber, Kurt: *Theorie der Programmierung I.* Universität Siegen, 2006. – Vorlesungsskript

Erklärung

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

Siegen, den 6. März 2007 ____________________________

(Benedikt Meurer)