Universität Siegen
Institut für Theoretische Informatik
Fachbereich Mathematik

SS 2007

Diplomarbeit

Theoretische Grundlagen der Objektorientierung

Benedikt Meurer

10. Juli 2007

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

Inhaltsverzeichnis

Einleitung

Narf, [Rém02, 50f].

1 Grundlagen

1.1 Mathematische Grundlagen

Seien A, B Mengen, $a \in A$, $b \in B$ und $f : A \rightharpoonup B$ eine partielle Funktion. Dann bezeichnet $f[{}^{b}/_{a}]$ die partielle Funktion $g : A \rightharpoonup B$ mit den folgenden Eigenschaften.

- $dom\,(g) = dom\,(f) \cup \{a\}$
- $g(a) = b$
- $\forall a' \in dom\,(g) : a' \neq a \Rightarrow g(a') = f(a')$

TODO: Potenzmenge definieren.
TODO: Eigenschaften von Relationen: Refl., Trans., Symmetrie, Antisymmetrie, Quasiordnungen, Halbordnungen, totale Ordnungen, Äquivalenzrelationen

1.1.1 Fixpunkttheorie

Dieser Abschnitt bietet eine kurze Einführung in die wichtigstens Begriffe der Fixpunkttheorie, und stellt insbesondere den Fixpunktsatz von Kleene vor. Die folgenden Ausführungen basieren im wesentlichen auf dem Material der Vorlesungen „Theorie der Programmierung III“ [Sie07] und „Semantik von Programmiersprachen“ [Kin05, S.85ff].

Definition 1.1 (Obere Schranken und Suprema) Sei $(D, \sqsubseteq)$ eine reflexive Ordnung, und $\Delta \subseteq D$.

(a) Ein Element $d \in D$ heisst *obere Schranke* von Δ, wenn $d \sqsubseteq \delta$ für alle $\delta \in \Delta$ gilt.

(b) Die kleinste obere Schranke von Δ wird als *Supremum* von Δ bezeichnet, geschrieben als $\bigsqcup \Delta$.

Entsprechend der oberen Schranke definiert man dann den Begriff der *unteren Schranke* und das *Infimum* als größte untere Schranke.

Definition 1.2 (Untere Schranken und Infima) Sei $(D, \sqsubseteq)$ eine reflexive Ordnung, und $\Delta \subseteq D$.

(a) Ein Element $d \in D$ heisst *untere Schranke* von Δ, wenn $d \sqsubseteq \delta$ für alle $\delta \in \Delta$ gilt.

(b) Die größte untere Schranke von Δ wird als *Infimum* von Δ bezeichnet, geschrieben als $\bigsqcap \Delta$.

TODO: Prosa

Definition 1.3 (Semantischer Bereich) Eine reflexive Ordnung $(D, \sqsubseteq)$ heisst *semantischer Bereich* , wenn für jede (unendliche) aufsteigende Folge $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ mit $d_n \in D$ das Supremum $\bigsqcup_{n\in\mathbb{N}} d_n$ existiert. Besitzt D darüber hinaus ein kleinstes Element, so bezeichnet man $(D, \sqsubseteq)$ als *semantischen Bereich mit kleinstem Element*. Das kleinste Element wird dann mit $\bot_D$ bezeichnet.

TODO: Prosa

Definition 1.4 (Monotone Abbildungen) Seien $(D_1, \sqsubseteq_1)$ und $(D_2, \sqsubseteq_2)$ reflexive Ordnungen. Eine totale Abbildung $f : D_1 \to D_2$ heisst *monoton*, wenn für alle $d, d' \in D_1$ mit $d \sqsubseteq_1 d'$ auch $f(d) \sqsubseteq_2 f(d')$ gilt.

Definition 1.5 (Stetige Abbildungen) Seien $(D_1, \sqsubseteq_1)$ und $(D_2, \sqsubseteq_2)$ semantische Bereiche. Eine Abbildung $f : D_1 \to D_2$ heisst *stetig*, wenn für jede nicht-leere aufsteigende Folge $d_0 \sqsubseteq_1 d_1 \sqsubseteq_1 d_2 \sqsubseteq_1 \ldots$ in D_1 das Supremum von $\{f(d_n) \mid n \in \mathbb{N}\}$ existiert und wenn gilt $\bigsqcup_{n\in\mathbb{N}} f(d_n) = f(\bigsqcup_{n\in\mathbb{N}} d_n)$.

Die Notation $\bigsqcup_{n\in\mathbb{N}} f(d_n)$ ist eine gängige Kurzschreibweise für $\bigsqcup\{f(d_n) \mid n \in \mathbb{N}\}$. Aus der Definition der Stetigkeit folgt unmittelbar das folgende Lemma:

Lemma 1.1 *Seien $(D_1, \sqsubseteq_1)$ und $(D_2, \sqsubseteq_2)$ semantische Bereiche. Jede stetige Abbildung $f : D_1 \to D_2$ ist monoton.*

Beweis: Klar. □

Mit diesen Begriffen läßt sich nun der Fixpunktsatz von Kleene formulieren und beweisen. Im Gegensatz zum wohlbekannten Fixpunktsatz von Knaster und Tarski, dem sogenannten *Knaster-Tarski-Theorem* [Tar55], stellt man hier eine strengere Forderung an die Funktion, nämlich Stetigkeit statt nur Monotonie, und kommt deshalb mit einem semantischen Bereich aus, anstatt eines vollständigen Verbands.

Satz 1.1 (Fixpunktsatz von Kleene) *Sei $(D, \sqsubseteq)$ ein semantischer Bereich mit kleinstem Element $\bot$, und sei $f : D \to D$ eine stetige Abbildung. Dann existiert das Supremum $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$ und es ist der kleinste Fixpunkt von f.*

Der kleinste Fixpunkt einer Funktion f wird üblicherweise mit μf bezeichnet. Der größte Fixpunkt wird entsprechend mit νf bezeichnet.

Beweis: Wir beweisen zunächst durch vollständige Induktion, dass $f^n(\bot) \sqsubseteq f^{n+1}(\bot)$ für jedes $n \in \mathbb{N}$ gilt:

- $n = 0$

  $f^0(\bot) = \bot \sqsubseteq f(\bot) = f^1(\bot)$

- $n \rightsquigarrow n+1$

  Nach Induktionsvoraussetzung gilt $f^n(\bot) \sqsubseteq f^{n+1}(\bot)$. Nach Voraussetzung ist f stetig und damit gemäß Lemma 1.1 auch monoton, also gilt $f(f^n(\bot)) \sqsubseteq f(f^{n+1}(\bot))$. ∎

Die $f^n(\bot)$ bilden also eine aufsteigende Folge in $(D, \sqsubseteq)$. Da $(D, \sqsubseteq)$ ein semantischer Bereich ist, existiert damit nach Definition 1.3 das Supremum $d = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$.

Es bleibt zu zeigen, dass dieses Supremum d der kleinste Fixpunkt von f ist. Dazu zeigen wir zuerst, dass d ein Fixpunkt von f ist:

$$\begin{array}{rcll} f(d) & = & f(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)) & \\ & = & \bigsqcup_{n\in\mathbb{N}} f(f^n(\bot)) & \text{da } f \text{ stetig} \\ & = & \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) & \\ & = & \bigsqcup_{n\in\mathbb{N}} f^n(\bot) & \text{da } f^0(\bot) = \bot \text{ kleinstes Element von } D \\ & = & d & \end{array}$$

Zuletzt zeigen wir, dass d der kleinste Fixpunkt von f ist. Sei dazu $d' \in D$ ein beliebiger Fixpunkt von f. Durch vollständige Induktion zeigen wir, dass $f^n(\bot) \sqsubseteq d'$ für alle $n \in \mathbb{N}$ gilt:

- $n = 0$

  $f^0(\bot) = \bot \sqsubseteq d'$, denn $\bot$ ist das kleinste Element von D.

- $n \rightsquigarrow n+1$

  Nach Induktionsvoraussetzung gilt $f^n(\bot) \sqsubseteq d'$. Da f nach Voraussetzung eine stetige Abbildung ist, und damit nach Lemma 1.1 auch monoton, gilt ebenfalls $f^{n+1}(\bot) \sqsubseteq f(d')$, und somit folgt $f^{n+1}(\bot) \sqsubseteq d'$, denn d' ist nach Annahme ein Fixpunkt von f.

Damit ist d' eine obere Schranke von $\{f^n(\bot) \mid n \in \mathbb{N}\}$. Da d die kleinste obere Schranke von $\{f^n(\bot) \mid n \in \mathbb{N}\}$ ist (das Supremum), gilt $d \sqsubseteq d'$, was zu zeigen war. □

1.2 Die funktionale Programmiersprache $\mathcal{L}_f$

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.

TODO: Etwas zu *structual typing* erzählen, insb. der Unterschied zu Java, o.ä.

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

Die Programmiersprache $\mathcal{L}_f$ orientiert sich, wie auch alle weiteren in diesem Dokument betrachteten Sprachen, syntaktisch stark an der Programmiersprache O'Caml[1], da O'Caml als Grundlage für den Inhalt der Vorlesung „Theorie der Programmierung“ dient. Bis auf kleinere syntaktische Unterschiede handelt es sich bei $\mathcal{L}_f$ um die funktionalen Sprache Caml Light[2], den Vorgänger von O'Caml, allerdings verzichten wir auf einige Konzepte wie Pattern Matching und Exceptions, da diese für die Betrachtungen in diesem Dokument nicht relevant sind. (**TODO:** ggfs. später auf Exceptions eingehen)

Definition 1.6 Vorgegeben sei

(a) eine unendliche Menge *Var* von *Variablen* x,

(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 *Var* bestimmten Beschränkungen unterwerfen, die zumeist auf Einschränkungen der konkreten Syntax zurückzuführen sind, wie zum Beispiel, dass *Var* keine Schlüsselwörter der Sprache enthalten darf, fordern wir lediglich, dass *Var* 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.

[1] `http://caml.inria.fr/ocaml/`
[2] `http://caml.inria.fr/caml-light/`

Definition 1.7 (Abstrakte Syntax von $\mathcal{L}_f$) Die Menge *Op* aller Operatoren *op* ist definiert durch die kontextfreie Grammatik

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

die Menge *Const* aller Konstanten *c* durch

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

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

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

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 würden. Wir werden Laufzeitfehlerbehandlung später als Erweiterung der Kernsprache betrachten (**TODO:** Nur, wenn das Exception-Kapitel rein kommt).

Nichtsdestrotrotz ist die Programmiersprache $\mathcal{L}_f$ hinreichend mächtig. Im weiteren Verlauf dieses Kapitels werden die Funktionen *div* und *mod* vorgestellt, die als Grundlage für die Einführung der Operatoren für Ganzzahldivision und Divisionsrest als syntaktischer Zucker in die Sprache dienen könnten. (**TODO:** Was'n das für'n komischer Satz?)

Frei vorkommende Variablen und Substitution

TODO: Blah blub

Definition 1.8 (Frei vorkommende Variablen) Die Menge $\mathit{free}(e)$ aller im Ausdruck $e \in \mathit{Exp}$ *frei vorkommenden Variablen* ist wie folgt induktiv definiert.

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

TODO: Blub blah

Definition 1.9 Ein Ausdruck $e \in \mathit{Exp}$ 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 Variablen enthält. Zum Beispiel sind im Ausdruck

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

die beiden Vorkommen der Variablen 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 Variablen durch **let**-Ausdrücke gebunden und der Gesamtausdruck ist abgeschlossen.

Definition 1.10 (Substitution) Für $e, e' \in \mathit{Exp}$ und $x \in \mathit{Var}$ ist der Ausdruck $e'[{}^{e}/_{x}]$, der aus e' durch *Substitution* von e für x entsteht, wie folgt induktiv über die Grösse von e definiert.

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

Gemäß obiger Definition ist allerdings die Substitution in dieser Form eine partielle Funktion. Zum Beispiel ist

$$(\lambda x.x\, y)[^x/_y]$$

nicht definiert, da $x \in \mathit{free}\,(x)$. Wir treffen deshalb die folgende Vereinbarung (vgl. [Pie02, S. 71]).

Konvention 1.1 Ausdrücke, die sich lediglich in den Namen von gebundenen Variablen unterscheiden, sind in jeder Beziehung austauschbar.

Wenn wir nun im Beispiel die gebundene Variable x umbenennen, zum Beispiel in x', so erhalten wir

$$(\lambda x'.x'\, y)[^x/_y]$$

und die Substitution ist definiert, da $x' \notin \{y\} \cup \mathit{free}\,(x)$. Dieses Austauschen der Namen gebundener Variablen wird in der Literatur als *gebundene Umbenennung* oder auch α-*Konversion* (**TODO:** Quelle Church) bezeichnet.

Vermöge dieser Konvention ist die Substitution nun eine totale Funktion. Denn falls eine Bedingung für die Substitution nicht erfüllt ist, können im Ausdruck gebundene Variablen entsprechend umbenannt werden, so dass die Bedingung erfüllt wird. Insbesondere sei daran erinnert, dass *Var* mindestens abzählbar unendlich ist, und somit stets „ein neuer Name gefunden werden kann“.

Gemäß dieser Betrachtungen könnten wir die Anwendung der Substitution auf λ-Ausdrücke auch wie folgt definieren.

$$\begin{array}{rcl}(\lambda x'.e')[^e/_x] & = & \lambda x''.e'[^{x''}/_{x'}][^e/_x] \\ & & \text{mit } x'' \notin \{x\} \cup \mathit{free}\,(e) \cup \mathit{free}\,(\lambda x'.e')\end{array}$$

Und entsprechend könnten auch die Fälle für **rec**- und **let**-Ausdrücke angepasst werden. Weiter können wir zeigen, dass wir für jeden solchen Ausdruck in effektiver Weise ein x'' angeben können, welches die Bedingung erfüllt. Zum Beispiel verwendet das an der Universität Siegen entwickelte Lernwerkzeug TPML[3] den folgenden einfachen Algorithmus.

```
public String generateVar (String var, Set forbidden) {
  while (forbidden.contains (var))
    var = var + "'";
  return var;
}
```

[3] http://www.informatik.uni-siegen.de/theo/tpml/

Im Falle der λ-Abstraktion wird `generateVar` dann mit x' und $\{x\} \cup \mathit{free}\,(e) \cup \mathit{free}\,(\lambda x'.e')$ aufgerufen und liefert einen „neuen“ Namen. Dazu werden lediglich solange Hochstriche an den Namen angehängt, bis ein Name gefunden wird, der nicht in der Menge der verbotenen Namen auftaucht. Da die Menge der verbotenen Namen offensichtlich endlich ist, findet der Algorithmus stets nach endlich vielen Schritten einen Namen, der nicht in dieser Liste auftaucht.

Damit kann für die Substitution ein Algorithmus formuliert werden kann, der die notwendigen gebundenen Umbenennung bei Bedarf durchführt. Es bleibt also festzuhalten, dass die Substitution eine totale, berechenbare Funktion ist.

Der Aspekt der Berechenbarkeit der Substitution spielt allerdings für die theoretischen Betrachtungen eine untergeordnete Rolle. Entscheidend ist, dass wir gemäß der getroffenen Vereinbarung, die Substitution als eine totale Funktion betrachten können.

Syntaktischer Zucker

$e_1\ op\ e_2$	für	$op\ e_1\ e_2$	Infixnotation
$-e$	für	$0 - e$	unäres Minus

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

$$
\begin{aligned}
\mathit{div} \;&=\; \mathbf{rec}\,\mathit{div}.\lambda x.\lambda y.\\
&\qquad \mathbf{if}\, x < y\, \mathbf{then}\, 0\\
&\qquad \mathbf{else}\ (\mathit{div}\ (x - y)\ y) + 1
\end{aligned}
$$

$$
\mathit{mod} \;=\; \lambda x.\lambda y.x - y * (\mathit{div}\, x\, y)
$$

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

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

In der *operationellen Semantik* einer Programmiersprache beschreibt man die Auswertung von Ausdrücken als ein Verfahren zur *Umformung* oder *Vereinfachung* der Ausdrücke. Im Gegensatz dazu versucht man in der denotationellen Semantik die Konstrukte der Programmiersprache direkt durch mathematische Funktionen zu beschreiben. Denotationelle Semantiken bieten einige Vorteile gegenüber der Beschreibung durch operationelle Semantik, sie sind dafür allerdings schwer oder garnicht erweiterbar, so dass für alle in diesem Dokument beschriebenen Sprachen jeweils neue Semantiken angeben müssten. Aus diesem Grund verwenden wir ausschliesslich operationelle Semantik zur Beschreibung der Programmiersprache.

Für die Beschreibung der Umformung von Ausdrücken in präziser mathematischer Form bieten sich zwei Möglichkeiten an, nämlich

(a) eine *iterative* Definition, bei der man explizit die einzelnen Umformungsschritte angibt, die aneinandergereiht werden dürfen,

(b) oder eine *rekursive* Definition, bei der man das Ergebnis einer Umformung auf die Ergebnisse von *Teilumformungen* zurückführt.

Den iterativen Ansatz bezeichnet man als *small step Semantik*, den rekursiven Ansatz als *big step Semantik*. Die small step Semantik eignet sich besonders gut für theoretische Betrachtungen, während die big step Semantik besser geeignet ist als Grundlage für die Implementation eines Interpreters. Wir werden uns deshalb zunächst ausschliesslich mit der small step Semantik für die Programmiersprachen $\mathcal{L}_f$ und $\mathcal{L}_o$ beschäftigen, und später eine äquivalente big step Semantik für die Programmiersprache $\mathcal{L}_o$ entwickeln.

Bevor wir uns nun der Definition der small step Semantik zuwenden, benötigen wir noch einige allgemeine Definition. Zunächst müssen wir festlegen, wie die Operatoren der Sprache zu interpretieren sind.

Definition 1.11 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. Stattdessen nehmen wir an, dass diese Funktionen entsprechend ihrer intuitiven Bedeutung definiert sind. Zum Beispiel entspricht also die Interpretation des Stern-Operators der Multiplikation

$$*^I : Int \times Int \to Int,\ (x, y) \mapsto x \cdot y$$

und die Interpretation des $<$-Operators der charakteristischen Funktion der $<$-Relation auf den ganzen Zahlen

$$<^I : Int \times Int \to Bool,\ (x, y) \mapsto \begin{cases} true & \text{falls } x < y \\ false & \text{sonst} \end{cases}$$

wobei wir nach Vereinbarung zwischen $\mathbb{Z}$ und Int nicht unterscheiden.

Definition 1.12 Die Menge $Val \subseteq Exp$ aller Werte v ist durch die kontextfreie Grammatik

v	::=	c	Konstante
	|	x	Variable
	|	$op\, v_1$	partielle Applikation
	|	$\lambda x.e_1$	λ-Abstraktion

definiert.

Im Terminus funktionaler Programmiersprachen ist ein Wert ein vollständig ausgewerteter Ausdruck, der nicht weiter vereinfacht werden kann. Man beachte, dass insbesondere λ-Abstraktionen, also Funktionen, bereits Werte sind. Dies lässt sich intuitiv einfach dadurch erklären, dass eine Funktion erst nach Anwendung auf ein Argument weiter ausgewertet werden kann. Entsprechendes gilt für Operatoren mit einem Operanden, da hier die Auswertung erst erfolgen kann, wenn beide Operanden vorhanden sind.

In diesem Kontext ist es auch üblich partielle Applikationen als Funktionen zu lesen. Beispielsweise entspricht der Ausdruck $+1$ der Nachfolgerfunktion auf den ganzen Zahlen, also der Funktion, die 1 auf ihr Argument addiert.

Small step Semantik

Nach diesen Vorbereitungen können wir nun die *small step Regeln* für die Programmiersprache $\mathcal{L}_f$ formulieren. Zunächst definieren wir dazu, was wir unter einem small step verstehen.

Definition 1.13 Ein *small step* ist eine Formel der Gestalt $e \rightarrow_e e'$ mit $e, e' \in Exp$.

Um small steps herleiten zu können benötigen wir einen Satz von Regeln, der durch die folgende Definition bestimmt wird.

Definition 1.14 (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) $(\lambda x.e)\, v \rightarrow_e e[{}^{v}/_{x}]$

(App-Left) $$\frac{e_1 \rightarrow_e e_1'}{e_1\, e_2 \rightarrow_e e_1'\, e_2}$$

(App-Right) $$\frac{e_2 \rightarrow_e e_2'}{v_1\, e_2 \rightarrow_e v_1\, e_2'}$$

(Unfold) $\mathbf{rec}\, x.e \rightarrow_e e[{}^{\mathbf{rec}\, x.e}/_{x}]$

(Let-Eval) $$\frac{e_1 \rightarrow_e e_1'}{\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \rightarrow_e \mathbf{let}\, x = e_1'\, \mathbf{in}\, e_2}$$

(Let-Exec) $\mathbf{let}\, x = v_1\, \mathbf{in}\, e_2 \rightarrow_e e_2[{}^{v_1}/_{x}]$

(Cond-Eval) $$\frac{e_0 \rightarrow_e e_0'}{\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \rightarrow_e \mathbf{if}\, e_0'\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2}$$

(Cond-True) $\mathbf{if}\, \mathit{true}\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \rightarrow_e e_1$

(Cond-False) $\mathbf{if}\, \mathit{false}\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \rightarrow_e e_2$

Das Regelwerk entspricht den small step Regeln der Programmiersprache $\mathcal{L}_2$ aus der Vorlesung „Theorie der Programmierung“ ([Sie06]). Nachfolgend sollen die einzelnen Regeln kurz erläutert werden.

- (Op) besagt, dass die Anwendung eines Operators auf zwei ganze Zahlen das intuitive Ergebnis liefert, zum Beispiel $* 21\, 2 \rightarrow 42$.
- Die Regel (Beta-V), die als *β-value*-Regel bezeichnet wird, beschreibt die Parameterübergabe: Die Anwendung einer Funktion $\lambda x.e$ auf einen Wert v bewirkt, dass jedes Vorkommen des formalen Parameters x im Rumpf e durch den aktuellen Parameter v ersetzt wird. Zu beachten ist, dass (Beta-V) nur anwendbar ist, wenn der aktuelle Parameter ein Wert, also schon vollständig ausgewertet ist[4].
- Die Regel (App-Left) erlaubt es den linken Teil einer Applikation weiter auszuwerten, zum Beispiel solange, bis eine λ-Abstraktion erreicht ist.
- Die Regel (App-Right) erlaubt es das Argument einer Applikation auszuwerten[5].

[4] Man bezeichnet diese Art der Parameterübergabe als *call-by-value*. Die unausgewertete Parameterübergabe wird als *call-by-name* bezeichnet.

[5] Diese Regel wird unter anderem benötigt, um im Zusammenspiel mit (Beta-V) das *call-by-value*-Prinzip zu realisieren.

- (UNFOLD) faltet einen rekursiven Ausdruck auf, indem der vollständige Ausdruck für jedes Vorkommen des formalen Parameters x im Rumpf e eingesetzt wird.
- (LET-EVAL) wertet den Teilausdruck e_1 eines **let**-Ausdrucks aus.
- Die Regel (LET-EXEC) greift, sobald der erste Teilsausdruck ausgewertet ist, und setzt dessen Wert v_1 für den formalen Parameter x in e_2 ein.
- (COND-EVAL) erlaubt es, die Bedingung e_0 eines bedingten Ausdrucks auszuwerten.
- Die Regeln (COND-TRUE) und (COND-FALSE) greifen, sobald die Bedingung eines bedingten Ausdrucks zu einer boolschen Konstanten ausgewertet ist.

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. Die Vorlesung „Theorie der Programmierung“ orientiert sich so weit möglich an der Programmiersprache O'Caml[1], entsprechend orientieren sich auch die in diesem Kapitel entwickelten Erweiterungen syntaktisch und semantisch an O'Caml (vgl. [RV97], [RV98] und [Rém02]).

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

TODO: Prosa.

Definition 2.1

(a) Vorgegeben seien

- eine unendliche Menge *Method* von *Methodennamen* m,
- eine unendliche Menge *Attribute* von *Attributnamen* a
- und die einelementige Menge *Self* von *Objektnamen* *self*.

Die Mengen *Var*, *Attribute* und *Self* werden als disjunkt angenommen.

(b) Die Menge *Id* aller *Namen* *id* ist wie folgt definiert.

$$Id \quad = \quad Var \cup Attribute \cup Self$$

Für *Method* gelten ähnlich wie für die Menge *Var* keinerlei Einschränkungen, außer dass die Menge mindestens abzählbar unendlich sein muss. Eine Implementierung wird üblicherweise $Method = Var$ wählen.

TODO: Was zu *Attribute* und *Self* sagen.

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

[1] `http://caml.inria.fr/ocaml/`

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

r	::=	ϵ	leere Reihe
	|	**val** $a = e;\ r_1$	Attribut
	|	**method** $m = e;\ r_1$	Methode

und die Menge *Exp* aller *Ausdrücke* e von $\mathcal{L}_o$ erhält man, indem man die kontextfreie Grammatik für Ausdrücke von $\mathcal{L}_f$ wie folgt erweitert.

e	::=	*id*	Variable, Attribut- oder Objektname
	|	$e_1 \# m$	Methodenaufruf
	|	**object** (*self*) r **end**	Objekt
	|	$\{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\}$	Duplikation

Zur Vereinfachung treffen wir für Reihen die folgende Vereinbarung.

Konvention 2.1 Die Namen der in einer Reihe deklarierten Attribute sind paarweise verschieden.

Diese Vereinbarung stellt keine wirkliche Einschränkung der Sprache dar. In der konkreten Syntax können mehrere Attributdeklarationen mit dem gleichen Namen in derselben Reihe durchaus zugelassen werden. Bei der syntaktischen Analyse werden dann durch geeignete Umbenennung die Namen der Attributdeklarationen in einer Reihe disjunkt gemacht.

TODO: Überleitung

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

(a) Die Menge der Attributnamen $dom_a(r)$ der Reihe r ist wie folgt induktiv definiert.

$$\begin{aligned} dom_a(\epsilon) &= \emptyset \\ dom_a(\textbf{val}\ a = e;\ r) &= \{a\} \cup dom_a(r) \\ dom_a(\textbf{method}\ m = e;\ r) &= dom_a(r) \end{aligned}$$

(b) Die Menge der Methodennamen $dom_m(r)$ der Reihe r ist wie folgt induktiv definiert.

$$\begin{aligned} dom_m(\epsilon) &= \emptyset \\ dom_m(\textbf{val}\ a = e;\ r) &= dom_m(r) \\ dom_m(\textbf{method}\ m = e;\ r) &= \{m\} \cup dom_m(r) \end{aligned}$$

(c) Für $a \in dom_a(r)$ ist der Ausdruck $r(a)$ wie folgt induktiv definiert.

$$\begin{aligned}(\textbf{val } a' = e;\, r)(a) &= \begin{cases} e & \text{falls } a' = a \\ r(a) & \text{sonst} \end{cases} \\ (\textbf{method } m = e;\, r)(a) &= r(a)\end{aligned}$$

(d) **TODO:** $r_1 \oplus r_2$ definieren (für die Typerhaltung von *self*-Substitution)

Für Reihen treffen wir analog zu Ausdrücken die folgende Vereinbarung (vgl. Konvention 1.1).

Konvention 2.2 Reihen, die sich lediglich in den Namen von gebundenen Variablen unterscheiden, sind in jeder Beziehung austauschbar.

Diese Konvention bezieht sich ausschliesslich auf Variablen und gilt nicht für Attributnamen oder Objektnamen.

Syntaktischer Zucker

$$\textbf{method } m\ x_1 \ldots x_n = e;\ r \quad \text{für} \quad \textbf{method } m = \lambda x_1. \ldots \lambda x_n.e;\ r$$

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

TODO: Prosa.
TODO: Einleitende Worte mit Bezug auf $\mathcal{L}_f$
Zunächst wollen wir präzisieren, was wir in der operationellen Semantik der Sprache $\mathcal{L}_o$ unter einem Wert verstehen (**TODO:** das geht auch besser).

Definition 2.4 (Werte und Reihenwerte)

(a) Die Menge $RVal \subseteq Row$ aller *Reihenwerte* ω von $\mathcal{L}_o$ ist durch die folgende kontextfreie Grammatik definiert.

$$\begin{aligned}\omega \;::=\; & \epsilon \\ \mid\; & \textbf{val } a = v;\, \omega \\ \mid\; & \textbf{method } m = e;\, \omega\end{aligned}$$

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

$$\begin{array}{rcll} v & ::= & \mathit{id} & \text{Name} \\ & | & \textbf{object}\,(\mathit{self})\;\omega\;\textbf{end} & \text{Objektwert} \end{array}$$

zur kontextfreien Grammatik von $\mathcal{L}_f$ (vgl. Definition 1.12) hinzunimmt.

TODO: Prosa
Zur Definition der operationellen Semantik der Sprache $\mathcal{L}_o$ benötigen wir noch den Hilfsausdruck

$$e \;::=\; \omega\#m$$

in der abstrakten Syntax, der als Zwischenschritt der Berechnung auftritt, dem Programmierer in der konkreten Syntax aber nicht zur Verfügung steht.

2.2.1 Frei vorkommende Namen und Substitution

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 Verallgemeinerung von Definition 1.8.

$$\begin{array}{rcl} \mathit{free}\,(c) & = & \emptyset \\ \mathit{free}\,(\mathit{id}) & = & \{\mathit{id}\} \\ \mathit{free}\,(e\#m) & = & \mathit{free}\,(e) \\ \mathit{free}\,(\omega\#m) & = & \mathit{free}\,(\omega) \\ \mathit{free}\,(\textbf{object}\,(\mathit{self})\;r\;\textbf{end}) & = & \mathit{free}\,(r) \setminus \{\mathit{self}\} \\ \mathit{free}\,(\{\langle a_i = e_i{}^{i=1\ldots n}\rangle\}) & = & \{\mathit{self}\} \cup \bigcup_{i=1}^{n} (\mathit{free}\,(e_i) \cup \{a_i\}) \end{array}$$

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

$$\begin{array}{rcl} \mathit{free}\,(\epsilon) & = & \emptyset \\ \mathit{free}\,(\textbf{val}\;a = e;\; r) & = & \mathit{free}\,(e) \cup (\mathit{free}\,(r) \setminus \{a\}) \\ \mathit{free}\,(\textbf{method}\;m = e;\; r) & = & \mathit{free}\,(e) \cup \mathit{free}\,(r)\,\{\mathit{self}\} \end{array}$$

TODO: Prosa, zum Beispiel, dass Methodennamen nicht in $\mathit{free}\,(e)$ auftauchen. Weiterhin erklären, warum *self* frei in Duplikationen und Methoden ist (mit Verweis auf das Typsystem, insb. (Dupl) und (Method) Regeln).

Definition 2.6 Die Menge $Exp^\star \subseteq Exp$ aller Ausdrücken, die keine freien Vorkommen von Attribut- und Objektnamen enthalten, ist wie folgt definiert.

$$Exp^\star \;=\; \left\{ e \in Exp \;\middle|\; free\,(e) \subseteq Var \right\}$$

TODO: Überleitung

Definition 2.7 (Reiheneinsetzung) Für $r \in Row$, $e \in Exp^\star$ und $a \in dom_a\,(r)$ ist die Reihe $r\langle {}^{e}/_{a} \rangle$, die durch *Reiheneinsetzung* aus r entsteht, indem die rechte Seite der Deklaration des Attributs a durch den Ausdruck e ersetzt wird, wie folgt definiert.

$$\begin{aligned}
(\mathbf{val}\ a' = e';\ r)\langle {}^{e}/_{a} \rangle &= \begin{cases} \mathbf{val}\ a' = e;\ r & \text{falls } a' = a \\ \mathbf{val}\ a' = e';\ r\langle {}^{e}/_{a} \rangle & \text{sonst} \end{cases} \\
(\mathbf{method}\ m = e';\ r)\langle {}^{e}/_{a} \rangle &= \mathbf{method}\ m = e';\ r\langle {}^{e}/_{a} \rangle
\end{aligned}$$

TODO: Überleitung

Definition 2.8 (Substitution)

(a) Für $e' \in Exp$, $e \in Exp^\star$ und $id \in Id$ erhält man den Ausdruck $e'[{}^{e}/_{id}]$, der aus e' durch *Substitution* von e für id entsteht, durch folgende Verallgemeinerung von Definition 1.10.

$$\begin{aligned}
id'[{}^{e}/_{id}] &= \begin{cases} e & \text{falls } id = id' \\ id' & \text{sonst} \end{cases} \\
e'\#m[{}^{e}/_{id}] &= (e'[{}^{e}/_{id}])\#m \\
\omega\#m[{}^{e}/_{id}] &= (\omega[{}^{e}/_{id}])\#m \\
\mathbf{object}\,(self)\ r\ \mathbf{end}[{}^{e}/_{id}] &= \begin{cases} \mathbf{object}\,(self)\ r\ \mathbf{end} & \text{falls } id = self \\ \mathbf{object}\,(self)\ r[{}^{e}/_{id}]\ \mathbf{end} & \text{sonst} \end{cases} \\
\{\langle a_i = e_i{}^{i=1\ldots n} \rangle\}[{}^{e}/_{id}] &= \begin{cases} \mathbf{let}\ \vec{x} = \vec{e}[{}^{e}/_{id}]\ \mathbf{in}\ \mathbf{object}\,(self)\ r\langle {}^{x_i}/_{a_i} \rangle^{i=1\ldots n}\ \mathbf{end} \\ \quad \text{falls } id = self \wedge e = \mathbf{object}\,(self)\ r\ \mathbf{end} \\ \{\langle a_i = e_i[{}^{e}/_{id}]^{i=1\ldots n} \rangle\} \\ \quad \text{sonst} \end{cases} \\
&\quad \text{mit } x_1, \ldots, x_n \notin free\,(r) \cup \textstyle\bigcup_{i=1}^{n} free\,(e)
\end{aligned}$$

(b) Für $r \in Row$, $e \in Exp^\star$ und $id \in Id$ ist die Reihe $r[{}^{e}/_{id}]$, die aus r durch *Substitution* von e für id entsteht, wie folgt induktiv definiert.

$$\begin{aligned}
\epsilon[{}^{e}/_{id}] &= \epsilon \\
(\mathbf{val}\ a = e';\ r)[{}^{e}/_{id}] &= \begin{cases} \mathbf{val}\ a = e'[{}^{e}/_{id}];\ r & \text{falls } id = a \\ \mathbf{val}\ a = e'[{}^{e}/_{id}];\ r[{}^{e}/_{id}] & \text{sonst} \end{cases} \\
(\mathbf{method}\ m = e';\ r)[{}^{e}/_{id}] &= \mathbf{method}\ m = e'[{}^{e}/_{id}];\ r[{}^{e}/_{id}]
\end{aligned}$$

Es ist wieder zu beachten, dass gemäß Konvention 1.1 und Konvention 2.2 die Substitution total ist, wenn der substituierte Ausdruck keine freien Vorkommen von Attribut- oder Objektnamen enthält. Insbesondere gilt aber zu beachten, dass die Substitution für Ausdrücke $e \in \mathit{Exp} \setminus \mathit{Exp}^\star$ undefiniert ist.

Hiermit lässt sich nun unmittelbar das folgende Lemma formulieren, welches uns zusichert, dass das Ergebnis der Anwendung einer Substitution auf einen Ausdruck oder eine Reihe stets eindeutig definiert ist durch den Ausdruck oder die Reihe, den Namen, der ersetzt werden soll, und den Ausdruck, der für den Namen eingesetzt wird.

Lemma 2.1

(a) Für $e' \in \mathit{Exp}$, $e \in \mathit{Exp}^\star$ und $id \in \mathit{Id}$ ist $e'[^e/_{id}]$ eindeutig bestimmt.

(b) Für $r \in \mathit{Row}$, $e \in \mathit{Exp}^\star$ und $id \in \mathit{Id}$ ist $r[^e/_{id}]$ eindeutig bestimmt.

Beweis: Folgt wegen Konvention 1.1 und Konvention 2.2 unmittelbar aus Definition 2.8. □

TODO: Noch ein Satz zum Lemma, vielleicht mit Beispiel, dass es nicht auf syntaktische Gleichheit ankommt.

Das nächste Lemma beschreibt den Zusammenhang zwischen Substitution und frei vorkommenden Namen.

Lemma 2.2

(a) $\forall e' \in \mathit{Exp}, e \in \mathit{Exp}^\star, id \in \mathit{Id} : id \notin \mathit{free}(e') \Rightarrow e' = e'[^e/_{id}]$

(b) $\forall r \in \mathit{Row}, e \in \mathit{Exp}^\star, id \in \mathit{Id} : id \notin \mathit{free}(r) \Rightarrow r = r[^e/_{id}]$

Beweis: Wir führen den Beweis durch simultane Induktion über die Grösse von e' und r und Fallunterscheidung nach der syntaktischen Form von e' und r. Dazu betrachten wir exemplarisch die folgenden Fälle.

1.) Für Konstanten c gilt bereits wegen Definition 2.8 $c[^e/_{id}] = c$.

2.) Für Namen id' gilt $id' \neq id$ wegen $id \notin \mathit{free}(id') = \{id'\}$, also insbesondere $id'[^e/_{id}] = id'$ wegen Definition 2.8.

3.) Für Applikationen $e_1\, e_2$ gilt $id \notin \mathit{free}(e_1)$ und $id \notin \mathit{free}(e_2)$. Wegen $(e_1\, e_2)[^e/_{id}] = e_1[^e/_{id}]\, e_2[^e/_{id}]$ folgt nach Induktionsvoraussetzung $e_1 = e_1[^e/_{id}]$ und $e_2 = e_2[^e/_{id}]$. Nach Definition 2.8 also $(e_1\, e_2)[^e/_{id}] = e_1\, e_2$.

4.) Bei Objekten **object** (*self*) r **end** sind zwei Fälle zu unterscheiden.

Für $id = self$ folgt (**object** (*self*) r **end**)$[^e/_{id}]$ = **object** (*self*) r **end** bereits aus Definition 2.8.

Für $id \neq self$ ist (**object** (*self*) r **end**)$[^e/_{id}]$ = **object** (*self*) $r[^e/_{id}]$ **end** und $id \notin free\,(r) \subseteq free\,(\textbf{object}\,(self)\ r\ \textbf{end}) \cup \{self\}$. Es gilt $r = r[^e/_{id}]$ nach Induktionsvoraussetzung, also (**object** (*self*) r **end**)$[^e/_{id}]$ = **object** (*self*) r **end**.

5.) Für Duplikationen $\{\langle a_i = e_i{}^{i=1\ldots n}\rangle\}$ gilt $id \neq self$ nach Definition 2.5. Folglich ist $\{\langle a_i = e_i{}^{i=1\ldots n}\rangle\}[^e/_{id}] = \{\langle a_i = e_i[^e/_{id}]^{i=1\ldots n}\rangle\}$ und nach Induktionsvoraussetzung $e_i = e_i[^e/_{id}]$ für alle $i = 1 \ldots n$. Somit gilt $\{\langle a_i = e_i{}^{i=1\ldots n}\rangle\}[^e/_{id}] = \{\langle a_i = e_i{}^{i=1\ldots n}\rangle\}$.

6.) Bei Attributdeklarationen **val** $a = e'$; r sind wieder zwei Fälle zu unterscheiden.

Für $a = id$ ist (**val** $a = e'$; $r)[^e/_{id}]$ = **val** $a = e'[^e/_{id}]$; r und nach Definition 2.5 $id \notin free\,(e')$. Also folgt mit Induktionsvoraussetzung $e' = e'[^e/_{id}]$ und somit (**val** $a = e'$; $r)[^e/_{id}]$ = **val** $a = e'$; r.

Für $a \neq id$ ist (**val** $a = e'$; $r)[^e/_{id}]$ = **val** $a = e'[^e/_{id}]$; $r[^e/_{id}]$ und analog zum vorhergehenden Fall gilt $id \notin free\,(e') \cup free\,(r) \cup \{a\}$. Nach Induktionsvoraussetzung folgt $e' = e'[^e/_{id}]$ und $r = r[^e/_{id}]$, also (**val** $a = e'$; $r)[^e/_{id}]$ = **val** $a = e'$; r.

Die übrigen Fälle verlaufen analog. □

Korollar 2.1

(a) $\forall e' \in Exp, e \in Exp^\star, id \in Id : free\,(e') = \emptyset \Rightarrow e' = e'[^e/_{id}]$

(b) $\forall r \in Row, e \in Exp^\star, id \in Id : free\,(r) = \emptyset \Rightarrow r = r[^e/_{id}]$

Beweis: Folgt wegen $id \notin \emptyset$ unmittelbar aus Lemma 2.2. □

TODO: Prosa

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 $e, e' \in Exp$ oder $r \rightarrow_r r'$ mit $r, r' \in Row$.

TODO: Ein bisschen Erläuterung

Definition 2.10 (Gültige small steps für $\mathcal{L}_o$) Ein small step, $e \to_e e'$ mit $e, e' \in \mathit{Exp}$ oder $r \to_r r'$ 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 1.14), sowie den folgenden small step Regeln für Objekte

$$\text{(Object-Eval)} \quad \frac{r \to_r r'}{\mathbf{object}\,(\mathit{self})\ r\ \mathbf{end} \to_e \mathbf{object}\,(\mathit{self})\ r'\ \mathbf{end}}$$

$$\text{(Send-Eval)} \quad \frac{e \to_e e'}{e\#m \to_e e'\#m}$$

$$\text{(Send-Unfold)} \quad \mathbf{object}\,(\mathit{self})\ \omega\ \mathbf{end}\#m \to_e \omega[{}^{\mathbf{object}\,(\mathit{self})\ \omega\ \mathbf{end}}/_{\mathit{self}}]\#m$$

$$\text{(Send-Attr)} \quad (\mathbf{val}\ a = v;\ \omega)\#m \to_e \omega[{}^{v}/_{a}]\#m$$

$$\text{(Send-Skip)} \quad (\mathbf{method}\ m' = e;\ \omega)\#m \to_e \omega\#m \quad \text{falls } m \neq m' \vee m \in \mathit{dom}_m(\omega)$$

$$\text{(Send-Exec)} \quad (\mathbf{method}\ m' = e;\ \omega)\#m \to_e e \quad \text{falls } m = m' \wedge m \notin \mathit{dom}_m(\omega)$$

und den small step Regeln für Reihen

$$\text{(Attr-Left)} \quad \frac{e \to_e e'}{\mathbf{val}\ a = e;\ r \to_r \mathbf{val}\ a = e';\ r}$$

$$\text{(Attr-Right)} \quad \frac{r \to_r r'}{\mathbf{val}\ a = v;\ r \to_r \mathbf{val}\ a = v;\ r'}$$

$$\text{(Method-Right)} \quad \frac{r \to_r r'}{\mathbf{method}\ m = e;\ r \to_r \mathbf{method}\ m = e;\ r'}$$

herleiten lässt.

TODO: Erläuterung der Regeln, insbesondere bei (Beta-V), (Unfold), (Let-Exec), (Send-Unfold) und (Send-Attr) zu beachten, dass der small step nur dann existiert, wenn der zu substituierende Ausdruck aus $\mathit{Exp}^\star$ ist, ansonsten bleibt der Interpreter stecken.

Im Folgenden betrachten wir $\to_e$ als eine Relation

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

und $\to_r$ als eine Relation

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

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 Relation $\rightarrow$, d.h. $\xrightarrow{*}$ bezeichnet eine endliche, möglicherweise leere Folge von small steps, und $\xrightarrow{+}$ eine endliche, nicht-leere Folge von small steps.

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

Lemma 2.3

(a) $v \not\rightarrow$ *für alle* $v \in Val$.

(b) $\omega \not\rightarrow$ *für alle* $\omega \in RVal$.

Beweis: Der Beweis ergibt sich durch simultane 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 $\rightarrow$ 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 $v =$ **object** $(self)\ \omega'$ **end** kommt nur ein small step mit (Object-Eval) in Frage. Nach Induktionsvorraussetzung existiert aber kein small step für ω'.

Für $\omega =$ **val** $a = v'$; ω' kommt nur ein small step mit (Attr-Left) oder (Attr-Right) in Frage. Nach Induktionsvoraussetzung existiert aber weder für ω' noch für v' ein small step, folglich ist keine der beiden Regeln anwendbar.

Es bleibt der Fall $\omega =$ **method** $m = e$; ω', 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 $e_1 \rightarrow e_2 \rightarrow \ldots$.

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

Im Falle einer unendlichen Berechnung für einen Ausdrucks, sagt man auch die Berechnung *divergiert*. Im Falle einer endlichen Berechnung, an deren Ende ein Wert steht, sagt man die Berechnung *terminiert*.

Damit kommen wir nun zum wichtigstens Satz über die small step Semantik der Programmiersprache $\mathcal{L}_o$, welcher ein deterministisches Verhalten bei der Auswertung von Ausdrücken und Reihen sichert. Das bedeutet insbesondere, dass für jeden Ausdruck und jede Reihe höchstens eine Berechnung existiert.

Satz 2.1 (Eindeutigkeit des Übergangsschritts) *Für jeden Ausdruck $e \in Exp$ existiert höchstens ein $e' \in Exp$ mit $e \to e'$ und für jede Reihe $r \in Row$ existiert höchstens ein $r' \in Row$ mit $r \to r'$.*

Beweis: Wir führen den Beweis durch simultane Induktion über die Grösse von Ausdrücken e und Reihen r, und Fallunterscheidung nach der syntaktischen Form von e und r.

1.) e ist von der Form c, id oder $\lambda x.e'$.

Wegen Lemma 2.3 existiert in diesen Fällen kein small step für e.

2.) e ist von der Form $e_1\, e_2$.

Für Applikationen kommen nur die vier small step Regeln (Op), (Beta-V), (App-Left) und (App-Right) in Frage.

Falls ein small step $e_1 \to e_1'$ existiert kann nur die Regel (App-Left) angewandt werden, denn (Op), (Beta-V) und (App-Right) erfordern einen Wert auf der linken Seite der Applikation. Nach Induktionsvoraussetzung ist e_1' eindeutig durch e_1 bestimmt, also ist auch $e_1'\, e_2$ eindeutig durch $e_1\, e_2$ bestimmt.

Wenn $e_1 \not\to$, aber ein small step $e_2 \to e_2'$ existiert, dann kommt nur (App-Right) in Frage, denn (Op) und (Beta-V) erfordern einen Wert auf der rechten Seite der Applikation. Nach Induktionsvoraussetzung ist e_2' eindeutig bestimmt durch e_2, folglich ist auch $e_1\, e_2'$ eindeutig bestimmt durch $e_1\, e_2$.

Es bleibt noch der Fall $e_1 \not\to$ und $e_2 \not\to$. Hier kommen höchstens (Op) und (Beta-V) in Frage. Diese schliessen sich offensichtlich gegenseitig aus, und führen beide zu einem eindeutigen Ergebnis.

3.) e ist von der Form $\mathbf{rec}\, x.e'$.

Dann kommt ausschliesslich die small step Regel (Unfold) in Frage, deren Ergebnis nach Lemma 2.1 eindeutig ist.

4.) e ist von der Form $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$.

Für **let**-Ausdrücke kommen nur die beiden small step Regeln (LET-EVAL) und (LET-EXEC) in Frage.

Existiert ein small step $e_1 \to e_1'$ kann nur die Regel (LET-EVAL) angewandt werden, da (LET-EXEC) als ersten Teilausdruck einen Wert erfordert. Nach Induktionsvoraussetzung ist e_1' eindeutig durch e_1 bestimmt, also ist auch $\mathbf{let}\, x = e_1'\, \mathbf{in}\, e_2$ eindeutig bestimmt durch $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$.

Gilt andererseits $e_1 \not\to$, dann kann (LET-EVAL) nicht angewandt werden, und (LET-EXEC) nur, falls $e_1 \in \mathit{Val}$. Wenn (LET-EXEC) anwendbar ist, so ist das Ergebnis nach Lemma 2.1 eindeutig.

5.) e ist von der Form $\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2$.

Für bedingte Ausdrücke kommen nur die drei small step Regeln (COND-EVAL), (COND-TRUE) und (COND-FALSE) in Frage.

Wenn ein small step $e_0 \to e_0'$ existiert, kann nur die Regel (COND-EVAL) angewendet werden, da (COND-TRUE) und (COND-FALSE) als ersten Teilausdruck eine Konstante erfordern. Nach Induktionsvoraussetzung ist e_0' eindeutig durch e_0 bestimmt, folglich ist auch $\mathbf{if}\, e_0'\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2$ eindeutig bestimmt durch $\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2$.

Für $e_0 \not\to$ kommen nur die Regeln (COND-TRUE) und (COND-FALSE) in Frage, da (COND-EVAL) erfordert, dass ein small step für e_0 existiert. Diese beiden Regeln schliessen sich aus offensichtlichen Gründen gegenseitig aus, und das Ergebnis ist stets eindeutig.

6.) e ist von der Form $e_1 \# m$.

Dann kommen nur die beiden small step Regeln (SEND-EVAL) und (SEND-UNFOLD) in Frage.

Existiert ein small step $e_1 \to e_1'$, so ist nur (SEND-EVAL) anwendbar, da (SEND-UNFOLD) einen Objektwert auf der linken Seite der Nachricht erfordert. Nach Induktionsvoraussetzung ist e_1' eindeutig bestimmt durch e_1, also ist auch $e_1' \# m$ eindeutig durch $e_1 \# m$ bestimmt.

Gilt andererseits $e_1 \not\to$, dann ist (SEND-EVAL) nicht anwendbar, und (SEND-EXEC) kann nur angewendet werden, wenn e_1 ein Objektwert ist. In diesem Fall ist das Ergebnis wegen Lemma 2.1 eindeutig bestimmt.

7.) e ist von der Form $\omega \# m$.

Dann kommen ausschliesslich die drei small step Regeln (SEND-ATTR), (SEND-SKIP) und (SEND-EXEC) in Frage.

Falls $\omega = \epsilon$ ist, kann keine der Regeln angewandt werden und es ist kein small step möglich.

Falls ω von der Form **val** $a = v$; ω' ist, dann kann nur Regel (SEND-ATTR) angewandt werden, und das Ergebnis ist nach Lemma 2.1 eindeutig.

Wenn r von der Form **method** $m' = e$; ω' ist, dann ist aus offensichtlichen Gründen nur exakt eine der beiden Regeln (SEND-SKIP) und (SEND-EXEC) anwendbar, und das Ergebnis ist mit Induktionsvoraussetzung eindeutig.

8.) r ist von der Form ϵ.

Dann existiert nach Lemma 2.3 kein small step für r.

9.) r ist von der Form **val** $a = e$; r_1.

Für Attributdeklarationen kommen nur die beiden small step Regeln (ATTR-LEFT) und (ATTR-RIGHT) in Frage.

Falls ein small step $e \rightarrow e'$ existiert, dann kann nur (ATTR-LEFT) angewendet werden, da (ATTR-RIGHT) einen Wert auf der linken Seite der Attributdeklaration erfordert. Nach Induktionsvoraussetzung ist e' eindeutig bestimmt durch e, also ist auch **val** $a = e'$; r_1 durch **val** $a = e$; r_1 eindeutig bestimmt.

Falls $e \in \mathit{Val}$ und ein small step $r_1 \rightarrow r_1'$ existiert, kommt nur (ATTR-RIGHT) in Frage. Nach Induktionsvoraussetzung ist r_1' durch r_1 eindeutig bestimmt, also ist auch **val** $a = e$; r_1' eindeutig durch **val** $a = e$; r_1 bestimmt.

Für $e \in \mathit{Val}$ und $r_1 \in \mathit{RVal}$ gilt nach Lemma 2.3 **val** $a = e$; $r_1 \not\rightarrow$.

10.) r ist von der Form **method** $m = e$; r_1.

Hier kommt lediglich die small step Regel (METHOD-RIGHT) in Frage. Falls ein small step $r_1 \rightarrow r_1'$ existiert, ist nach Induktionsvoraussetzung r_1' durch r_1 eindeutig bestimmt, also ist auch **method** $m = e$; r_1' eindeutig durch **method** $m = e$; r_1 bestimmt.

Für $r_1 \in \mathit{RVal}$ gilt nach Lemma 2.3 **method** $m = e$; $r_1 \not\rightarrow$. □

Die bereits angedeutete Eindeutigkeit von Berechnungen läßt sich nun einfach als Korollar des vorangegangenen Satzes formulieren.

Korollar 2.2

(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. □

Nun stellt sich sicherlich die Frage, warum die Eigenschaft der Eindeutigkeit der small step Semantik so entscheidend ist. Neben dem offensichtlichen praktischen Nutzen, nämlich der Existenz eines Interpreters für die Programmiersprache $\mathcal{L}_o$, ist die Eindeutigkeit für die späteren Beweise eine erhebliche Erleichterung[2].

2.2.3 Big step Semantik

Wie bereits in Kapitel 1 angedeutet wurde, existiert neben dem *iterativen Ansatz* zur Definition von Semantiken noch ein *rekursiver Ansatz*, der als *big step Semantik* bezeichnet wird. Hierbei wird das Ergebnis einer Umformung auf die Ergebnisse von Teilumformungen zurückgeführt. In der Praxis wird für Interpreter fast ausschliesslich der rekursive Ansatz benutzt, da dieser eher der intuitiven Interpretation von Programmen entspricht.

In diesem Abschnitt wollen wir eine big step Semantik für die Programmiersprache $\mathcal{L}_o$ entwickeln, die äquivalent zu der im vorangegangenen Abschnitt definierten small step Semantik ist. Dazu legen wir zunächst fest, was wir unter einem big step verstehen.

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

Analog zur small step Semantik geben wir nun ein System von Regeln an, mit denen sich gültige big steps der Form $e \Downarrow_e e'$ und $r \Downarrow_r r'$ herleiten lassen.

Definition 2.13 (Gültige big steps) Ein big step, $e \Downarrow_e v$ mit $e \in \mathit{Exp}, v \in \mathit{Val}$ oder $r \Downarrow_r \omega$ mit $r \in \mathit{Row}, \omega \in \mathit{RVal}$, heisst *gültig*, wenn er sich mit den big step Regeln für die funktionale Sprache $\mathcal{L}_f$

[2]Teilweise ermöglicht sie überhaupt erst den Beweis von Eigenschaften.

(Val) $v \Downarrow_e v$

(Op) $op\, n_1\, n_2 \Downarrow_e op^I(n_1, n_2)$

(Beta-V) $$\frac{e[^{v}/_{x}] \Downarrow_e v'}{(\lambda x.e)\, v \Downarrow_e v'}$$

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

(Unfold) $$\frac{e[^{\mathbf{rec}\, x.e}/_{x}] \Downarrow_e v}{\mathbf{rec}\, x.e \Downarrow_e v}$$

(Let) $$\frac{e_1 \Downarrow_e v_1 \quad e_2[^{v_1}/_{x}] \Downarrow_e v_2}{\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \Downarrow_e v_2}$$

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

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

sowie den big step Regeln für Objekte

(Object) $$\frac{r \Downarrow_r \omega}{\mathbf{object}\, (\mathit{self})\ r\ \mathbf{end} \Downarrow_e \mathbf{object}\, (\mathit{self})\ \omega\ \mathbf{end}}$$

(Send) $$\frac{e \Downarrow_e \mathbf{object}\, (\mathit{self})\ \omega\ \mathbf{end} \quad \omega[^{\mathbf{object}\, (\mathit{self})\ \omega\ \mathbf{end}}/_{\mathit{self}}]\#m \Downarrow_e v}{e\#m \Downarrow_e v}$$

(Send-Attr) $$\frac{\omega[^{v}/_{a}]\#m \Downarrow_e v'}{(\mathbf{val}\ a = v;\ \omega)\#m \Downarrow_e v'}$$

(Send-Skip) $$\frac{m' \neq m \vee m \in \mathit{dom}_m(\omega) \quad \omega\#m \Downarrow_e v}{(\mathbf{method}\ m' = e;\ \omega)\#m \Downarrow_e v}$$

(Send-Exec) $$\frac{m' = m \wedge m \notin \mathit{dom}_m(\omega) \quad e \Downarrow_e v}{(\mathbf{method}\ m' = e;\ \omega)\#m \Downarrow_e v}$$

und den big step Regeln für Reihen

(Omega) $\omega \Downarrow_r \omega$

(Attr) $$\frac{e \Downarrow_e v \quad r \Downarrow_r \omega}{\mathbf{val}\ a = e;\ r \Downarrow_r \mathbf{val}\ a = v;\ \omega}$$

(Method) $$\frac{r \Downarrow_r \omega}{\mathbf{method}\ m = e;\ r \Downarrow_r \mathbf{method}\ m = e;\ \omega}$$

herleiten lässt.

Analog zur small step Semantik betrachten wir $\Downarrow_e \subseteq \mathit{Exp} \times \mathit{Val}$ und $\Downarrow_r \subseteq \mathit{Row} \times \mathit{RVal}$ 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.

Intuitiv wird sofort ein elementarer Zusammenhang zwischen den small step Regeln und den big step Regeln ersichtlich, so dass man vermuten kann, dass die beiden Semantik das gleiche Laufzeitverhalten mit Blick auf das Ergebnis einer Berechnung aufweisen sollten. Der Äquivalenzsatz sichert genau diese Übereinstimmung der beiden Semantiken.

Satz 2.2 (Äquivalenzsatz)

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

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

Der Begriff der Äquivalenz besagt in diesem Zusammenhang lediglich, dass eine terminierende Ausführung eines Programms mit den unterschiedlichen Semantiken zum gleichen Ergebnis führt.

Beweis: Der Beweis erfolgt in zwei Schritten. Zunächst zeigen wir, dass für jeden big step eine äquivalente 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.

„⇒“ Diese Richtung beweisen wir durch simultane Induktion über die Länge der Herleitungen der big steps $e \Downarrow v$ und $r \Downarrow \omega$, und Fallunterscheidung nach der zuletzt angewandten big step Regel.

1.) Im Fall von $v \Downarrow v$ mit big step Regel (Val) gilt $v \xrightarrow{*} v$ wegen der Reflexivität von $\xrightarrow{*}$.

2.) Für $op\, n_1\, n_2 \Downarrow op^I(n_1, n_2)$ mit big step Regel (Op) folgt $op\, n_1\, n_2 \to op^I(n_1, n_2)$ mit small step Regel (Op).

3.) $(\lambda x.e)\, v \Downarrow v'$ mit big step Regel (Beta-V) bedingt $e[^v/_x] \Downarrow v'$. Nach Induktionsvoraussetzung existiert also eine Berechnung $e[^v/_x] \xrightarrow{*} v'$, mit small step Regel (Beta-V) gilt also $(\lambda x.e)\, v \to e[^v/_x] \xrightarrow{*} v'$.

4.) $e_1\, e_2 \Downarrow v$ mit big step Regel (App) kann nur aus Prämissen der Form $e_1 \Downarrow v_1$, $e_2 \Downarrow v_2$ und $v_1\, v_2 \Downarrow v$ folgen. Wegen Induktionsvoraussetzung gilt $e_1 \xrightarrow{*} v_1$, $e_2 \xrightarrow{*} v_2$ und $v_1\, v_2 \xrightarrow{*} v$. Dann existiert eine Berechnungsfolge $e_1\, e_2 \xrightarrow{*} v_1\, e_2$

mit small step Regel (App-Left), und eine Berechnungsfolge $v_1\,e_2 \xrightarrow{*} v_1\,v_2$ mit small step Regel (App-Right). Also existiert insgesamt eine Berechnung $e_1\,e_2 \xrightarrow{*} v_1\,e_2 \xrightarrow{*} v_1\,v_2 \xrightarrow{*} v$.

5.) Für $e\#m \Downarrow v$ mit big step Regel (Send) muss gelten $e \Downarrow \mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}$ und $\omega[^{\mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}}/_{\mathit{self}}]\#m \Downarrow v$, also gilt nach Induktionsvoraussetzung $e \xrightarrow{*} \mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}$ und $\omega[^{\mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}}/_{\mathit{self}}]\#m \xrightarrow{*} v$. Wegen Regel (Send-Unfold) gilt $\mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end} \to \omega[^{\mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}}/_{\mathit{self}}]\#m$, weiter existiert eine Berechnungsfolge $e\#m \xrightarrow{*} \mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}\#m$ mit small step Regel (Send-Eval), also existiert insgesamt eine Berechnung

$$\begin{array}{ll} & e\#m \\ \xrightarrow{*} & \mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}\#m \\ \to & \omega[^{\mathbf{object}\,(\mathit{self})\;\omega\;\mathbf{end}}/_{\mathit{self}}]\#m \\ \xrightarrow{*} & v. \end{array}$$

Die restlichen Fälle sind ebenso einfach zu beweisen.

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

1.) Für $e \xrightarrow{0} v$ gilt $e = v$, also existiert ein big step $e \Downarrow v$ mit Regel (Val), und entsprechend für $r \xrightarrow{0} \omega$ mit big step Regel (Omega).

2.) Im Falle einer Applikation $e = e_1\,e_2$ mit $e_1\,e_2 \xrightarrow{n+1} v$ unterscheiden wir nach der semantischen Form von e_1 und e_2.

1.) Wenn $e_1, e_2 \in \mathit{Val}$, dann kommen für den ersten small step der Herleitung nur (Op) und (Beta-V) in Frage.
Für (Op) folgt die Behauptung trivialerweise mit big step Regel (Op).
Für (Beta-V) muss e_1 eine λ-Abstraktion sein, also $e_1 = \lambda x.e_1'$, und die Berechnungsfolge die Form

$$(\lambda x.e_1')\,e_2 \to e_1'[^{e_2}/_x] \xrightarrow{n} v$$

haben. Nach Induktionsvoraussetzung existiert ein big step $e_1'[^{e_2}/_x] \Downarrow v$ und mit big step Regel (Beta-V) folgt $(\lambda x.e_1')\,e_2 \Downarrow v$.

2.) Es bleibt der Fall zu betrachten, dass e_1 oder e_2 kein Wert ist. Dann existieren $v_1, v_2 \in \mathit{Val}$, so dass die Berechnungsfolge wie folgt aussieht:

$$e_1\,e_2 \xrightarrow{*} v_1\,e_2 \xrightarrow{*} v_1\,v_2 \xrightarrow{*} v$$

Die Berechnungsfolge $e_1\,e_2 \xrightarrow{*} v_1\,e_2$ kann, sofern nicht leer, ausschliesslich mit small step Regel (App-Left) aus $e_1 \xrightarrow{*} v_1$ folgen. Entsprechend kann

$v_1\, e_2 \xrightarrow{*} v_1\, v_2$, sofern nicht leer, nur mit small step Regel (App-Right) aus $e_2 \xrightarrow{*} v_2$ folgen.
Je nachdem, ob $e_1 \in \mathit{Val}$ bzw. $e_2 \in \mathit{Val}$, folgt $e_1 \Downarrow v_1$ bzw. $e_2 \Downarrow v_2$ entweder mit big step Regel (Val) oder nach Induktionsvoraussetzung. $v_1\, v_2 \Downarrow v$ jedoch folgt in jedem Fall mit Induktionsvoraussetzung, da zumindest einer der Ausdrücke noch kein Wert ist, und somit die Berechnungsfolge $v_1\, v_2 \xrightarrow{*} v$ echt kürzer ist als $e_1\, e_2 \xrightarrow{*} v$.
Insgesamt folgt also $e_1\, e_2 \Downarrow v$ mit big step Regel (App).

3.) Für $e = \textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2$ ist die Berechnungsfolge $e \xrightarrow{*} v$ von der Form

$$\textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \xrightarrow{*} \textbf{if}\, v_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \xrightarrow{*} v,$$

wobei sich $\textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \xrightarrow{*} \textbf{if}\, v_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2$ mit small step Regel (Cond-Eval) aus $e_0 \xrightarrow{*} v_0$ ergibt, und die Berechnungsfolge

$$\textbf{if}\, v_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \xrightarrow{*} v$$

entweder mit small step Regel (Cond-True), für $v_0 = \mathit{true}$, oder mit small step Regel (Cond-False), für $v_0 = \mathit{false}$, beginnen muss. Für $e_0 \xrightarrow{*} v_0$ existiert nach Induktionsvoraussetzung ein big step $e_0 \Downarrow v_0$.
Wenn $e_0 = \mathit{true}$, dann hat die restliche Berechnungsfolge die Form

$$\textbf{if}\, \mathit{true}\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \to e_1 \xrightarrow{*} v$$

mit (Cond-True) als erster small step Regel und nach Induktionsvoraussetzung existiert ein big step $e_1 \Downarrow v$. Zusammengefasst folgt also

$$\textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \Downarrow v$$

mit big step Regel (Cond-True).
Entsprechend folgt für $v_0 = \mathit{false}$ die Behauptung wegen big step Regel (Cond-False).

Die weiteren Fälle verlaufen ähnlich. □

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

Korollar 2.3

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

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

Beweis: Folgt mit dem Äquivalenzsatz unmittelbar aus Satz 2.1 über die Eindeutigkeit des Übergangsschritts. □

In Anhang A findet sich eine Implementierung eines Interpreters basierend auf der in diesem Abschnitt entwickelten big step Semantik. Im weiteren Verlauf werden wir allerdings ausschliesslich die für Beweise einfacher zu handhabende small step Semantik betrachten, dieser Abschnitt sollte lediglich zeigen, wie eine aus theoretischen Überlegungen entstandene Programmiersprache auf eine praktische Implementierung übertragen werden kann.

2.3 Ein einfaches Typsystem

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

structual typing statt *opaque typing* (wie z.B. in Java oder C++), da dies in der Theorie traditionell der verbreitetere Ansatz ist. Gewinnt aber auch in der Praxis im Bereich von verteilten Systemen zunehmend an Bedeutung; z.B. bei Java RMI problematisch Objekte zu versenden, beide Seiten müssen über die Klasse (und alle beteiligten Klassen verfügen). Einfacher/flexibler wäre „klassenlose“ Objekte zu versenden, und strukturelle Typen zu verwenden.

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

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

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

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

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

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.

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

Für Reihentypen treffen wir die folgende Vereinbarung.

Konvention 2.3 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 $m_1, m_2 \in \mathit{Method}$, $\tau_1, \tau_2 \in \mathit{Type}$ und $\phi \in \mathit{RType}$.

TODO: Überleitung

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 Methodentypen 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 2.1 *Die Vereinigung $\phi_1 \oplus \phi_2$ zweier disjunkter Reihentypen $\phi_1, \phi_2 \in RType$ 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.

$$\textbf{method}\ m\ (x_1 : \tau_1) \ldots (x_n : \tau_n) = e;\ r \quad \text{für} \quad \textbf{method}\ m = \lambda x_1 : \tau_1. \ldots \lambda x_n : \tau_n . e;\ r$$

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 $c \in Const, \tau \in Type$. 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 $\tau \in Type$ hat.

Definition 2.18 (Typumgebungen)

(a) Eine *Typumgebung* ist eine partielle Funktion $\Gamma : Id \rightharpoonup Type$ mit endlichem Definitionsbereich.

(b) Sei Γ eine Typumgebung, dann bezeichnet $\Gamma^\star$ die Typumgebung mit den folgenden Eigenschaften.

- $dom\,(\Gamma^\star) = dom\,(\Gamma) \cap Var$
- $\forall id \in dom\,(\Gamma^\star) : \Gamma^\star(id) = \Gamma(id)$

(c) Sei Γ eine Typumgebung, dann bezeichnet Γ^+ die Typumgebung mit den folgenden Eigenschaften.

- $dom\,(\Gamma^+) = dom\,(\Gamma) \setminus dom\,(\Gamma^\star)$
- $\forall id \in dom\,(\Gamma^+) : \Gamma^+(id) = \Gamma(id)$

(d) Die Menge $TEnv = \{\Gamma \mid \Gamma \text{ ist eine Typumgebung}\}$ enthält alle Typumgebungen.

TODO: Prosa.

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

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

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

Die Listenschreibweise verdeutlicht, dass eine Typumgebung letztlich nichts anderes ist als eine „Tabelle“, in der die Typen der bereits bekannten Namen eingetragen sind[3]. Das „Nachschlagen“ in der Tabelle Γ können wir durch die Funktionsanwendung zum Ausdruck bringen: Falls $id \in dom\,(\Gamma)$, dann liefert $\Gamma(id)$ den Typ τ, die für den Namen id in der Tabelle Γ eingetragen wurden, sonst ist $\Gamma(id)$ undefiniert.

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 2.19 (Typurteile für Ausdrücke und Reihen)

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

(a) Ein *Typurteil für Ausdrücke* ist eine Formel der Gestalt $\Gamma \rhd_e e :: \tau$ mit $\Gamma : Id \rightharpoonup Type$, $e \in Exp$ und $\tau \in Type$.

(b) Ein *Typurteil für Reihen* ist eine Formel der Gestalt $\Gamma \rhd_r r :: \phi$ mit $\Gamma : Id \rightharpoonup Type$, $r \in Row$ und $\phi \in RType$.

TODO: Prosa

Definition 2.20 (Gültige Typurteile für $\mathcal{L}_o^t$) Ein Typurteil $\Gamma \rhd_e e :: \tau$ oder $\Gamma \rhd_r r :: \phi$ heisst *gültig* für $\mathcal{L}_o^t$, wenn es sich mit den Typregeln für die funktionale Kernsprache

$$\text{(Id)} \quad \Gamma \rhd_e id :: \tau \quad \text{falls } id \in dom(\Gamma) \wedge \Gamma(id) = \tau$$

$$\text{(Const)} \quad \frac{c :: \tau}{\Gamma \rhd_e c :: \tau}$$

$$\text{(App)} \quad \frac{\Gamma \rhd_e e_1 :: \tau_2 \to \tau \quad \Gamma \rhd e_2 :: \tau_2}{\Gamma \rhd_e e_1\, e_2 :: \tau}$$

$$\text{(Abstr)} \quad \frac{\Gamma[\tau/_x] \rhd_e e :: \tau'}{\Gamma \rhd_e \lambda x : \tau.e :: \tau \to \tau'}$$

$$\text{(Rec)} \quad \frac{\Gamma[\tau/_x] \rhd_e e :: \tau}{\Gamma \rhd_e \mathbf{rec}\, x : \tau.e :: \tau}$$

$$\text{(Let)} \quad \frac{\Gamma \rhd_e e_1 :: \tau_1 \quad \Gamma[\tau_1/_x] \rhd_e e_2 :: \tau_2}{\Gamma \rhd_e \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 :: \tau_2}$$

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

sowie den Typregeln für Objekte

$$\text{(Send)} \quad \frac{\Gamma \rhd_e e :: \langle m : \tau;\, \phi \rangle}{\Gamma \rhd_e e\#m :: \tau}$$

$$\text{(Send')} \quad \frac{\Gamma \rhd_e \omega :: (m : \tau;\, \phi)}{\Gamma \rhd_e \omega\#m :: \tau}$$

$$\text{(Object)} \quad \frac{\Gamma^\star[\tau/_{self}] \rhd_r r :: \phi \quad \tau = \langle \phi \rangle}{\Gamma \rhd_e \mathbf{object}\, (self : \tau)\, r\, \mathbf{end} :: \tau}$$

$$\text{(Dupl)} \quad \frac{\Gamma \rhd_e self :: \tau \quad \forall i = 1 \ldots n : \Gamma \rhd_e a_i :: \tau_i \wedge \Gamma \rhd_e e_i :: \tau_i}{\Gamma \rhd_e \{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\} :: \tau}$$

und den Typregeln für Reihen

$$\text{(Empty)} \quad \Gamma \rhd_r \epsilon :: \emptyset$$

$$\text{(Attr)} \quad \frac{\Gamma^\star \rhd_e e :: \tau \quad \Gamma[^\tau/_a] \rhd_r r_1 :: \phi}{\Gamma \rhd_r \textbf{val}\ a = e;\ r_1 :: \phi}$$

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

herleiten lässt. Hierbei wird die spezielle Typregel (Send') ausschliesslich für den Beweis der Typsicherheit benötigt.

TODO: Erläuterung des Regelwerks, Beispiel einer Typherleitung

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.

Definition 2.21 (Wohlgetyptheit)

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

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

TODO: Prosa

Satz 2.3 (Typeindeutigkeit)

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

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

Beweis: Der Beweis erfolgt durch simultane Induktion über die Grösse von e und r und Fallunterscheidung nach der Form von e und r. Im einzelnen ist zu zeigen, dass für jeden Ausdruck und jede Reihe höchstens eine Typregel anwendbar ist, was unmittelbar aus der Definition der gültigen Typregeln folgt, da für jede syntaktische Form nur jeweils genau eine Typregel existiert. □

Korollar 2.4 *Sei* Γ *eine Typumgebung.*

(a) Ist $e \in \mathit{Exp}$ *wohlgetypt in* Γ*, dann existiert genau ein* $\tau \in \mathit{Type}$ *mit* $\Gamma \rhd e :: \tau$.

(b) Ist $r \in Row$ wohlgetypt in Γ, dann existiert genau ein $\phi \in RType$ mit $\Gamma \triangleright e :: \phi$.

Beweis: Folgt vermöge Definition 2.21 direkt aus Satz 2.3. □

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

2.3.3 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 Ausdrucks nicht steckenbleiben kann. Das bedeutet, das Typsystem soll möglichst alle Ausdrücke identifizieren, deren Berechnung stecken bleiben würde.

Wichtig zu bemerken ist hierbei, dass durch ein Typsystem, welches ausschliesslich auf syntaktischen Informationen arbeitet[4], keine strikte Trennung zwischen Programmen, deren Berechnung stecken bleibt, und Programmen, die mit einem Wert terminieren oder divergieren, möglich ist. Eine derartige eindeutige Trennung widerspricht der Unentscheidbarkeit des Halteproblems, da die Programmiersprache $\mathcal{L}_o^t$ turing-vollständig ist (vgl. [Spr92], [Wag94, S.93ff] und [Weg93, S.21ff]).

Man muss sich also damit abfinden, dass ein Typsystem für eine turing-vollständige Programmiersprache immer auch Programme ablehnen wird, die während der Auswertung nicht stecken bleiben würden. Man versucht deshalb Typsysteme immer weiter zu verbessern, so dass immer weniger korrekte Programme abgelehnt werden. Wichtiger für unsere Betrachtung jedoch ist die Eigenschaft, dass das Typsystem definitiv alle Programme identifiziert, die stecken bleiben würden.

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

$$\begin{array}{rcl} v & ::= & \lambda x : \tau.e \\ & | & \textbf{object}\,(\mathit{self} : \tau)\ \omega\ \textbf{end} \end{array}$$

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.

[4] Dies trifft auf alle in diesem Dokument vorgestellten Typsysteme zu.

$$\text{(Beta-V)} \qquad (\lambda x : \tau.e)\, v \to_e e[{}^{v}/_{x}]$$

$$\text{(Unfold)} \qquad \mathbf{rec}\, x : \tau.e \to_e e[{}^{\mathbf{rec}\, x:\tau.e}/_{x}]$$

$$\text{(Object-Eval)} \qquad \frac{r \to_r r'}{\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} \to_e \mathbf{object}\,(\mathit{self} : \tau)\ r'\ \mathbf{end}}$$

$$\text{(Send-Unfold)} \qquad \mathbf{object}\,(\mathit{self} : \tau)\ \omega\ \mathbf{end}\#m \to_e \omega[{}^{\mathbf{object}\,(\mathit{self}:\tau)\ \omega\ \mathbf{end}}/_{\mathit{self}}]\#m$$

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 2.22 Für jeden Ausdruck $e \in \mathit{Exp}(\mathcal{L}_o^t)$ sei $\mathit{erase}\,(e) \in \mathit{Exp}(\mathcal{L}_o)$ der Ausdruck, der aus e durch Entfernen aller Typen entsteht[5], also

$$\begin{aligned}
\mathit{erase}\,(c) &= c \\
\mathit{erase}\,(\mathit{id}) &= \mathit{id} \\
\mathit{erase}\,(e_1\, e_2) &= \mathit{erase}\,(e_1)\ \mathit{erase}\,(e_2) \\
\mathit{erase}\,(\lambda x : \tau.e) &= \lambda x.\mathit{erase}\,(e) \\
\mathit{erase}\,(\mathbf{rec}\, x : \tau.e) &= \mathbf{rec}\, x.\mathit{erase}\,(e) \\
\mathit{erase}\,(\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2) &= \mathbf{let}\, x = \mathit{erase}\,(e_1)\, \mathbf{in}\, \mathit{erase}\,(e_2) \\
\mathit{erase}\,(\mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) &= \mathbf{if}\, \mathit{erase}\,(e_0)\, \mathbf{then}\, \mathit{erase}\,(e_1)\, \mathbf{else}\, \mathit{erase}\,(e_2) \\
\mathit{erase}\,(e\#m) &= \mathit{erase}\,(e)\#m \\
\mathit{erase}\,(\omega\#m) &= \mathit{erase}\,(\omega)\#m \\
\mathit{erase}\,(\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}) &= \mathbf{object}\,(\mathit{self})\ \mathit{erase}\,(r)\ \mathbf{end} \\
\mathit{erase}\,(\{\langle a_i = {e_i}^{i=1\ldots n}\rangle\}) &= \{\langle a_i = \mathit{erase}\,(e_i)^{i=1\ldots n}\rangle\}
\end{aligned}$$

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

$$\begin{aligned}
\mathit{erase}\,(\epsilon) &= \epsilon \\
\mathit{erase}\,(\mathbf{val}\ a = e;\ r) &= \mathbf{val}\ a = \mathit{erase}\,(e);\ \mathit{erase}\,(r) \\
\mathit{erase}\,(\mathbf{method}\ m = e;\ r) &= \mathbf{method}\ m = \mathit{erase}\,(e);\ \mathit{erase}\,(r).
\end{aligned}$$

Bei der Funktion $\mathit{erase} : \mathit{Exp}(\mathcal{L}_o^t) \to \mathit{Exp}(\mathcal{L}_o)$ handelt es sich also offensichtlich um einen Übersetzer zwischen der Sprache $\mathcal{L}_o^t$ und der Sprache $\mathcal{L}_o$. Bei Übersetzern zwischen

[5] 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.

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 2.4

(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 2.5 $\forall e \in Exp(\mathcal{L}_o^t), v \in Val(\mathcal{L}_o^t) : e \xrightarrow{*} v \Leftrightarrow erase\,(e) \xrightarrow{*} erase\,(v)$

Beweis: Trivial. □

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

Definition 2.23 Seien $\Gamma_1, \Gamma_2 \in TEnv$ und $A \subseteq Id$. Γ_1 und Γ_2 *stimmen überein auf* A, wenn gilt:

- $A \subseteq dom\,(\Gamma_1)$
- $A \subseteq dom\,(\Gamma_2)$
- $\forall id \in A : \Gamma_1(id) = \Gamma_2(id)$

In diesem Fall schreiben wir $\Gamma_1 =_A \Gamma_2$.

TODO: Eigentlich zu trivial für ein Lemma...

Lemma 2.4 *Sei* $\Gamma_1, \Gamma_2 \in TEnv$. *Dann gilt:*

(a) $\forall A \subseteq Id, B \subseteq A : \Gamma_1 =_A \Gamma_2 \Rightarrow \Gamma_1 =_B \Gamma_2$

(b) $\forall A \subseteq Var : \Gamma_1 =_A \Gamma_2 \Rightarrow \Gamma_1^\star =_A \Gamma_2^\star$

(c) $\forall A \subseteq Id, id \in Id, \tau \in Type : \Gamma_1 =_A \Gamma_2 \Rightarrow \Gamma_1[\tau/_{id}] =_{A \cup \{id\}} \Gamma_2[\tau/_{id}]$

Beweis:

(a) Sei $B \subseteq A$. Dann folgt unmittelbar $B \subseteq A \subseteq dom\,(\Gamma_1)$, $B \subseteq A \subseteq dom\,(\Gamma_2)$ und $\forall id \in B \subseteq A : \Gamma_1(id) = \Gamma_2(id)$.

(b) Folgt unmittelbar aus der Definition, da $dom\,(\Gamma^\star) = dom\,(\Gamma) \cap Var$.

(c) Folgt ebenso problemlos aus $A \cup \{id\} \subseteq dom\,(\Gamma_1[^\tau/_{id}])$, $A \cup \{id\} \subseteq dom\,(\Gamma_2[^\tau/_{id}])$ und $\Gamma_1[^\tau/_{id}](id) = \tau = \Gamma_2[^\tau/_{id}](id)$. □

TODO: Überleitung

Lemma 2.5 (Typumgebungen und frei vorkommende Namen)

(a) $\forall \Gamma \in TEnv, e \in Exp, \tau \in Type : \Gamma \triangleright e :: \tau \Rightarrow free\,(e) \subseteq dom\,(\Gamma)$

(b) $\forall \Gamma \in TEnv, r \in Row, \phi \in RType : \Gamma \triangleright r :: \phi \Rightarrow free\,(r) \subseteq dom\,(\Gamma)$

Beweis: Wir führen den Beweis durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi$ und Fallunterscheidung nach der zuletzt angewandten Typregel. Dazu betrachten wir exemplarisch die folgenden Fälle.

1.) $\Gamma \triangleright c :: \tau$ mit Typregel (Const). Dann gilt $free\,(c) = \emptyset \subseteq dom\,(\Gamma)$.

2.) $\Gamma \triangleright id :: \tau$ mit Typregel (Id). Dann gilt $id \in dom\,(\Gamma)$ und $\Gamma(id) = \tau$, also folgt $free\,(id) = \{id\} \subseteq dom\,(\Gamma)$.

3.) $\Gamma \triangleright e_1\, e_2 :: \tau$ mit Typregel (App) kann nur aus Prämissen der Form $\Gamma \triangleright e_1 :: \tau' \to \tau$ und $\Gamma \triangleright e_2 :: \tau'$ folgen. Nach Induktionsvoraussetzung gilt $free\,(e_1) \subseteq dom\,(\Gamma)$ und $free\,(e_2) \subseteq dom\,(\Gamma)$. Wegen Definition 2.5 folgt schliesslich $free\,(e_1\, e_2) \subseteq dom\,(\Gamma)$.

4.) $\Gamma \triangleright$ **object** $(self : \tau)\ r$ **end** $:: \tau$ mit Typregel (Object) kann ausschliesslich aus $\Gamma^\star[^\tau/_{self}] \triangleright r :: \phi$ und $\tau = \langle\phi\rangle$ folgen. Nach Induktionsvoraussetzung gilt $free\,(r) \subseteq dom\,(\Gamma^\star[^\tau/_{self}])$ und wegen $dom\,(\Gamma^\star[^\tau/_{self}]) = dom\,(\Gamma) \cup \{self\}$ folgt $free\,(r) \setminus \{self\} \subseteq dom\,(\Gamma)$. Vermöge Definition 2.5 gilt folglich $free\,($**object** $(self : \tau)\ r$ **end**$) \subseteq dom\,(\Gamma)$.

5.) $\Gamma \triangleright \{\langle a_i = e_i{}^{i=1\ldots n}\rangle\} :: \tau$ mit Typregel (Dupl) kann nur aus $\Gamma \triangleright self :: \tau$ und $\Gamma \triangleright a_i :: \tau_i \wedge \Gamma \triangleright e_i :: \tau_i$ für alle $i = 1, \ldots, n$ folgen. Nach Induktionsvoraussetzung gilt dann einerseits $free\,(self) \subseteq dom\,(\Gamma)$, also insbesondere $self \in dom\,(\Gamma)$, und andererseits $free\,(a_i) \subseteq dom\,(\Gamma)$ und $free\,(e_i) \subseteq dom\,(\Gamma)$ für $i = 1, \ldots, n$. Nach Definition 2.5 folgt somit $free\,(\{\langle a_i = e_i{}^{i=1\ldots n}\rangle\}) \subseteq dom\,(\Gamma)$.

6.) $\Gamma \triangleright \textbf{val}\ a = e;\ r :: \phi$ mit Typregel (Attr) kann nur aus Prämissen der Form $\Gamma^\star \triangleright e :: \tau$ und $\Gamma[^\tau/_a] \triangleright r :: \phi$ folgen. Mit Induktionsvoraussetzung folgt hieraus $\mathit{free}\,(e) \subseteq \mathit{dom}\,(\Gamma^\star)$ und $\mathit{free}\,(r) \subseteq \mathit{dom}\,(\Gamma[^\tau/_a])$. Wegen $\mathit{dom}\,(\Gamma^\star) \subseteq \mathit{dom}\,(\Gamma)$ und $\mathit{dom}\,(\Gamma[^\tau/_a]) = \mathit{dom}\,(\Gamma) \cup \{a\}$ folgt $\mathit{free}\,(e) \cup (\mathit{free}\,(r) \setminus \{a\}) \subseteq \mathit{dom}\,(\Gamma)$, und gemäß Definition 2.5 gilt somit $\mathit{free}\,(\textbf{val}\ a = e;\ r) \subseteq \mathit{dom}\,(\Gamma)$.

Die restlichen Fälle verlaufen analog. □

Insbesondere folgt, dass Ausdrücke, die in einer Typumgebung $\Gamma^\star$ wohlgetypt sind, keine freien Vorkommen von Attribut- oder Objektnamen enthalten können. Diese Folgerung des Lemmas werden wir in den folgenden Beweisen häufiger benutzen, und formulieren sie deshalb als Korollar.

Korollar 2.6 $\forall \Gamma \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma^\star \triangleright e :: \tau \Rightarrow e \in \mathit{Exp}^\star$

Beweis: Folgt unmittelbar aus Lemma 2.5. □

TODO: Überleitung

Lemma 2.6 (Koinzidenzlemma)

(a) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma_1 \triangleright e :: \tau \wedge \Gamma_1 =_{\mathit{free}\,(e)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright e :: \tau$

(b) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma_1 \triangleright r :: \phi \wedge \Gamma_1 =_{\mathit{free}\,(r)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright r :: \phi$

In Worten ausgedrückt besagt das Lemma lediglich, dass Typumgebungen, die auf den frei vorkommenden Namen eines Ausdrucks oder einer Reihe übereinstimmen, bezüglich der Typurteile über diesen Ausdrücke oder Reihen austauschbar sind.

Beweis: Erfolgt durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma_1 \triangleright e :: \tau$ und $\Gamma_1 \triangleright r :: \phi$ und Fallunterscheidung nach der zuletzt angewandten Typregel. Dazu betrachten wir exemplarisch die folgenden Fälle.

1.) $\Gamma_1 \triangleright c :: \tau$ mit Typregel (Const) kann nur aus $c :: \tau$ folgen. Dann gilt aber ebenfalls $\Gamma_2 \triangleright c :: \tau$.

2.) $\Gamma_1 \triangleright \mathit{id} :: \tau$ mit Typregel (Id) kann ausschliesslich aus $\mathit{id} \in \mathit{dom}\,(\Gamma_1)$ und $\Gamma_1(\mathit{id}) = \tau$ folgen. Da $\mathit{free}\,(\mathit{id}) = \{\mathit{id}\}$ gilt nach Definition 2.23 aber ebenfalls $\mathit{id} \in \mathit{dom}\,(\Gamma_2)$ und $\Gamma_2(\mathit{id}) = \Gamma_1(\mathit{id}) = \tau$. Also folgt mit Typregel (Id) $\Gamma_2 \triangleright \mathit{id} :: \tau$.

3.) $\Gamma_1 \triangleright e_1\, e_2 :: \tau$ mit Typregel (App) kann nur aus Prämissen der Form $\Gamma_1 \triangleright e_1 :: \tau' \to \tau$ und $\Gamma_1 \triangleright e_2 :: \tau'$ folgen. Wegen Lemma 2.4 gilt nach Induktionsvoraussetzung $\Gamma_2 \triangleright e_1 :: \tau' \to \tau$ und $\Gamma_2 \triangleright e_2 :: \tau'$, also $\Gamma_2 \triangleright e_1\, e_2 :: \tau$ mit Typregel (App).

4.) $\Gamma_1 \triangleright \textbf{object}\,(\mathit{self} : \tau)\ r\ \textbf{end} :: \tau$ mit Typregel (OBJECT) kann nur aus $\Gamma_1^\star[^\tau/_{\mathit{self}}] \triangleright r :: \phi$ und $\tau = \langle\phi\rangle$ folgen. Wegen Lemma 2.4 folgt mit Induktionsvoraussetzung $\Gamma_2^\star[^\tau/_{\mathit{self}}] \triangleright r :: \phi$, und mit Typregel (OBJECT) folgt daraus $\Gamma_2 \triangleright \textbf{object}\,(\mathit{self} : \tau)\ r\ \textbf{end} :: \tau$.

5.) $\Gamma_1 \triangleright \{\langle a_i = e_i{}^{i=1\ldots n}\rangle\} :: \tau$ mit Typregel (DUPL) kann nur aus $\Gamma_1 \triangleright \mathit{self} :: \tau$ und $\Gamma_1 \triangleright a_i :: \tau_i \wedge \Gamma_1 \triangleright e_i :: \tau_i$ für $i = 1, \ldots, n$ folgen. Mit Lemma 2.4 gilt nach Induktionsvoraussetzung $\Gamma_2 \triangleright \mathit{self} :: \tau$ und $\Gamma_2 \triangleright a_i :: \tau_i \wedge \Gamma_2 \triangleright e_i :: \tau_i$ für alle $i = 1, \ldots, n$. Also folgt mit Typregel (DUPL) $\Gamma_2 \triangleright \{\langle a_i = e_i{}^{i=1\ldots n}\rangle\} :: \tau$.

6.) Für $\Gamma_1 \triangleright \textbf{method}\ m = e; r :: (m : \tau; \emptyset) \oplus \phi$ mit Typregel (METHOD) gilt nach Voraussetzung $\Gamma_1 \triangleright e :: \tau$ und $\Gamma_1 \triangleright r :: \phi$. Wegen Induktionsvoraussetzung folgt somit $\Gamma_2 \triangleright e :: \tau$ und $\Gamma_2 \triangleright r :: \phi$, und mit Typregel (METHOD) schliesslich die Behauptung.

Die übrigen Fälle verlaufen analog. □

Korollar 2.7

(a) $\forall \Gamma \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma^\star \triangleright e :: \tau \Rightarrow \Gamma \triangleright e :: \tau.$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma^\star \triangleright r :: \phi \Rightarrow \Gamma \triangleright r :: \phi.$

Beweis: Folgt mit Lemma 2.5 unmittelbar aus Lemma 2.6. □

TODO: Überleitung

Lemma 2.7 (Typurteile und Substitution) *Sei* $\mathit{id} \in \mathit{Attribute} \cup \mathit{Var}$, $\Gamma \in \mathit{TEnv}$, $\tau \in \mathit{Type}$ *und* $e \in \mathit{Exp}$. *Dann gilt:*

(a) $\forall e' \in \mathit{Exp} : \forall \tau' \in \mathit{Type} : \Gamma[^\tau/_{\mathit{id}}] \triangleright e' :: \tau' \wedge \Gamma^\star \triangleright e :: \tau \Rightarrow \Gamma \triangleright e'[^e/_{\mathit{id}}] :: \tau'$

(b) $\forall r \in \mathit{Row} : \forall \phi \in \mathit{RType} : \Gamma[^\tau/_{\mathit{id}}] \triangleright r :: \phi \wedge \Gamma^\star \triangleright e :: \tau \Rightarrow \Gamma \triangleright r[^e/_{\mathit{id}}] :: \phi$

Beweis: Zunächst ist festzuhalten, dass vermöge Lemma 2.5 die Substitution $e'[^e/_{\mathit{id}}]$ bzw. $r[^e/_{\mathit{id}}]$ stets definiert ist. Damit können wir dann den Beweis durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[^\tau/_{\mathit{id}}] \triangleright e' :: \tau'$ und $\Gamma[^\tau/_{\mathit{id}}] \triangleright r :: \phi$ und Fallunterscheidung nach der zuletzt angewandten Typregel führen. Wir betrachten dazu die folgenden Fälle.

1.) $\Gamma[^\tau/_{\mathit{id}}] \triangleright c :: \tau'$ mit Typregel (CONST).

Nach Voraussetzung gilt $c :: \tau'$ und wegen Definition 2.8 ist $c = c[^e/_{\mathit{id}}]$. Also folgt $\Gamma \triangleright c[^e/_{\mathit{id}}] :: \tau'$ mit Typregel (CONST).

2.) $\Gamma[^{\tau}/_{id}] \triangleright id' :: \tau'$ mit Typregel (Id).

Nach Voraussetzung gilt $id' \in dom(\Gamma)$ und $\Gamma(id') = \tau'$.

Für $id = id'$ gilt nach Definition 2.18 $\tau = \tau'$. Wegen Definition 2.8 ist $id'[^{e}/_{id}] = e$ und aus $free(e) \subseteq dom(\Gamma^{\star}) \subseteq dom(\Gamma)$ folgt mit Lemma 2.6 $\Gamma \triangleright id'[^{e}/_{id}] :: \tau'$.

Für $id \neq id'$ ist wegen Definition 2.8 $id'[^{e}/_{id}] = id'$ und wegen Definition 2.18 folgt $\Gamma \triangleright id'[^{e}/_{id}] :: \tau'$.

3.) $\Gamma[^{\tau}/_{id}] \triangleright e_1\, e_2 :: \tau'$ mit Typregel (App).

Dann gilt nach Voraussetzung $\Gamma[^{\tau}/_{id}] \triangleright e_1 :: \tau'' \to \tau'$ und $\Gamma[^{\tau}/_{id}] \triangleright e_2 :: \tau''$. Mit Induktionsvoraussetzung folgt $\Gamma \triangleright e_1[^{e}/_{id}] :: \tau'' \to \tau'$ und $\Gamma \triangleright e_2[^{e}/_{id}] :: \tau''$, mit Typregel (App) also $\Gamma \triangleright e_1[^{e}/_{id}]\, e_2[^{e}/_{id}] :: \tau'$ und wegen Definition 2.8 folgt schliesslich $\Gamma \triangleright (e_1\, e_2)[^{e}/_{id}] :: \tau'$.

4.) $\Gamma[^{\tau}/_{id}] \triangleright \lambda x : \tau'.e' :: \tau' \to \tau''$ mit Typregel (Abstr).

Dann gilt $\Gamma[^{\tau}/_{id}][^{\tau'}/_{x}] \triangleright e' :: \tau''$ und durch gebundene Umbenennung lässt sich die Voraussetzung $x \notin \{id\} \cup free(e)$ für die Substitution herstellen. Wegen $x \neq id$ folgt $\Gamma[^{\tau'}/_{x}][^{\tau}/_{id}] \triangleright e' :: \tau''$ und nach Induktionsvoraussetzung gilt dann $\Gamma[^{\tau'}/_{x}] \triangleright e'[^{e}/_{id}] :: \tau''$. Mit Typregel (Abstr) folgt daraus $\Gamma \triangleright \lambda x : \tau'.e'[^{e}/_{id}] :: \tau' \to \tau''$ und wegen $id \notin \{id\} \cup free(e)$ mit Definition 2.8 schliesslich $\Gamma \triangleright (\lambda x : \tau'.e')[^{e}/_{id}] :: \tau' \to \tau''$.

5.) $\Gamma[^{\tau}/_{id}] \triangleright \mathbf{val}\ a = e'; r :: \phi$ mit Typregel (Attr).

Nach Voraussetzung gilt $(\Gamma[^{\tau}/_{id}])^{\star} \triangleright e' :: \tau'$ und $\Gamma[^{\tau}/_{id}][^{\tau'}/_{a}] \triangleright r :: \phi$.

Für $id \in Var$ gilt insbesondere $\Gamma^{\star}[^{\tau}/_{id}] \triangleright e' :: \tau'$, nach Induktionsvoraussetzung also $\Gamma^{\star} \triangleright e'[^{e}/_{id}] :: \tau'$. Gilt andererseits $id \notin Var$ folgt daraus $\Gamma^{\star} \triangleright e' :: \tau'$, und da $id \notin dom(\Gamma^{\star})$ muss nach Lemma 2.5 ebenfalls $id \notin free(e')$ gelten, somit also $e' = e'[^{e}/_{id}]$ nach Lemma 2.2.

Für $id = a$ gilt dann $\Gamma[^{\tau'}/_{a}] \triangleright r :: \phi$, also $\Gamma \triangleright \mathbf{val}\ a = e'[^{e}/_{id}]; r :: \phi$ wegen Typregel (Attr), und mit Definition 2.8 folgt schliesslich $\Gamma \triangleright (\mathbf{val}\ a = e'; r)[^{e}/_{id}] :: \phi$.

Für $id \neq a$ andererseits gilt $\Gamma[^{\tau'}/_{a}][^{\tau}/_{id}] \triangleright r :: \phi$ und nach Induktionsvoraussetzung $\Gamma[^{\tau'}/_{a}] \triangleright r[^{e}/_{id}] :: \phi$. Mit Typregel (Attr) somit $\Gamma \triangleright \mathbf{val}\ a = e'[^{e}/_{id}]; r[^{e}/_{id}] :: \phi$ und wegen Definition 2.8 gilt dann $\Gamma \triangleright (\mathbf{val}\ a = e'; r)[^{e}/_{id}] :: \phi$.

Die übrigen Fälle verlaufen analog. □

TODO: Überleitung, das folgende Lemma braucht man zum Beweis der Typerhaltung von *self*-Substitution

Lemma 2.8 (Typurteile und Reiheneinsetzung) *Sei* $\Gamma \in TEnv$, $r \in Row$, $\phi \in RType$, $a \in Attribute$, $\tau \in Type$ *und* $e \in Exp$. *Dann gilt:*

$$\Gamma \triangleright r :: \phi \wedge \Gamma^\star \triangleright r(a) :: \tau \wedge \Gamma^\star \triangleright e :: \tau \;\Rightarrow\; \Gamma \triangleright r\langle {}^{e}/_{a}\rangle :: \phi$$

Beweis: Wegen $\Gamma^\star \triangleright e :: \tau$ ist nach Lemma 2.5 $e \in Exp^\star$, und wegen $\Gamma^\star \triangleright r(a) :: \tau$ ist $a \in dom_a(r)$. Also ist die Reiheneinsetzung $r\langle {}^{e}/_{a}\rangle$ gemäss den Voraussetzungen stets definiert. Damit können wir dann den Beweis mittels Induktion über die Grösse von r und Fallunterscheidung nach der syntaktischen Form von r führen.

1.) Für die leere Reihe ϵ ist $a \notin dom_a(\epsilon) = \emptyset$ und somit kann $\Gamma^\star \triangleright \epsilon(a) :: \tau$ nicht gelten.

2.) $\Gamma \triangleright \mathbf{val}\ a' = e';\ r' :: \phi$ kann nur mit Typregel (Attr) aus $\Gamma^\star \triangleright e' :: \tau'$ und $\Gamma[{}^{\tau'}/_{a'}] \triangleright r' :: \phi$ folgen. Es sind zwei Fälle zu unterscheiden.

Für $a = a'$ gilt $(\mathbf{val}\ a' = e';\ r')(a) = e'$ und somit $\tau = \tau'$. Mit Typregel (Attr) folgt daraus $\Gamma \triangleright \mathbf{val}\ a' = e;\ r' :: \phi$ und gemäss Definition 2.7 also $\Gamma \triangleright (\mathbf{val}\ a' = e';\ r')\langle {}^{e}/_{a}\rangle :: \phi$.

Für $a \neq a'$ ist $(\mathbf{val}\ a' = e';\ r')\langle {}^{e}/_{a}\rangle = \mathbf{val}\ a' = e';\ r'\langle {}^{e}/_{a}\rangle$ und $(\mathbf{val}\ a' = e';\ r')(a) = r'(a)$. Wegen $(\Gamma[{}^{\tau'}/_{a'}])^\star = \Gamma^\star$ folgt nach Induktionsvoraussetzung $\Gamma[{}^{\tau'}/_{a'}] :: r'\langle {}^{e}/_{a}\rangle\phi$, mit Typregel (Attr) also $\Gamma \triangleright \mathbf{val}\ a' = e';\ r'\langle {}^{e}/_{a}\rangle :: \phi$ und mit Definition 2.7 schliesslich $\Gamma \triangleright (\mathbf{val}\ a' = e';\ r')\langle {}^{e}/_{a}\rangle :: \phi$.

3.) $\Gamma \triangleright \mathbf{method}\ m = e';\ r' :: (m : \tau';\ \emptyset) \oplus \phi$ kann nur mit Typregel (Method) aus Prämissen der Form $\Gamma \triangleright e' :: \tau'$ und $\Gamma \triangleright r' :: \phi$ folgen. Nach Definition 2.7 ist $(\mathbf{method}\ m = e';\ r')\langle {}^{e}/_{a}\rangle = \mathbf{method}\ m = e';\ r'\langle {}^{e}/_{a}\rangle$ und mit Induktionsvoraussetzung folgt $\Gamma \triangleright r'\langle {}^{e}/_{a}\rangle :: \phi$. Wegen Typregel (Method) gilt also $\Gamma \triangleright \mathbf{method}\ m = e';\ r'\langle {}^{e}/_{a}\rangle :: (m : \tau';\ \emptyset) \oplus \phi$ und somit gemäß Definition 2.7 $\Gamma \triangleright (\mathbf{method}\ m = e';\ r')\langle {}^{e}/_{a}\rangle :: (m : \tau';\ \emptyset) \oplus \phi$. □

Weiterhin brauchen wir für den Beweis des Preservation-Theorems noch das nachfolgende Lemma, welches die Typerhaltung für die *self*-Substitution sichert, da dieser Fall noch nicht durch das allgemeine Lemma über Typurteile und Substitution (2.7) abgedeckt wird.

Lemma 2.9 (Typurteile und *self*-Substitution) *Sei* $\Gamma \in TEnv$, $self \in Self$, $\tau \in Type$ *und* $r \in Row$. *Dann gilt:*

(a) Für alle $e \in Exp$ *und* $\tau' \in Type$*:*

(1) $\Gamma[{}^{\tau}/_{self}] \triangleright e :: \tau'$

(2) $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$

(3) $\forall a \in \mathit{dom}\,(\Gamma) \cap \mathit{Attribute}, \tau_a \in \mathit{Type} : \Gamma^\star \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$

$\Rightarrow \Gamma \triangleright e[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$

(b) Für alle $r' \in \mathit{Row}$ *und* $\phi \in \mathit{RType}$*:*

(1) $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright r' :: \phi$

(2) $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r \oplus r'\ \mathbf{end} :: \tau$

(3) $\forall a \in \mathit{dom}\,(\Gamma) \cap \mathit{Attribute}, \tau_a \in \mathit{Type} : \Gamma^\star \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$

$\Rightarrow \Gamma \triangleright r'[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\oplus r'\ \mathbf{end}}/_{\mathit{self}}] :: \phi$

Beweis: Wegen Lemma 2.5 gilt, dass die Substitution stets definiert ist. Den Beweis führen wir dann durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright e :: \tau'$ und $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright r' :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel. Dazu betrachten wir exemplarisch die folgenden Fälle.

1.) $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright c :: \tau'$ mit (Const).

Wegen $\mathit{free}\,(c) = \emptyset$ gilt nach Lemma 2.6 $\Gamma \triangleright c :: \tau'$ und wegen Definition 2.8 ist $c = c[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}]$, also gilt $\Gamma \triangleright c[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$.

2.) $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright \mathit{id} :: \tau'$ mit (Id).

Nach Voraussetzung gilt $\mathit{id} \in \mathit{dom}\,(\Gamma[^{\tau}/_{\mathit{self}}])$ und $\Gamma[^{\tau}/_{\mathit{self}}](\mathit{id}) = \tau'$. Dann sind zwei Fälle zu unterscheiden.

Für $\mathit{id} = \mathit{self}$ ist $\mathit{id}[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] = \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}$ und $\tau = \tau'$. Also gilt $\Gamma^\star \triangleright \mathit{id}[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$ und wegen Lemma 2.6 ebenfalls $\Gamma \triangleright \mathit{id}[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$.

Für $\mathit{id} \neq \mathit{self}$ ist $\mathit{id}[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] = \mathit{id}$ und insbesondere $\mathit{self} \notin \mathit{free}\,(\mathit{id})$. Es folgt unmittelbar $\Gamma \triangleright \mathit{id}[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$.

3.) $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright e_1\, e_2 :: \tau'$ mit (App).

Nach Voraussetzung gilt $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright e_1 :: \tau'' \to \tau'$ und $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright e_2 :: \tau''$, und mit Induktionsvoraussetzung folgt $\Gamma \triangleright e_1[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'' \to \tau'$ und $\Gamma \triangleright e_2[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau''$. Mit Typregel (App) und Definition 2.8 folgt daraus $\Gamma \triangleright (e_1\, e_2)[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$.

4.) $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright \{\langle a_i = e_i{}^{i=1\ldots n}\rangle\} :: \tau'$ mit (Dupl).

Wir schreiben $\tilde{o}$ für $\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}$. Nach Voraussetzung gilt dann $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright \mathit{self} :: \tau'$, also $\tau = \tau'$, sowie $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright a_i :: \tau_i$ und $\Gamma[^{\tau}/_{\mathit{self}}] \triangleright e_i :: \tau_i$

für $i = 1, \ldots, n$. $\Gamma^\star \rhd \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ andererseits kann nur mit Typregel (OBJECT) aus Prämissen der Form $\Gamma^\star[^\tau/_{\mathit{self}}] \rhd r :: \phi$ und $\tau = \langle\phi\rangle$ folgen.

Dann gilt $\Gamma \rhd e_i[^{\tilde{o}}/_{\mathit{self}}] :: \tau_i$ nach Induktionsvoraussetzung für alle $i = 1, \ldots, n$.

Seien nun $x_1, \ldots, x_n \notin \mathit{free}\,(r) \cup \bigcup_{i=1}^n \mathit{free}\,(e_i)$. Dann folgt mit n-facher Anwendung von Lemma 2.8 aus $\Gamma^\star[^\tau/_{\mathit{self}}][^{\vec{\tau}}/_{\vec{x}}] \rhd x_i :: \tau_i$ und $\Gamma^\star[^{\vec{\tau}}/_{\vec{x}}] \rhd a_i :: \tau_i$ für alle $i = 1, \ldots, n$, sowie $\Gamma^\star[^\tau/_{\mathit{self}}][^{\vec{\tau}}/_{\vec{x}}] \rhd r :: \phi$, dass $\Gamma^\star[^\tau/_{\mathit{self}}][^{\vec{\tau}}/_{\vec{x}}] \rhd r\langle{}^{x_i}/_{a_i}\rangle^{i=1\ldots n} :: \phi$ gilt.

Wegen $\mathit{Self} \cap \mathit{Attribute} = \emptyset$, $\Gamma^\star[^{\vec{\tau}}/_{\vec{x}}] = (\Gamma[^{\vec{\tau}}/_{\vec{x}}])^\star$ und $\tau = \langle\phi\rangle$ folgt dann mit Typregel (OBJECT) $\Gamma[^{\vec{\tau}}/_{\vec{x}}] \rhd \mathbf{object}\,(\mathit{self} : \tau)\ r\langle{}^{x_i}/_{a_i}\rangle^{i=1\ldots n}\ \mathbf{end} :: \tau$, also mit Typregel (LET) $\Gamma \rhd \mathbf{let}\,\vec{x} = \vec{e}[^{\tilde{o}}/_{\mathit{self}}]\,\mathbf{in}\,\mathbf{object}\,(\mathit{self} : \tau)\ r\langle{}^{x_i}/_{a_i}\rangle^{i=1\ldots n}\ \mathbf{end} :: \tau$.

5.) $\Gamma[^\tau/_{\mathit{self}}] \rhd \mathbf{val}\ a = e;\ r' :: \phi$ mit (ATTR).

Wir schreiben $\tilde{o}$ für $\mathbf{object}\,(\mathit{self} : \tau)\ r \oplus \mathbf{val}\ a = e;\ r'\ \mathbf{end}$. Nach Voraussetzung gilt $\Gamma^\star \rhd e :: \tau'$ und $\Gamma[^\tau/_{\mathit{self}}][^{\tau'}/_a] \rhd r' :: \phi$, da $(\Gamma[^\tau/_{\mathit{self}}])^\star = \Gamma^\star$.

Wegen $\mathit{self} \notin \mathit{dom}\,(\Gamma^\star)$ folgt nach Lemma 2.5 $\mathit{self} \notin \mathit{free}\,(e)$ und mit Lemma 2.2 also $e = e[^{\tilde{o}}/_{\mathit{self}}]$.

Desweiteren folgt $(r \oplus \mathbf{val}\ a = e;\ r')(a) = e$ wegen $a \notin \mathit{dom}_a\,(r) \cup \mathit{dom}_a\,(r')$, also $\Gamma \rhd (r \oplus \mathbf{val}\ a = e;\ r')(a) :: \tau'$ und wegen $\mathit{Self} \cap \mathit{Attribute} = \emptyset$ gilt $\Gamma[^{\tau'}/_a][^\tau/_{\mathit{self}}] \rhd r' :: \phi$. Nach Induktionsvoraussetzung gilt dann $\Gamma[^{\tau'}/_a] \rhd r'[^{\tilde{o}}/_{\mathit{self}}] :: \phi$.

Mit Typregel (ATTR) folgt daraus $\Gamma \rhd \mathbf{val}\ a = e[^{\tilde{o}}/_{\mathit{self}}];\ r'[^{\tilde{o}}/_{\mathit{self}}] :: \phi$, also nach Definition 2.8 $\Gamma \rhd (\mathbf{val}\ a = e;\ r')[^{\tilde{o}}/_{\mathit{self}}] :: \phi$.

6.) $\Gamma[^\tau/_{\mathit{self}}] \rhd \mathbf{method}\ m = e;\ r' :: \phi \oplus (m : \tau';\ \emptyset)$ mit (METHOD).

Wir schreiben $\tilde{o}$ für $\mathbf{object}\,(\mathit{self} : \tau)\ r \oplus \mathbf{method}\ m = e;\ r'\ \mathbf{end}$. Nach Voraussetzung gilt $\Gamma[^\tau/_{\mathit{self}}] \rhd e :: \tau'$ und $\Gamma[^\tau/_{\mathit{self}}] \rhd r' :: \phi$. Mit Induktionsvoraussetzung folgt daraus $\Gamma \rhd e[^{\tilde{o}}/_{\mathit{self}}] :: \tau'$ und $\Gamma \rhd r'[^{\tilde{o}}/_{\mathit{self}}] :: \phi$. Wegen Typregel (METHOD) gilt also $\Gamma \rhd \mathbf{method}\ m = e[^{\tilde{o}}/_{\mathit{self}}];\ r'[^{\tilde{o}}/_{\mathit{self}}] :: \phi \oplus (m : \tau';\ \emptyset)$, und nach Definition 2.8 $\Gamma \rhd (\mathbf{method}\ m = e;\ r')[^{\tilde{o}}/_{\mathit{self}}] :: \phi \oplus (m : \tau';\ \emptyset)$.

Die restlichen Fälle verlaufen ähnlich. □

Nach diesen Vorbereitungen können wir nun daran gehen das *Preservation-Theorem* zu formulieren und beweisen, welches uns die erste wichtige Eigenschaft für die Typsicherheit der Programmiersprache $\mathcal{L}_o^t$ liefert: Die Wohlgetyptheit und der Typ eines Ausdrucks oder einer Reihe bleibt über small steps hinweg erhalten. Anders ausgedrückt, wenn sich für einen Ausdruck e in einer Typumgebung Γ der Typ τ herleiten läßt, und für diesen Ausdruck existiert ein small step $e \to e'$, dann läßt sich auch für e' in der Typumgebung Γ der Typ τ herleiten.

Satz 2.5 (Typerhaltung, „Preservation“)

(a) $\forall \Gamma \in \mathit{TEnv} : \forall e, e' \in \mathit{Exp} : \forall \tau \in \mathit{Type} : \Gamma^\star \triangleright e :: \tau \wedge e \to e' \;\Rightarrow\; \Gamma^\star \triangleright e' :: \tau$

(b) $\forall \Gamma \in \mathit{TEnv} : \forall r, r' \in \mathit{Row} : \forall \phi \in \mathit{RType} : \Gamma \triangleright r :: \phi \wedge r \to r' \;\Rightarrow\; \Gamma \triangleright r' :: \phi$

Wir wählen $\Gamma^\star$ statt Γ, da die Substitution nur definiert ist, wenn $e \in \mathit{Exp}^\star$, also $\Gamma^\star \triangleright e :: \tau$. Für (Send-Unfold) und (Send-Attr) wird dies bereits durch die entsprechenden Typregeln sichergestellt. Aber für (Let-Exec), (Beta-V) und (Unfold) wäre das Preservation-Theorem andernfalls nicht zu beweisen. Dies entspricht der üblichen Vorgehensweise beliebige freie Variablen im Ausdruck zuzulassen (vgl. [RV98, S. 7], [Pie02, S. 106]).

Beweis: Der Beweis erfolgt durch simultane Induktion über die Länge der Herleitung der small steps $e \to e'$ und $r \to r'$, und Fallunterscheidung nach der zuletzt angewandten small step Regel. Wir betrachten dazu exemplarisch die folgenden Fälle.

1.) $\mathit{op}\, n_1\, n_2 \to \mathit{op}^I(n_1, n_2)$ mit (Op).

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

Sei andererseits $\mathit{op} \in \{<, >, \leq, \geq, =\}$. Nach Voraussetzung gilt $\Gamma^\star \triangleright \mathit{op}\, n_1\, n_2 :: \mathbf{bool}$ und insbesondere $\mathit{op} :: \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ wegen (Rop). Dann muss also gelten $\mathit{op}^I(n_1, n_2) \in \mathit{Bool}$ und somit $\Gamma^\star \triangleright \mathit{op}^I(n_1, n_2) :: \mathbf{bool}$.

2.) $(\lambda x : \tau.e)\, v \to e[^v/_x]$ mit (Beta-V).

$\Gamma^\star \triangleright (\lambda x : \tau.e)\, v :: \tau'$ kann nur mit Typregel (App) aus Prämissen der Form $\Gamma^\star \triangleright v :: \tau$ und $\Gamma^\star \triangleright \lambda x : \tau.e :: \tau \to \tau'$ folgen. Letzteres kann wiederum nur mit Typregel (Abstr) aus $\Gamma^\star[^\tau/_x] \triangleright e :: \tau'$ folgen. Mit Lemma 2.7 folgt daraus $\Gamma^\star \triangleright e[^v/_x] :: \tau'$.

3.) $e_1\, e_2 \to e_1'\, e_2$ mit (App-Left) aus $e_1 \to e_1'$.

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

4.) $\mathbf{object}\,(\mathit{self} : \tau)\; r\; \mathbf{end} \to \mathbf{object}\,(\mathit{self} : \tau)\; r'\; \mathbf{end}$ mit (Object-Eval) aus $r \to r'$.

$\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ kann nur mit Typregel (OBJECT) aus Prämissen der Form $(\Gamma^\star)^\star[^\tau/_{\mathit{self}}] \triangleright r :: \phi$ und $\tau = \langle\phi\rangle$ folgen. Nach Induktionsvoraussetzung gilt dann $(\Gamma^\star)^\star[^\tau/_{\mathit{self}}] \triangleright r' :: \phi$ und mit Typregel (OBJECT) folgt $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r'\ \mathbf{end} :: \tau$.

5.) $\mathbf{val}\ a = e;\ r \to \mathbf{val}\ a = e';\ r$ mit (ATTR-EVAL) aus $e \to e'$.

$\Gamma \triangleright \mathbf{val}\ a = e;\ r :: \phi$ kann nur mit Typregel (ATTR) aus Prämissen der Form $\Gamma^\star \triangleright e :: \tau$ und $\Gamma[^\tau/_a] \triangleright r :: \phi$ folgen. Nach Induktionsvoraussetzung gilt also $\Gamma^\star \triangleright e' :: \tau$, und somit wegen (ATTR) $\Gamma \triangleright \mathbf{val}\ a = e';\ r :: \phi$.

6.) $\mathbf{object}\,(\mathit{self} : \tau)\ \omega\ \mathbf{end}\#m \to \omega[^{\mathbf{object}\,(\mathit{self}:\tau)\ \omega\ \mathbf{end}}/_{\mathit{self}}]\#m$ mit (SEND-UNFOLD).

$\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ \omega\ \mathbf{end}\#m :: \tau'$ kann nur mit Typregel (SEND) folgen aus $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ \omega\ \mathbf{end} :: \langle m : \tau';\ \phi\rangle$, was wiederum nur mit Typregel (OBJECT) aus $(\Gamma^\star)^\star[^\tau/_{\mathit{self}}] \triangleright \omega :: (m : \tau';\ \phi)$ folgen kann. Wegen $\mathit{dom}\,(\Gamma^\star) \cap \mathit{Attribute} = \emptyset$ gilt $\Gamma^\star \triangleright \omega[^{\mathbf{object}\,(\mathit{self}:\tau)\ \omega\ \mathbf{end}}/_{\mathit{self}}] :: (m : \tau';\ \phi)$ nach Lemma 2.9. Hieraus wiederum folgt $\Gamma^\star \triangleright \omega[^{\mathbf{object}\,(\mathit{self}:\tau)\ \omega\ \mathbf{end}}/_{\mathit{self}}]\#m :: \tau'$ mit Typregel (SEND').

7.) $(\mathbf{val}\ a = v;\ \omega)\#m \to \omega[^v/_a]\#m$ mit (SEND-ATTR).

$\Gamma^\star \triangleright (\mathbf{val}\ a = v;\ \omega)\#m :: \tau$ kann ausschliesslich mit Typregel (SEND') aus $\Gamma^\star \triangleright \mathbf{val}\ a = v;\ \omega :: (m : \tau;\ \phi)$ folgen. Dies wiederum kann nur mit Typregel (ATTR) aus Prämissen der Form $(\Gamma^\star)^\star \triangleright v :: \tau'$ und $\Gamma^\star[^{\tau'}/_a] \triangleright \omega :: (m : \tau;\ \phi)$ folgen. Wegen Lemma 2.7 gilt dann $\Gamma^\star \triangleright \omega[^v/_a] :: (m : \tau;\ \phi)$ und mit Typregel (SEND') folgt schliesslich $\Gamma^\star \triangleright \omega[^v/_a]\#m :: \tau$.

8.) $(\mathbf{method}\ m = e;\ \omega)\#m \to e$ mit (SEND-EXEC).

$\Gamma^\star \triangleright (\mathbf{method}\ m = e;\ \omega)\#m :: \tau$ kann wieder nur mit Typregel (SEND') aus $\Gamma^\star \triangleright \mathbf{method}\ m = e;\ \omega :: (m : \tau;\ \emptyset) \oplus \phi$ folgen. Dies kann nur mit Typregel (METHOD) aus Prämissen der Form $\Gamma^\star \triangleright e :: \tau$ und $\Gamma^\star \triangleright \omega :: \phi$ folgen. Die erste Prämisse sichert somit die Typerhaltung.

Die übrigen Fälle verlaufen analog. □

Korollar 2.8 $\forall e, e' \in \mathit{Exp} : \forall \tau \in \mathit{Type} :\ [\,] \triangleright e :: \tau \ \wedge\ e \to e' \ \Rightarrow\ [\,] \triangleright e' :: \tau$

Beweis: Folgt unmittelbar aus Satz 2.5. □

TODO: Überleitung, wir beschränken uns hierbei auf die interessanten Aussagen, die wir im Beweis des Progress-Theorems benutzen werden.

Lemma 2.10 (Canonical Forms)

(a) Für alle $v \in \mathit{Val}$ *mit* $[\,] \triangleright v :: \mathbf{int}$ *gilt* $v \in \mathit{Int}$.

(b) Für alle $v \in \mathit{Val}$ *mit* $[\,] \triangleright v :: \tau \to \tau'$ *gilt genau eine der folgenden Aussagen:*

1. $v \in \mathit{Op}$
2. $v = \mathit{op}\, v_1$ *mit* $\mathit{op} \in \mathit{Op}$ *und* $v_1 \in \mathit{Val}$
3. $v = \lambda x : \tau.e$ *mit* $x \in \mathit{Var}$ *und* $e \in \mathit{Exp}$.

(c) Für alle $v \in \mathit{Val}$ *mit* $[\,] \triangleright v :: \langle\phi\rangle$ *gilt* $v = \mathbf{object}\,(\mathit{self} : \langle\phi\rangle)\ \omega\ \mathbf{end}$.

Beweis: Trivial. □

Satz 2.6 (Existenz des Übergangsschritts, „Progress“)

(a) $\forall e \in \mathit{Exp}, \tau \in \mathit{Type} : [\,] \triangleright e :: \tau \Rightarrow (e \in \mathit{Val} \vee \exists e' \in \mathit{Exp} : e \to e')$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma^+ \triangleright r :: \phi \Rightarrow (r \in \mathit{RVal} \vee \exists r' \in \mathit{Row} : r \to r')$

In Worten besagt das *Progress-Theorem* letztlich, dass für ein abgeschlossener, wohlgetypter Ausdruck entweder bereits ein Wert ist, oder für diesen Ausdruck ein small step existiert. Wegen Lemma 2.3 ist auch klar, dass sich diese beiden Möglichkeiten gegenseitig ausschliessen. Entsprechendes gilt für Reihen, wobei hier allerdings keine Abgeschlossenheit für die Reihe gefordert wird, sondern lediglich keine freien Variablen. Im Beweis wird beim Fall für die (Object) klar, warum der Satz sonst nicht zu beweisen wäre.

Beweis: Wir führen den Beweis durch simultane Induktion über die Länge der Herleitung der Typurteile $[\,] \triangleright e :: \tau$ und $\Gamma^+ \triangleright r :: \phi$ und Fallunterscheidung nach der Form von e und r. Wir betrachten exemplarisch die folgenden Fälle.

1.) Für alle Konstanten c gilt $c \in \mathit{Val}$.

2.) Für Namen id existiert kein τ mit $[\,] \triangleright \mathit{id} :: \tau$.

3.) Für Abstraktionen $\lambda \mathit{id} : \tau'.e'$ gilt $(\lambda \mathit{id} : \tau'.e') \in \mathit{Val}$.

4.) Für Applikationen $e_1\, e_2$ kann $[\,] \triangleright e_1\, e_2 :: \tau$ nur mit Typregel (App) aus Prämissen der Form $[\,] \triangleright e_1 :: \tau' \to \tau$ und $[\,] \triangleright e_2 :: \tau'$ folgen.

Wenn $e_1 \notin \mathit{Val}$, dann existiert nach Induktionsvoraussetzung ein e_1' mit $e_1 \to e_1'$. Folglich existiert mit Regel (App-Left) ein small step $e_1\, e_2 \to e_1'\, e_2$.

Andererseits existiert für $e_1 \in \mathit{Val}, e_2 \notin \mathit{Val}$ nach Induktionsvoraussetzung ein e_2' mit $e_2 \to e_2'$, und somit ein small step $e_1\, e_2 \to e_1\, e_2'$ mit Regel (App-Right).

Es bleibt der Fall $e_1, e_2 \in \mathit{Val}$. Wegen $[\,] \triangleright e_1 :: \tau' \to \tau$ sind gemäß Lemma 2.10 die folgenden drei Fälle zu unterscheiden.

1.) Für $e_1 \in \mathit{Op}$ ist $e_1\, e_2 \in \mathit{Val}$.

2.) Für $e_1 = \mathit{op}\, v_1$ ist zunächst zu bemerken, dass op nur vom Typ $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$ oder $\mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ sein kann. Mit dem Canonical Forms Lemma (2.10) folgt aus $[\,] \triangleright v_1 :: \mathbf{int}$ und $[\,] \triangleright e_2 :: \mathbf{int}$, dass $v_1, e_2 \in \mathit{Int}$ gilt, und somit existiert ein small step $\mathit{op}\, v_1\, e_2 \to \mathit{op}^I(v_1, e_2)$ mit Regel (Op).

3.) Für $e_1 = \lambda x : \tau_2'.e_1'$ existiert ein small step $(\lambda x : \tau_2'.e_1')\, e_2 \to e_1'[{}^{e_2}/_{x}]$ mit Regel (Beta-V), da nach Voraussetzung $e_2 \in \mathit{Val}$.

5.) Für $\omega \# m$ kann $[\,] \triangleright \omega \# m :: \tau$ nur mit Typregel (Send') aus $[\,] \triangleright \omega :: (m : \tau;\, \phi)$ folgen. Damit muss gelten $\omega \neq \epsilon$.

Falls $\omega = (\mathbf{val}\ a = v;\, \omega')$, dann existiert mit Regel (Send-Attr) ein small step $(\mathbf{val}\ a = v;\, \omega')\#m \to \omega'[{}^{v}/_{a}]\#m$.

Für $\omega = (\mathbf{method}\ m' = e;\, \omega')$ mit $m \neq m' \vee m \in \mathit{dom}_m(\omega')$ existiert ein small step $(\mathbf{method}\ m' = e;\, \omega')\#m \to \omega'\#m$ mit Regel (Send-Skip).

Für $\omega = (\mathbf{method}\, m = e;\, \omega')$ mit $m \notin \mathit{dom}_m(\omega')$ existiert mit Regel (Send-Exec) ein small step $(\mathbf{method}\ m = e;\, \omega')\#m \to e$.

6.) Für Methodenaufrufe $e\#m$ kann $[\,] \triangleright e\#m :: \tau'$ nur mit Typregel (Send) aus $[\,] \triangleright e :: \tau$ mit $\tau = \langle m : \tau';\, \phi \rangle$ folgen.

Für $e \notin \mathit{Val}$ existiert nach Induktionsvoraussetzung ein e' mit $e \to e'$, und somit ein small step $e\#m \to e'\#m$ mit Regel (Send-Eval).

Für $e \in \mathit{Val}$ gilt nach Lemma 2.10 $e = \mathbf{object}\,(\mathit{self} : \tau)\ \omega\ \mathbf{end}$. Somit existiert ein small step $\mathbf{object}\,(\mathit{self} : \tau)\ \omega\ \mathbf{end}\#m \to \omega[{}^{\mathbf{object}\,(\mathit{self}:\tau)\ \omega\ \mathbf{end}}/_{\mathit{self}}]\#m$ mit Regel (Send-Unfold).

7.) Für Objekte $\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}$ kann $[\,] \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ ausschliesslich mit Typregel (Object) aus Prämissen der Form $[\mathit{self} : \tau] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen.

Falls $r \notin \mathit{RVal}$, dann existiert nach Induktionsvoraussetzung ein r' mit $r \to r'$. Also existiert ein small step $\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} \to \mathbf{object}\,(\mathit{self} : \tau)\ r'\ \mathbf{end}$ mit Regel (Object-Eval).

Ist andererseits $r \in \mathit{RVal}$, dann gilt $(\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}) \in \mathit{Val}$.

8.) Für Attributdeklarationen $\mathbf{val}\ a = e;\, r$ kann $\Gamma^+ \triangleright \mathbf{val}\ a = e;\, r :: \phi$ nur mit Typregel (Attr) aus $(\Gamma^+)^\star \triangleright e :: \tau$ und $\Gamma^+[{}^{\tau}/_{a}] \triangleright r :: \phi$ folgen.

Für $e \notin \mathit{Val}$ existiert wegen $(\Gamma^+)^\star = [\,]$ nach Induktionsvoraussetzung ein e' mit $e \to e'$. Also existiert ein small step $\mathbf{val}\ a = e;\ r \to \mathbf{val}\ a = e';\ r$ mit Regel (Attr-Left).

Für $e \in \mathit{Val}, r \notin \mathit{RVal}$ existiert nach Induktionsvoraussetzung ein r' mit $r \to r'$, und folglich ein small step $\mathbf{val}\ a = e;\ r \to \mathbf{val}\ a = e;\ r'$ mit Regel (Attr-Right).

Es bleibt der Fall $e \in \mathit{Val}, r \in \mathit{RVal}$. Dann ist $(\mathbf{val}\ a = e;\ r) \in \mathit{RVal}$.

9.) Für Methodendeklarationen $\mathbf{method}\ m = e;\ r$ kann $\Gamma^+ \triangleright \mathbf{method}\ m = e;\ r :: (m : \tau; \emptyset) \oplus \phi$ ausschliesslich mit Typregel (Method) aus Prämissen der Form $\Gamma^+ \triangleright e :: \tau$ und $\Gamma^+ \triangleright r :: \phi$ folgen.

Falls $r \notin \mathit{RVal}$, dann existiert nach Induktionsvoraussetzung ein r' mit $r \to r'$, also auch ein small step $\mathbf{method}\ m = e;\ r \to \mathbf{method}\ m = e;\ r'$ mit Regel (Method-Right).

Für $r \in \mathit{RVal}$ ist $(\mathbf{method}\ m = e;\ r) \in \mathit{RVal}$.

Die restlichen Fälle verlaufen analog. □

TODO: Überleitung

Satz 2.7 (Typsicherheit, „Safety“) *Wenn $[\,] \triangleright e :: \tau$, dann bleibt die Berechnung für e nicht stecken.*

Beweis: Wir führen den Beweis indirekt. Sei dazu angenommen, die Berechnung für e bleibe stecken. Dann ist die Berechnung von der Form $e = e_0 \to \ldots \to e_n$ mit $e_n \not\to$ und $e_n \notin \mathit{Val}$ für ein $n \geq 0$. Wegen $[\,] \triangleright e :: \tau$ folgt mit Satz 2.5 induktiv $[\,] \triangleright e_i :: \tau$ für alle $i = 1, \ldots, n$. Also gilt insbesondere $[\,] \triangleright e_n :: \tau$, aber $e_n \notin \mathit{Val}$ und $e_n \not\to$. Dies steht im Widerspruch zu Satz 2.6. □

TODO: Prosa

Korollar 2.9 *Sei $e \in \mathit{Exp}$ und $\tau \in \mathit{Type}$. Wenn $[\,] \triangleright e :: \tau$, dann gilt genau eine der folgenden Aussagen.*

(a) Es existiert ein $v \in \mathit{Val}$ mit $e \xrightarrow{*} v$.

(b) Die Berechnung von e divergiert.

Beweis: Folgt unmittelbar aus Satz 2.7. □

3 Subtyping

Im vorangegangen Kapitel wurde eine einfache objekt-orientierte Programmiersprache eingeführt, basierend auf den Grundlagen einer einfach getypten, funktionalen Programmiersprache. In diesem Kapitel betrachten wir die Erweiterung dieser Programmiersprache um ein fundamentales, objekt-orientiertes Konzept: *Subtyping*.

Subtyping ist eines der Schlüsselkonzepte der objekt-orientierten Denkweise, obwohl es in den meisten gängigen Programmiersprachen nicht in Reinform, sondern nur gekoppelt an andere Konzepte wie *Vererbung* auftritt. In der Literatur, insbesondere im Bereich der Softwaretechnik, wird Subtyping oft als *subtype polymorphism* oder einfach *polymorphism* bezeichnet, obwohl diese Bezeichnung eigentlich falsch ist, da es sich beim Subtyping lediglich eine um mögliche Form der Polymorphie handelt[1].

Wir werden in diesem Kapitel Subtyping als eigenständiges Konzept untersuchen. Dazu übertragen wir die in [Pie02, S. 182ff] für eine funktionale Programmiersprache mit Records beschriebenen Überlegungen auf die im vorherigen Kapitel eingeführte Programmiersprache $\mathcal{L}_o^t$.

3.1 Motivation

3.2 Die Subtyprelation

Definition 3.1 (Subtyping Regeln) Die Subtyprelation $\leq$ ist die kleinste Relation auf $\mathit{Type} \times \mathit{Type}$, die sich mit den folgenden Regeln herleiten läßt.

[1] [Pie02, S. 340f] beschreibt unterschiedliche Formen von Polymorphie.

(S-Refl) $\tau \leq \tau$

(S-Trans) $\dfrac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$

(S-Arrow) $\dfrac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$

(S-Obj-Width) $\langle m_1 : \tau_1; \ldots; m_{n+k} : \tau_{n+k} \rangle \leq \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle$

(S-Obj-Depth) $\dfrac{\tau_i \leq \tau_i' \text{ für } i = 1, \ldots, n}{\langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle \leq \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle}$

TODO: Wir wollen nun zeigen, dass die Subtyprelation antisymmetrisch ist.

Lemma 3.1 $\forall \tau_1, \tau_2 \in \mathit{Type} : \tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_1 \Rightarrow \tau_1 = \tau_2$

Beweis: Wir führen den Beweis mittels Induktion über die Länge der Herleitung von $\tau_1 \leq \tau_2$ und Fallunterscheidung nach der zuletzt angewandten Regel.

1.) $\tau_1 \leq \tau_2$ mit Regel (S-Refl). Dann folgt unmittelbar $\tau_1 = \tau_2$.

2.) $\tau_1 \leq \tau_2$ mit Regel (S-Trans). Dann existiert ein $\tau_3 \in \mathit{Type}$ mit $\tau_1 \leq \tau_3$ und $\tau_3 \leq \tau_2$. Aus $\tau_2 \leq \tau_1$ und $\tau_1 \leq \tau_3$ folgt mit Regel (S-Trans) $\tau_2 \leq \tau_3$, und mit $\tau_3 \leq \tau_2$ folgt nach Induktionsvoraussetzung $\tau_2 = \tau_3$. Analog folgt $\tau_3 \leq \tau_1$ aus $\tau_3 \leq \tau_2$ und $\tau_2 \leq \tau_1$, und mit $\tau_1 \leq \tau_3$ gilt nach Induktionsvoraussetzung $\tau_1 \leq \tau_3$. Zusammenfassend folgt als $\tau_1 = \tau_3 = \tau_2$.

3.) $\tau_1 \leq \tau_2$ mit Regel (S-Arrow) kann nur mit $\tau_1 = \tau_1' \to \tau_1''$ und $\tau_2 = \tau_2' \to \tau_2''$ aus Prämissen der Form $\tau_2' \leq \tau_1'$ und $\tau_1'' \leq \tau_2''$ folgen. $\tau_2 \leq \tau_1$ kann also nur mit den Regeln (S-Refl), (S-Trans) und (S-Arrow) hergeleitet werden:

   1.) $\tau_2 \leq \tau_1$ mit (S-Refl). Dann folgt unmittelbar $\tau_2 = \tau_1$.

   2.) $\tau_2 \leq \tau_1$ mit (S-Trans). Dann existiert nach Voraussetzung ein $\tau_3 \in \mathit{Type}$ mit $\tau_2 \leq \tau_3$ und $\tau_3 \leq \tau_1$. Aus $\tau_1 \leq \tau_2$ und $\tau_2 \leq \tau_3$ folgt mit Regel (S-Trans) $\tau_1 \leq \tau_3$, und nach Induktionsvoraussetzung somit $\tau_1 = \tau_3$. Analog folgt aus $\tau_3 \leq \tau_1$ und $\tau_1 \leq \tau_2$ dann $\tau_3 \leq \tau_2$ und mit Induktionsvoraussetzung $\tau_3 = \tau_2$. Zusammenfassend gilt also $\tau_1 = \tau_3 = \tau_2$.

   3.) $\tau_2 \leq \tau_1$ mit (S-Arrow) kann ausschliesslich aus Prämissen der Form $\tau_1' \leq \tau_2'$ und $\tau_2'' \leq \tau_1''$ entstanden sein. Mit Induktionsvoraussetzung folgt $\tau_1' = \tau_2'$ und $\tau_1'' = \tau_2''$, also insbesondere $\tau_1' \to \tau_1'' = \tau_2' \to \tau_2''$.

4.) $\tau_1 \leq \tau_2$ mit Regel (S-Obj-Width). Dann ist $\tau_1 = \langle m_1 : \tau'_1; \ldots; m_{n+k} : \tau'_{n+k} \rangle$ und $\tau_2 = \langle m_1 : \tau'_1; \ldots; m_n : \tau'_n \rangle$ mit $n, k \in \mathbb{N}$. Also ist $\tau_2 \leq \tau_1$ nur mit den Regeln (S-Refl), (S-Trans), (S-Obj-Depth) und (S-Obj-Width) herleitbar:

   1.) $\tau_2 \leq \tau_1$ mit (S-Refl) oder (S-Trans) folgt analog zum vorhergehenden Fall.

   2.) $\tau_2 \leq \tau_1$ mit (S-Obj-Depth) oder (S-Obj-Width) kann nur mit $k = 0$ folgen, also ist $\tau_2 = \tau_1$.

5.) $\tau_1 \leq \tau_2$ mit Regel (S-Obj-Depth) kann nur mit $\tau_1 = \langle m_1 : \tau'_1; \ldots; m_n : \tau'_n \rangle$ und $\tau_2 = \langle m_1 : \tau''_1; \ldots; m_n : \tau''_n \rangle$ aus Prämissen der Form $\tau'_i \leq \tau''_i$ für $i = 1, \ldots, n$ folgen. Also kann $\tau_2 \leq \tau_1$ ausschliesslich mit den Regel (S-Refl), (S-Trans), (S-Obj-Depth) und (S-Obj-Width) hergeleitet werden:

   1.) $\tau_2 \leq \tau_1$ mit (S-Refl) oder (S-Trans) folgt analog zum vorhergehenden Fall.

   2.) $\langle m_1 : \tau''_1; \ldots; m_n : \tau''_n \rangle \leq \langle m_1 : \tau'_1; \ldots; m_n : \tau'_n \rangle$ mit Regel (S-Obj-Depth) kann nur aus $\tau''_i \leq \tau'_i$ für $i = 1, \ldots, n$ folgen. Nach Induktionsvoraussetzung gilt $\tau'_i = \tau''_i$ für $i = 1, \ldots, n$ und es folgt $\tau_1 = \tau_2$.

   3.) $\langle m_1 : \tau''_1; \ldots; m_n : \tau''_n \rangle \leq \langle m_1 : \tau'_1; \ldots; m_n : \tau'_n \rangle$ mit Regel (S-Obj-Width) kann nur mit $\tau''_i = \tau'_i$ für $i = 1, \ldots, n$ folgen. Also ist $\tau_1 = \tau_2$. □

TODO: Überleitung

Proposition 3.1 *$(Type, \leq)$ ist eine partielle Ordnung.*

Beweis: Eine partielle Ordnung ist reflexiv, transitiv und antisymmetrisch. Reflexivität und Transitivität folgen unmittelbar aus der Definition des Regelwerks, und wegen Lemma 3.1 folgt die Antisymmetrie. □

TODO: Betrachten wir noch einige weitere Eigenschaften der Subtyprelation.

Lemma 3.2 (Maximale und minimale Typen)

(a) $\forall \tau \in Type, \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} : \tau \leq \tau_\beta \vee \tau_\beta \leq \tau \Rightarrow \tau = \tau_\beta$

(b) $\forall \tau \in Type : \langle\rangle \leq \tau \Rightarrow \tau = \langle\rangle$

Im wesentlichen besagt das Lemma, dass Basistypen bezüglich der Subtyprelation sowohl maximal als auch minimal sind, denn es existiert weder ein größerer noch ein kleinerer Typ. Desweiteren ist der leere Objekttyp maximal. Wir werden dieses Lemma benutzen um nachfolgend eine Verallgemeinerung, das sogenannte Subtyping-Lemma, zu beweisen.

Beweis:

(a) Der Beweis erfolgt durch Induktion über die Länge der Herleitung von $\tau \leq \tau_\beta$ und Fallunterscheidung nach der zuletzt angewandten Regel. Offensichtlich kommen für $\tau \leq \tau_\beta$ nur die Regeln (S-Refl) und (S-Trans) in Frage.

1.) $\tau \leq \tau_\beta$ mit (S-Refl), dann ist $\tau = \tau_\beta$.

2.) $\tau \leq \tau_\beta$ mit (S-Trans). Dann existiert ein $\tau' \in \mathit{Type}$ mit $\tau \leq \tau'$ und $\tau' \leq \tau_\beta$. Nach Induktionsvoraussetzung gilt $\tau' = \tau_\beta$ und $\tau = \tau'$, also $\tau = \tau_\beta$.

Für $\tau_\beta \leq \tau$ folgt ebenso einfach $\tau_\beta = \tau$.

(b) Wir führen den Beweis wiederum durch Induktion über die Länge der Herleitung von $\langle\rangle \leq \tau$ und Fallunterscheidung nach der zuletzt angewandten Regel. Dazu sind lediglich die Regeln (S-Refl), (S-Trans), (S-Obj-Depth) und (S-Obj-Width) zu betrachten.

1.) Für $\langle\rangle \leq \tau$ mit (S-Refl) gilt $\tau = \langle\rangle$.

2.) Für $\langle\rangle \leq \tau$ mit (S-Trans) existiert nach Definition ein $\tau' \in \mathit{Type}$ mit $\langle\rangle \leq \tau'$ und $\tau' \leq \tau$. Nach Induktionsvoraussetzung folgt $\tau' = \langle\rangle$ aus $\langle\rangle \leq \tau'$. Wiederum mit Induktionsvoraussetzung folgt schliesslich aus $\tau' = \langle\rangle$ und $\tau' \leq \tau$ das gewünschte Ergebnis $\tau = \tau' = \langle\rangle$.

3.) $\langle\rangle \leq \tau$ kann mit (S-Obj-Depth) nur dann hergeleitet werden, wenn $\tau = \langle m_1 : \tau_1; \ldots; m_n : \tau_n\rangle$ mit $n = 0$ gilt, also $\tau = \langle\rangle$.

4.) Analog folgt $\tau = \langle\rangle$ für den Fall, dass (S-Obj-Width) die zuletzt angewandte Regel ist. □

Basierend auf diesen Erkenntnissen können wir nun noch einige leicht zu beweisende Eigenschaften der Subtyprelation festhalten, die intuitiv sofort ersichtlich sind.

Proposition 3.2

(a) $(\mathit{Type}, \leq)$ ist keine totale Ordnung.

(b) Es existiert kein größter Typ bezüglich der Subtyprelation.

(c) Es existiert kein kleinster Typ bezüglich der Subtyprelation.

Beweis: Die Beweise erfolgen indirekt. Wir betrachten dazu nur den ersten Teil.

(a) Angenommen die Ordnung $(\mathit{Type}, \leq)$ wäre total, dann müsste für alle Typen $\tau_1, \tau_2 \in \mathit{Type}$ entweder $\tau_1 \leq \tau_2$ oder $\tau_2 \leq \tau_1$ gelten. Sei dazu $\tau_1 = \mathbf{bool}$ und $\tau_2 = \mathbf{int}$. Nach Lemma 3.2 folgt sowohl aus $\mathbf{bool} \leq \mathbf{int}$ wie auch aus $\mathbf{int} \leq \mathbf{bool}$, dass $\mathbf{bool} = \mathbf{int}$ gelten würde, also der gesuchte Widerspruch.

(b) Angenommen $\tau_\top \in \mathit{Type}$ wäre der größte Typ bezüglich der Subtyprelation. Insbesondere müsste dann $\mathbf{int} \leq \tau_\top$ und $\langle\rangle \leq \tau_\top$ gelten. Wegen Lemma 3.2 wäre dann folglich $\tau_\top = \mathbf{int}$ und $\tau_\top = \langle\rangle$, also $\langle\rangle = \mathbf{int}$, der gesuchte Widerspruch.

(c) Ist ebenso offensichtlich und läßt sich leicht zu einem Widerspruch führen. □

TODO: Überleitung

Lemma 3.3 (Subtyping-Lemma) *$\tau \leq \tau'$ gilt genau dann, wenn eine der folgenden Aussagen zutrifft:*

(a) $\tau = \tau'$

(b) $\tau = \tau_1 \to \tau_2$ *und* $\tau' = \tau_1' \to \tau_2'$ *mit* $\tau_1' \leq \tau_1$ *und* $\tau_2 \leq \tau_2'$

(c) $\tau = \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle$ *und* $\tau' = \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle$ *mit* $\{m_1', \ldots, m_l'\} \subseteq \{m_1, \ldots, m_k\}$ *und* $\tau_i \leq \tau_j'$ *für alle* $i, j \in \mathbb{N}$ *mit* $m_i = m_j'$

Beweis: Wir beweisen die Äquivalenz in zwei Schritten.

„⇐“ Für diese Richtung genügt es zu zeigen, dass sich für die angegebenen Fälle die Aussage $\tau \leq \tau'$ mit den Regeln fürs Subtyping aus den Bedingungen herleiten lässt.

(a) Aus $\tau = \tau'$ folgt unmittelbar $\tau \leq \tau'$ mit Regel (S-Refl).

(b) Aus $\tau_1' \leq \tau_1$ und $\tau_2 \leq \tau_2'$ folgt mit Regel (S-Arrow) $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$, also $\tau \leq \tau'$.

(c) Für die Formel $\langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle \leq \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle$ geben wir eine Herleitung an, wobei bei der Rückwärtsanwendung der Regel (S-Trans) hierbei zunächst ein Zwischentyp zu finden ist, welcher genau die Methodennamen von τ mit den Methodentypen von τ' enthält. Sei dazu im Folgenden

$$\tau_i'' \quad = \quad \begin{cases} \tau_j' & \text{falls } m_i = m_j' \\ \tau_i & \text{sonst.} \end{cases}$$

$$\text{S-Trans}\ \frac{\text{S-Obj-Depth}\ \dfrac{\tau_1 \leq \tau_1'' \quad \ldots \quad \tau_k \leq \tau_k''}{\langle m_i : \tau_i^{i=1,\ldots,k} \rangle \leq \langle m_i : \tau_i''^{i=1,\ldots,k} \rangle} \qquad \text{S-Obj-Width}\ \langle m_i : \tau_i''^{i=1,\ldots,k} \rangle \leq \langle m_i' : \tau_i'^{i=1,\ldots,l} \rangle}{\langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle \leq \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle}$$

$\tau_i \leq \tau_i''$ folgt hierbei entweder nach Voraussetzung, falls $\tau_i'' = \tau_j'$ für $m_i = m_j'$, oder mit Regel (S-Refl), da in diesem Fall $\tau_i'' = \tau_i$ gilt.

„⇒“ Induktion über die Länge der Herleitung von $\tau \leq \tau'$ und Fallunterscheidung nach der zuletzt angewandten Regel.

1.) $\tau \leq \tau'$ mit Regel (S-Refl), dann gilt $\tau = \tau'$.

2.) $\tau \leq \tau'$ mit Regel (S-Trans), dann existiert ein τ'' mit $\tau \leq \tau''$ und $\tau'' \leq \tau'$. Nach Induktionsvoraussetzung gilt jeweils eine der Aussagen des Lemmas für $\tau \leq \tau''$ und $\tau'' \leq \tau$. Durch Fallunterscheidung nach der Form von τ'' zeigen wir, dass dann auch für $\tau \leq \tau'$ eine der Aussagen gilt:

   1.) Für $\tau'' \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$ folgt nach Lemma 3.2 $\tau = \tau'' = \tau'$.

   2.) Für $\tau'' = \tau_1'' \to \tau_2''$ muss $\tau = \tau_1 \to \tau_2$ und $\tau' = \tau_1' \to \tau_2'$ gelten. O.B.d.A. gilt dann $\tau_1'' \leq \tau_1$, $\tau_2 \leq \tau_2''$, $\tau_1' \leq \tau_1''$ und $\tau_2'' \leq \tau_2'$. Mit Regel (S-Trans) folgt unmittelbar $\tau_1' \leq \tau_1$ und $\tau_2' \leq \tau_2$.

   3.) Der Fall $\tau = \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle$ verläuft analog.

3.) $\tau \leq \tau'$ mit Regel (S-Arrow), dann ist $\tau = \tau_1 \to \tau_2$ und $\tau' = \tau_1' \to \tau_2'$ und es gilt $\tau_1' \leq \tau_1$ und $\tau_2 \leq \tau_2'$.

4.) $\tau \leq \tau'$ mit (S-Obj-Width) kann nur aus $\tau = \langle m_1 : \tau_1; \ldots; m_{n+k} : \tau_{n+k} \rangle$ und $\tau' = \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle$ folgen, und es gilt $\{m_1, \ldots, m_n\} \subseteq \{m_1, \ldots, m_{n+k}\}$. Mit Regel (S-Refl) folgt darüberhinaus $\tau_i \leq \tau_i$ für $i = 1, \ldots, n$.

5.) $\tau \leq \tau'$ mit (S-Obj-Depth) kann nur mit $\tau = \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle$ und $\tau' = \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle$ aus $\tau_i \leq \tau_i'$ für $i = 1, \ldots, n$ folgen. Offensichtlich gilt $\{m_1, \ldots, m_n\} \subseteq \{m_1, \ldots, m_n\}$. □

Das Subtyping-Lemma lässt sich für Funktionstypen auch auf eine etwas andere Weise ausdrücken, die den Einsatz des Lemmas in den späteren Beweis vielleicht noch etwas deutlicher hervorhebt. Betrachten wir dazu das folgende Korollar.

Korollar 3.1 *Seien $\tau, \tau_1, \tau_2 \in Type$. Dann gilt:*

(a) $\tau \leq \tau_1 \to \tau_2 \Rightarrow \exists \tau_1', \tau_2' \in Type : \tau = \tau_1' \to \tau_2' \wedge \tau_1 \leq \tau_1' \wedge \tau_2' \leq \tau_2$

(b) $\tau_1 \to \tau_2 \leq \tau \Rightarrow \exists \tau_1', \tau_2' \in Type : \tau = \tau_1' \to \tau_2' \wedge \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2'$

In Worten besagt das Korollar, dass ein Funktionstyp immer nur zusammen mit einem anderen Funktionstyp in der Subtyprelation stehen kann, wobei die Typen der Parameter kontravariant und die Typen der Ergebnisse kovariant bezüglich der Subtyprelation sein müssen.

Beweis: Folgt unmittelbar aus Lemma 3.3. □

Ein ähnliches – wenn auch weniger übersichtliches – Korollar läßt sich für Objekttypen aus dem Lemma herleiten.

Korollar 3.2 *Seien* $\tau, \tau'_1, \ldots, \tau'_l \in \mathit{Type}$ *und* $m'_1, \ldots, m'_l \in \mathit{Method}$. *Dann gilt:*

(a) $\tau \leq \langle m'_1 : \tau'_1; \ldots; m'_l : \tau'_l \rangle$
$\Rightarrow \exists k \geq l : \exists m_1, \ldots, m_k \in \mathit{Method} : \exists \tau_1, \ldots, \tau_k \in \mathit{Type} :$
$\tau = \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle \wedge \{m'_1, \ldots, m'_l\} \subseteq \{m_1, \ldots, m_k\}$
$\wedge [\forall 1 \leq i \leq k : \forall 1 \leq j \leq l : m_i = m'_j \Rightarrow \tau_i \leq \tau'_j]$

(b) $\langle m'_1 : \tau'_1; \ldots; m'_l : \tau'_l \rangle \leq \tau$
$\Rightarrow \exists k \leq l : \exists m_1, \ldots, m_k \in \mathit{Method} : \exists \tau_1, \ldots, \tau_k \in \mathit{Type} :$
$\tau = \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle \wedge \{m_1, \ldots, m_k\} \subseteq \{m'_1, \ldots, m'_l\}$
$\wedge [\forall 1 \leq i \leq k : \forall 1 \leq j \leq l : m_i = m'_j \Rightarrow \tau'_j \leq \tau_i]$

Beweis: Folgt ebenso unmittelbar aus Lemma 3.3. □

3.3 Die Sprache $\mathcal{L}_o^{sub}$

Nachdem wir uns nun mit der Subtyprelation vertraut gemacht haben, und einige wichtige Eigenschaften bewiesen haben, wollen wir uns nun der Frage zuwenden, wie Subtyping nun in eine typsichere objekt-orientierte Programmiersprache eingebaut werden kann. Für *structual typing* existieren dazu heute im wesentlichen zwei Ansätze:

- Beim *impliziten Subtyping* wird während der Typüberprüfung mit der *Subsumption-Regel* automatisch zu einem größeren Typ übergegangen sofern notwendig. Die Syntax und Semantik der Programmiersprache bleiben dabei unangetastet.

- Beim *expliziten Subtyping* hingegen ist der Programmier dafür verantwortlich, durch sogenannte *coercions* im Programmtext, einen Übergang zu einem Supertyp durchzuführen.

Während das implizite Subtyping in der Theorie am verbreitetsten ist, wird in der Praxis für *structural typing* fast ausschliesslich Subtyping mit coercions verwendet[2]. Der wesentliche Grund hierfür liegt in der Performance des generierten Maschinencodes (vgl. dazu [Pie02, S.200ff]).

In diesem Abschnitt werden wir die Programmiersprache $\mathcal{L}_o^t$ aus dem vorangegangen Kapitel um Subsumption zur Sprache $\mathcal{L}_o^{sub}$ erweitern, und den Aspekt der Typsicherheit dieser Programmiersprache betrachten. Die Syntax und Semantik der Programmiersprache $\mathcal{L}_o^{sub}$ stimmen mit der Programmiersprache $\mathcal{L}_o^t$ überein, es kommt lediglich eine neue Typregel hinzu.

Definition 3.2 (Gültige Typurteile für $\mathcal{L}_o^{sub}$) Ein Typurteil $\Gamma \triangleright_m e :: \tau$ oder $\Gamma \triangleright_m r :: \psi$ heisst *gültig* für $\mathcal{L}_o^{sub}$, wenn es sich mit dem Typregeln von $\mathcal{L}_o^t$ (vgl. Definition 2.20) sowie der Subsumption-Regel

$$(\textsc{Subsume}) \quad \frac{\Gamma \triangleright_m e :: \tau \quad \tau \leq \tau'}{\Gamma \triangleright_m e :: \tau'}$$

herleiten lässt.

Es ist leicht zu sehen, dass durch die Hinzunahme der (Subsume)-Regel die Typeindeutig verloren geht. Betrachten wir dazu beispielsweise den folgenden Ausdruck.

$$\textbf{object}\,(\mathit{self} : \langle m : \textbf{int}\rangle)\ \textbf{method}\ m = 1;\ \ \textbf{end}$$

Nur mit den Typregeln der Sprache $\mathcal{L}_o^t$ lässt sich für diesen Ausdruck in der leeren Typumgebung der Objekttyp $\langle m : \textbf{int}\rangle$ wie folgt herleiten.

$$\dfrac{\dfrac{\dfrac{}{1 :: \textbf{int}}\,\textsc{Int}}{[\mathit{self} : \langle m : \textbf{int}; \emptyset\rangle] \triangleright 1 :: \textbf{int}}\,\textsc{Const} \qquad \dfrac{}{[\mathit{self} : \langle m : \textbf{int}; \emptyset\rangle] \triangleright \epsilon :: \emptyset}\,\textsc{Empty}}{[\mathit{self} : \langle m : \textbf{int}; \emptyset\rangle] \triangleright \textbf{method}\ m = 1; \epsilon :: (m : \textbf{int}; \emptyset)}\,\textsc{Method}$$
$$\dfrac{}{[\,] \triangleright \textbf{object}\,(\mathit{self} : \langle m : \textbf{int}; \emptyset\rangle)\ \textbf{method}\ m = 1; \epsilon\ \textbf{end} :: \langle m : \textbf{int}; \emptyset\rangle}\,\textsc{Object}$$

Andererseits lässt sich für den gleichen Ausdruck in der gleichen Typumgebung ebenfalls der leere Objekttyp $\langle\rangle$ herleiten, durch Anwenden der (Subsume)-Regel im Anschluss an die obige Herleitung des Objekttyps.

[2] Im Gegensatz zu objekt-orientierten Programmiersprachen mit opaquen Typen, wie z.B. Java oder C++, bei denen durchgehend Subsumption zum Einsatz kommt.

$$\textsc{Subsume}\ \dfrac{\textsc{Object}\ \dfrac{\vdots}{[\,] \triangleright \mathbf{object} \ldots \mathbf{end} :: \langle m : \mathbf{int};\, \emptyset \rangle} \qquad \textsc{S-Obj-Width}\ \langle m : \mathbf{int};\, \emptyset \rangle \leq \langle \emptyset \rangle}{[\,] \triangleright \mathbf{object}\,(\mathit{self} : \langle m : \mathbf{int};\, \emptyset \rangle)\ \mathbf{method}\ m = 1;\ \epsilon\ \mathbf{end} :: \langle \emptyset \rangle}$$

Vergleicht man die (Subsume)-Regel mit den bisherigen Typregeln, so wird auch sofort die Ursache dieser Uneindeutigkeit offensichtlich. Bisher waren Typregeln eineindeutig mit der syntaktischen Form eines Ausdrucks verknüpft, zum Beispiel war (App) die einzige auf Applikationen anwendbare Regel und ebenso war (App) ausschliesslich auf Applikationen anwendbar. Im Typsystem der Sprache $\mathcal{L}_o^{sub}$ kann nun aber unabhängig von der syntaktischen Form des Ausdrucks immer entweder die aus der Sprache $\mathcal{L}_o^t$ geerbte Typregel oder die (Subsume)-Regel angewandt werden, wie an den beiden Typherleitungen für den Ausdruck

$$\mathbf{object}\,(\mathit{self} : \langle m : \mathbf{int} \rangle)\ \mathbf{method}\ m = 1;\ \ \mathbf{end}$$

gut zu ersehen ist. Wir werden später auf das Thema Typeindeutigkeit in einer Sprache mit Subtyping eingehen. Für den Augenblick halten wir lediglich fest, dass die Typeindeutigkeit in der Sprache $\mathcal{L}_o^{sub}$ nicht gilt.

3.3.1 Typsicherheit

Wir wollen nun für die Sprache $\mathcal{L}_o^{sub}$, ähnlich wie zuvor für die Sprache $\mathcal{L}_o^t$, zeigen, dass die Berechnung eines wohlgetypten, abgeschlossenen Ausdrucks nicht stecken bleiben kann, also die Typsicherheit der Programmiersprache $\mathcal{L}_o^{sub}$. Dazu überzeugen wir uns zunächst davon, dass diese nicht trivialerweise aus der Typsicherheit der Programmiersprache $\mathcal{L}_o^t$ folgt, indem wir einen Ausdruck suchen, der in $\mathcal{L}_o^{sub}$ wohlgetypt ist, nicht aber in $\mathcal{L}_o^t$. Betrachten wir dazu die Funktion

$$\lambda o : \langle m : \mathbf{int} \rangle . o\#m,$$

die ein Objekt mit einer Methode m von Typ **int** als Parameter erwartet und diesem Objekt die Nachricht m sendet. Für diese Funktion lässt sich in beiden Typsystemen in der leeren Typumgebung der Typ $\langle m : \mathbf{int} \rangle \rightarrow \mathbf{int}$ wie folgt herleiten.

$$\textsc{Abstr}\ \dfrac{\textsc{Send}\ \dfrac{\textsc{Id}\ \dfrac{}{[o : \langle m : \mathbf{int} \rangle] \triangleright o :: \langle m : \mathbf{int} \rangle}}{[o : \langle m : \mathbf{int} \rangle] \triangleright o\#m :: \mathbf{int}}}{[\,] \triangleright \lambda o : \langle m : \mathbf{int} \rangle . o\#m :: \langle m : \mathbf{int} \rangle \rightarrow \mathbf{int}}$$

Betrachten wir weiterhin das Objekt

$$\textbf{object}\,(\mathit{self} : \langle m : \textbf{int}; n : \textbf{int}\rangle)\ \textbf{method}\ m = 1;\ \textbf{method}\ n = 2;\ \ \textbf{end},$$

welches zwei Methoden von Typ **int** bereitstellt. Mit den Typregeln von $\mathcal{L}_o^t$ lässt sich hierfür offensichtlich der Typ $\langle m : \textbf{int}; n : \textbf{int}\rangle$ herleiten. Gemäß der Typeindeutig der Sprache $\mathcal{L}_o^t$ sind dies die einzigen Typen, die sich mit den Typregeln von $\mathcal{L}_o^t$ für diese beiden Ausdrücke herleiten lassen.

Nennen wir nun die Funktion f_m und das Objekt *obj* und betrachten den folgenden Ausdruck.[3]

$$f_m\, obj$$

Sei $\Gamma = [f_m : \langle m : \textbf{int}\rangle \to \textbf{int}, obj : \langle m : \textbf{int}; n : \textbf{int}\rangle]$ die Typumgebung, in der ein Typ für den Ausdruck bestimmt werden soll. Betrachten wir zunächst eine mögliche Typherleitung in der Sprache $\mathcal{L}_o^t$.

$$\textsc{App}\ \frac{\begin{array}{ll}\textsc{Id} & \textsc{Id}\\ \Gamma \rhd f_m :: \langle m : \textbf{int}\rangle \to \textbf{int} & \Gamma \rhd obj :: \langle m : \textbf{int}; n : \textbf{int}\rangle\end{array}}{\Gamma \rhd f_m\, obj ::}$$

Die Typregel (App) fordert $\langle m : \textbf{int}\rangle = \langle m : \textbf{int}; n : \textbf{int}\rangle$, ein Widerspruch. Also kann für diesen Ausdruck kein Typ hergeleitet werden. Im Typsystem der Sprache $\mathcal{L}_o^{sub}$ hingegen lässt sich ein Typ für diesen Ausdruck herleiten. Hierzu existieren für diesen speziellen Ausdruck grundsätzlich zwei Möglichkeiten den Widerspruch aufzuheben:

(a) den Typ von f_m anpassen zu $\langle m : \textbf{int}; n : \textbf{int}\rangle \to \textbf{int}$,

(b) oder den Typ von *obj* anpassen zu $\langle m : \textbf{int}\rangle$.

Die Typanpassung erfolgt mit der (Subsume)-Regel und den entsprechenden Subtyping-Regeln. Wir betrachten dazu exemplarisch den ersten Fall.

$$\textsc{Subsume}\ \frac{\begin{array}{ll}\textsc{Id} & \textsc{S-Obj-Width} \quad \textsc{S-Refl}\\ \Gamma \rhd f_m :: \langle m : \textbf{int}\rangle \to \textbf{int} & \dfrac{\langle m : \textbf{int}; n : \textbf{int}\rangle \le \langle m : \textbf{int}\rangle \qquad \textbf{int} \le \textbf{int}}{\langle m : \textbf{int}\rangle \to \textbf{int} \le \langle m : \textbf{int}; n : \textbf{int}\rangle \to \textbf{int}}\ \textsc{S-Arrow}\end{array}}{\Gamma \rhd f_m :: \langle m : \textbf{int}; n : \textbf{int}\rangle \to \textbf{int}}$$

[3] Streng genommen müssten wir dafür **let**-Ausdrücke benutzen, der Übersichtlichkeit wegen vermeiden wir dies jedoch an dieser Stelle.

Nach dieser Typanpassung sind die Voraussetzungen für die Typregel (App) erfüllt und für den Gesamtausdruck lässt sich somit der Typ **int** herleiten.

Anhand des vorangegangenen Beispiels haben wir gesehen, dass im Typsystem der Sprache $\mathcal{L}_o^{sub}$ mehr wohlgetypte Ausdrücke existieren als im Typsystem der Sprache $\mathcal{L}_o^t$. Somit können also die Ergebnisse über die Typsicherheit von $\mathcal{L}_o^t$ nicht einfach auf $\mathcal{L}_o^{sub}$ übertragen werden, sondern es müssen teilweise neue Überlegungen für die Sprache mit Subtyping angestellt werden.

Hierzu betrachten wir zunächst die für den Beweis der Typerhaltung notwendigen Lemmata aus Kapitel 2 für die Sprache $\mathcal{L}_o^{sub}$. Die Aussagen der Lemmata wurden bereits für $\mathcal{L}_o^t$ entsprechend geschickt gewählt, so dass diese übernommen werden können. In den Beweisen muss nun zusätzlich zu den Typregeln für die Programmiersprache $\mathcal{L}_o^t$ die (Subsume)-Regel betrachtet werden.

Lemma 3.4 (Typumgebungen und frei vorkommende Namen)

(a) $\forall \Gamma \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma \triangleright e :: \tau \Rightarrow \mathit{free}(e) \subseteq \mathit{dom}(\Gamma)$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma \triangleright r :: \phi \Rightarrow \mathit{free}(r) \subseteq \mathit{dom}(\Gamma)$

Beweis: Der Beweis erfolgt wie gehabt durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi$ und Fallunterscheidung nach der zuletzt angewandten Typregel. Wir betrachten nur den neuen Fall der Subsumption-Regel.

1.) $\Gamma \triangleright e :: \tau$ mit Typregel (Subsume) kann nur aus Prämissen der Form $\Gamma \triangleright e :: \tau'$ und $\tau' \leq \tau$ folgen. Nach Induktionsvoraussetzung gilt also $\mathit{free}(e) \subseteq \mathit{dom}(\Gamma)$, was zu zeigen war.

Die übrigen Fälle sind völlig identisch zum Beweis von Lemma 2.5. □

TODO: Überleitung

Lemma 3.5 (Koinzidenzlemma)

(a) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma_1 \triangleright e :: \tau \wedge \Gamma_1 =_{\mathit{free}(e)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright e :: \tau$

(b) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma_1 \triangleright r :: \phi \wedge \Gamma_1 =_{\mathit{free}(r)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright r :: \phi$

Beweis: Hier ist ebenfalls nur der neue Fall für die Subsumption-Regel zu betrachten, die übrigen Fälle sind bereits durch den Beweis von Lemma 2.6 abgedeckt.

1.) $\Gamma_1 \triangleright e :: \tau$ mit Typregel (Subsume) kann nur aus Prämissen der Form $\Gamma_1 \triangleright e :: \tau'$ und $\tau' \leq \tau$ folgen. Mit Induktionsvoraussetzung folgt daraus $\Gamma_2 \triangleright e :: \tau'$ und mit Typregel (Subsume) schliesslich das gesuchte Ergebnis $\Gamma_2 \triangleright e :: \tau$. □

TODO: Überleitung

Lemma 3.6 (Typurteile und Substitution) *Sei* $id \in Attribute \cup Var$, $\Gamma \in TEnv$, $\tau \in Type$ *und* $e \in Exp$. *Dann gilt:*

(a) $\forall e' \in Exp : \forall \tau' \in Type : \Gamma[^{\tau}/_{id}] \rhd e' :: \tau' \wedge \Gamma^{\star} \rhd e :: \tau \Rightarrow \Gamma \rhd e'[^{e}/_{id}] :: \tau'$

(b) $\forall r \in Row : \forall \phi \in RType : \Gamma[^{\tau}/_{id}] \rhd r :: \phi \wedge \Gamma^{\star} \rhd e :: \tau \Rightarrow \Gamma \rhd r[^{e}/_{id}] :: \phi$

Beweis: Mit der gleichen Argumentation wie im Beweis von Lemma 2.7 für die Sprache $\mathcal{L}_o^t$, wobei ein neuer Fall für die (SUBSUME)-Regel hinzukommt, der allerdings direkt mit Induktionsvoraussetzung folgt. □

Neben dem Lemma für die Substitution von Ausdrücken für Variablen und Attributnamen muss ebenfalls der Beweis für Lemma 2.9 über Substitution für *self* erweitert werden. Hier ist wiederum der Beweis lediglich um einen neuen Fall für die (SUBSUME)-Regel zu ergänzen.

Lemma 3.7 (Typurteile und *self*-Substitution) *Sei* $\Gamma \in TEnv$, $self \in Self$, $\tau \in Type$ *und* $r \in Row$. *Dann gilt:*

(a) Für alle $e \in Exp$ *und* $\tau' \in Type$*:*

(1) $\Gamma[^{\tau}/_{self}] \rhd e :: \tau'$

(2) $\Gamma^{\star} \rhd \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau$

(3) $\forall a \in dom(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^{\star} \rhd r(a) :: \tau_a \Leftrightarrow \Gamma \rhd a :: \tau_a$

$\Rightarrow \Gamma \rhd e[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau'$

(b) Für alle $r' \in Row$ *und* $\phi \in RType$*:*

(1) $\Gamma[^{\tau}/_{self}] \rhd r' :: \phi$

(2) $\Gamma^{\star} \rhd \mathbf{object}\,(self : \tau)\ r \oplus r'\ \mathbf{end} :: \tau$

(3) $\forall a \in dom(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^{\star} \rhd r(a) :: \tau_a \Leftrightarrow \Gamma \rhd a :: \tau_a$

$\Rightarrow \Gamma \rhd r'[^{\mathbf{object}\,(self:\tau)\ r\oplus r'\ \mathbf{end}}/_{self}] :: \phi$

Beweis: Identisch zum Beweis von Lemma 2.9, mit einem neuen Fall für die Subsumption-Regel, der wiederum direkt mit der Induktionsvoraussetzung folgt. □

Zum Beweis der Preservation für die Programmiersprache $\mathcal{L}_o^{sub}$ benötigen wir neben den bisherigen Lemmata noch das folgende Lemma, welches im wesentlichen einer Umkehrung der Typrelation entspricht (üblicherweise als *inversion lemma* bezeichnet). Wir beschränken uns dabei auf die Aussagen, die wir im Beweis der Preservation- und Progress-Sätze benutzen werden, statt das Lemma in seiner allgemeinsten Form anzugeben.

Lemma 3.8 (Umkehrung der Typrelation)

(a) Wenn $\Gamma \triangleright op :: \tau$, *dann gilt* $\tau = \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ *für* $op \in \{+, -, *\}$ *oder* $\tau = \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ *für* $op \in \{\leq, \geq, <, >, =\}$.

(b) Wenn $\Gamma \triangleright \lambda x : \tau_1.e :: \tau_1' \to \tau_2$, *dann gilt* $\Gamma[{}^{\tau_1}/_{x}] \triangleright e :: \tau_2$ *und* $\tau_1' \leq \tau_1$.

(c) Wenn $\Gamma \triangleright e_1\, e_2 :: \tau$, *dann gilt* $\Gamma \triangleright e_1 :: \tau_2' \to \tau$, $\Gamma \triangleright e_2 :: \tau_2$ *und* $\tau_2 \leq \tau_2'$.

(d) Wenn $\Gamma \triangleright \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau'$, *dann gilt* $\Gamma^\star[{}^{\tau}/_{self}] \triangleright r :: \phi$, $\tau = \langle\phi\rangle$ *und* $\tau \leq \tau'$.

Beweis: Die Beweise erfolgen jeweils durch vollständige Induktion über die Länge der Herleitung des Typurteils und Fallunterscheidung nach der zuletzt angewandten Typregel. Hierbei kommen jeweils nur zwei Typregeln in Frage.

(a) $\Gamma \triangleright op :: \tau$ kann nur mit einer der Typregeln (Const) oder (Subsume) hergeleitet worden sein.

1.) Im Fall von (Const) gilt $op :: \tau$, welches wiederum nur mit (Aop) oder (Rop) hergeleitet worden sein kann. Für (Aop) gilt $op \in \{+, -, *\}$ und $\tau = \mathbf{int} \to \mathbf{int} \to \mathbf{int}$, andernfalls muss gelten $op \in \{\leq, \geq, <, >, =\}$ und $\tau = \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$.

2.) Für $\Gamma \triangleright op :: \tau$ mit Typregel (Subsume) gilt $\Gamma \triangleright op :: \tau'$ und $\tau' \leq \tau$. Nach Induktionsvoraussetzung ist $\tau' = \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ und $op \in \{+, -, *\}$ oder $\tau' = \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ und $op \in \{\leq, \geq, <, >, =\}$.

Wir betrachten nur den Fall $\mathbf{int} \to \mathbf{int} \to \mathbf{int} \leq \tau$, der zweite Fall folgt entsprechend. Gemäß Subtyping-Lemma (3.3) existieren $\tau_1, \tau_2 \in \mathit{Type}$ mit $\tau = \tau_1 \to \tau_2$, $\tau_1 \leq \mathbf{int}$ und $\mathbf{int} \to \mathbf{int} \leq \tau_2$. Nach erneuter Anwendung des Lemmas also $\tau_2 = \tau_2' \to \tau_2''$, $\tau_2' \leq \mathbf{int}$ und $\mathbf{int} \leq \tau_2''$. Mit Lemma 3.2 folgt schliesslich $\tau_1 = \tau_2' = \tau_2'' = \mathbf{int}$, also $\tau = \mathbf{int} \to \mathbf{int} \to \mathbf{int}$, was zu zeigen war.

(b) Zur Herleitung des Typurteils $\Gamma \triangleright \lambda x : \tau_1.e :: \tau_1' \to \tau_2$ kommen nur die Typregeln (Abstr) und (Subsume) in Frage.

1.) Falls zuletzt die (Abstr)-Regel angewandt worden ist, gilt nach Voraussetzung $\Gamma[^{\tau_1}/_x] \triangleright e :: \tau_2$ und $\tau_1' = \tau_1$. Mit Subtyping-Regel (S-Refl) folgt dann $\tau_1' \leq \tau_1$.

2.) Für $\Gamma \triangleright \lambda x : \tau_1.e :: \tau_1' \to \tau_2$ mit (Subsume) existiert nach Voraussetzung ein $\tau'' \in \mathit{Type}$ mit $\Gamma \triangleright \lambda x : \tau_1.e :: \tau''$ und $\tau'' \leq \tau_1' \to \tau_2$. Gemäß dem Subtyping-Lemma (3.3) existieren somit $\tau_1'', \tau_2'' \in \mathit{Type}$ mit $\tau'' = \tau_1'' \to \tau_2''$, $\tau_1' \leq \tau_1''$ und $\tau_2'' \leq \tau_2$. Für $\Gamma \triangleright \lambda x : \tau_1.e :: \tau_1'' \to \tau_2''$ folgt dann mit Induktionsvoraussetzung $\Gamma[^{\tau_1}/_x] \triangleright e :: \tau_2''$ und $\tau_1'' \leq \tau_1$. Aus $\tau_1' \leq \tau_1''$ und $\tau_1'' \leq \tau_1$ folgt mit Subtyping-Regel (S-Trans) $\tau_1' \leq \tau_1$, und wegen $\Gamma[^{\tau_1}/_x] \triangleright e :: \tau_2''$ und $\tau_2'' \leq \tau_2$ folgt schliesslich mit Typregel (Subsume) $\Gamma[^{\tau_1}/_x] \triangleright e :: \tau_2$.

(c) Das Typurteil $\Gamma \triangleright e_1\, e_2 :: \tau$ kann ausschliesslich mit Typregel (App) oder Typregel (Subsume) folgen.

1.) $\Gamma \triangleright e_1\, e_2 :: \tau$ mit Typregel (App) bedingt $\Gamma \triangleright e_1 :: \tau_2 \to \tau$ und $\Gamma \triangleright e_2 :: \tau_2$, und mit (S-Refl) folgt $\tau_2 \leq \tau_2$.

2.) $\Gamma \triangleright e_1\, e_2 :: \tau$ mit Typregel (Subsume) kann nur aus Prämissen der Form $\Gamma \triangleright e_1\, e_2 :: \tau'$ und $\tau' \leq \tau$ folgen. Nach Induktionsvoraussetzung existieren $\tau_2, \tau_2' \in \mathit{Type}$ mit $\Gamma \triangleright e_1 :: \tau_2' \to \tau'$, $\Gamma \triangleright e_2 :: \tau_2$ und $\tau_2 \leq \tau_2'$. Schliesslich folgt wegen $\Gamma \triangleright e_1 :: \tau_2' \to \tau'$ und $\tau' \leq \tau$ mit Typregel (Subsume) und den Subtyping-Regeln (S-Arrow) und (S-Refl) $\Gamma \triangleright e_1 :: \tau_2' \to \tau$.

(d) Das Typurteil $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ kann nur mit einer der Typregeln (Object) oder (Subsume) hergeleitet worden sein.

1.) Falls $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ mit Typregel (Object) hergeleitet wurde, existiert nach Voraussetzung ein $\phi \in \mathit{RType}$ mit $\Gamma^\star[^{\tau}/_{\mathit{self}}] \triangleright r :: \phi$ und $\tau = \langle\phi\rangle = \tau'$. Wegen (S-Refl) also $\tau \leq \tau'$.

2.) $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ mit Typregel (Subsume) kann nur aus Prämissen der Form $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau''$ und $\tau'' \leq \tau'$ folgen. Nach Induktionsvoraussetzung existiert ein $\phi \in \mathit{RType}$, so dass gilt $\Gamma^\star[^{\tau}/_{\mathit{self}}] \triangleright r :: \phi$ mit $\tau = \langle\phi\rangle$ und $\tau \leq \tau''$. Mit Subtyping-Regel (S-Trans) folgt $\tau \leq \tau'$ wegen $\tau \leq \tau''$ und $\tau'' \leq \tau'$. □

TODO

Satz 3.1 (Typerhaltung, „Preservation“)

(a) $\forall \Gamma \in \mathit{TEnv} : \forall e, e' \in \mathit{Exp} : \forall \tau \in \mathit{Type} : \Gamma^\star \triangleright e :: \tau \wedge e \to e' \ \Rightarrow\ \Gamma^\star \triangleright e' :: \tau$

(b) $\forall \Gamma \in \mathit{TEnv} : \forall r, r' \in \mathit{Row} : \forall \phi \in \mathit{RType} : \Gamma \triangleright r :: \phi \wedge r \to r' \ \Rightarrow\ \Gamma \triangleright r' :: \phi$

Beweis: Für die Programmiersprache $\mathcal{L}_o^t$ wurde die Typerhaltung durch simultane Induktion über die Länge der Herleitung der small steps und Fallunterscheidung nach der zuletzt angewandten small step Regel bewiesen. Damit war der Beweis sehr einfach, denn für jeden Ausdruck und jede Reihe war jeweils höchstens eine Typregel anwendbar und der Typ – so er denn existierte – eindeutig bestimmt.

Durch Hinzunahme der Subsumptionregel ist diese Eindeutigkeit verloren gegangen. In der Programmiersprache $\mathcal{L}_o^{sub}$ kann auf jede syntaktische Form eines Ausdrucks entweder die Typregel, deren conclusio ebenfalls die syntaktische Form aufweist, oder die (SUBSUME)-Regel angewandt werden. Entsprechend muss die Struktur des Beweises für die Sprache mit Subsumption geändert werden.

Wir führen den Beweis durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma^\star \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi$, mit Fallunterscheidung nach der letzten Typregel in der Herleitung.

1.) Für (CONST), (ID), (ABSTR) und (EMPTY) gilt gemäß Lemma 2.3 $e \not\to$ bzw. $r \not\to$.

2.) $\Gamma^\star \triangleright e :: \tau$ mit (SUBSUME) kann nur aus Prämissen der Form $\Gamma^\star \triangleright e :: \tau'$ und $\tau' \leq \tau$ folgen. Nach Induktionsvoraussetzung gilt dann $\Gamma^\star \triangleright e' :: \tau'$, und mit Typregel (SUBSUME) folgt schliesslich $\Gamma^\star \triangleright e' :: \tau$.

3.) $\Gamma^\star \triangleright e_1\, e_2 :: \tau$ mit Typregel (APP) kann nur aus Prämissen der Form $\Gamma^\star \triangleright e_1 :: \tau_2 \to \tau$ und $\Gamma^\star \triangleright e_2 :: \tau_2$ folgen. Der small step $e_1\, e_2 \to e'$ kann nur mit einer der Regeln (APP-LEFT), (APP-RIGHT), (OP) oder (BETA-V) hergeleitet worden sein.

   1.) Im Fall von (APP-LEFT) gilt $e' = e_1'\, e_2$ und es existiert ein small step $e_1 \to e_1'$. Nach Induktionsvoraussetzung gilt $\Gamma^\star \triangleright e_1' :: \tau_2 \to \tau$ und mit Typregel (APP) folgt daraus $\Gamma^\star \triangleright e_1'\, e_2 :: \tau$.

   2.) $e_1\, e_2 \to e'$ mit (APP-RIGHT) kann nur aus $e_2 \to e_2'$ und $e' = e_1\, e_2'$ folgen. Mit Induktionsvoraussetzung folgt $\Gamma^\star \triangleright e_2' :: \tau_2$ und mit Typregel (APP) somit $\Gamma^\star \triangleright e_1\, e_2' :: \tau$.

   3.) Für $e_1\, e_2 \to e'$ mit Regel (OP) muss $e_1 = op\, n_1$ mit $op \in Op$, $n_1 \in Int$ und $e_2 = n_2 \in Int$ gelten. Weiterhin ist $e' = op^I(n_1, n_2)$. Dann ist zu unterscheiden zwischen arithmetischen und relationalen Operatoren. Hierbei ist zu beachten, dass gemäß Lemma 3.2 Anwendungen der (SUBSUME)-Regel im Folgenden ignoriert werden können.

   Sei also $op \in \{+, -, *\}$, so gilt nach Voraussetzung $\Gamma^\star \triangleright op\, n_1\, n_2 :: \mathbf{int}$ und insbesondere $op :: \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ wegen (AOP). Also ist $op^I(n_1, n_2) \in Int$, und es gilt $\Gamma^\star \triangleright op^I(n_1, n_2) :: \mathbf{int}$.

Sei andererseits $op \in \{<, >, \leq, \geq, =\}$. Es gilt $\Gamma^\star \triangleright op\, n_1\, n_2 :: \mathbf{bool}$ nach Voraussetzung und insbesondere $op :: \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ wegen (ROP). Dann muss also gelten $op^I(n_1, n_2) \in \mathit{Bool}$ und somit $\Gamma^\star \triangleright op^I(n_1, n_2) :: \mathbf{bool}$.

4.) Falls $e_1\, e_2 \to e'$ mit (BETA-V) hergeleitet worden ist, so gilt $e_1 = \lambda x : \tau_2'.e_1'$, $e_2 \in \mathit{Val}$ und $e' = e_1'[{}^{e_2}/_x]$. Nach Lemma 3.8 gilt $\Gamma^\star[{}^{\tau_2'}/_x] \triangleright e_1' :: \tau$ und $\tau_2 \leq \tau_2'$. Aus $\Gamma^\star \triangleright e_2 :: \tau_2$ und $\tau_2 \leq \tau_2'$ folgt mit Typregel (SUBSUME) $\Gamma^\star \triangleright e_2 :: \tau_2'$, und nach Lemma 3.6 gilt dann $\Gamma^\star \triangleright e_1'[{}^{e_2}/_x] :: \tau$ wegen $\Gamma^\star[{}^{\tau_2'}/_x] \triangleright e_1' :: \tau$ und $\Gamma^\star \triangleright v_2 :: \tau_2'$.

4.) Für $\Gamma^\star \triangleright e_1 \# m :: \tau$ mit Typregel (SEND) gilt nach Voraussetzung $\Gamma^\star \triangleright e_1 :: \langle m : \tau; \emptyset \rangle$. Der small step $e_1 \# m \to e'$ kann entweder mit (SEND-EVAL) oder (SEND-UNFOLD) hergeleitet worden sein.

1.) Im Fall von (SEND-EVAL) existiert ein $e_1' \in \mathit{Exp}$ mit $e' = e_1' \# m$ und ein small step $e_1 \to e_1'$. Nach Induktionsvoraussetzung gilt $\Gamma^\star \triangleright e_1' :: \langle m : \tau; \emptyset \rangle$ und mit Typregel (SEND) folgt $\Gamma^\star \triangleright e_1' \# m :: \tau$.

2.) $e_1 \# m \to e'$ mit (SEND-UNFOLD) andererseits bedingt $e_1 = \mathbf{object}\,(\mathit{self} : \tau')\ \omega\ \mathbf{end}$ sowie $e' = \omega[{}^{\mathbf{object}\,(\mathit{self}:\tau')\ \omega\ \mathbf{end}}/_{\mathit{self}}] \# m$, also gilt insbesondere $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ \omega\ \mathbf{end} :: \langle m : \tau; \phi \rangle$.

Nach Lemma 3.8 existiert ein $\phi' \in \mathit{RType}$ mit $(\Gamma^\star)^\star[{}^{\tau'}/_{\mathit{self}}] \triangleright \omega :: \phi'$, $\tau' = \langle \phi' \rangle$ und $\langle \phi' \rangle \leq \langle m : \tau; \phi \rangle$. Mit Typregel (OBJECT) folgt $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau')\ \omega\ \mathbf{end} :: \tau'$ wegen $(\Gamma^\star)^\star[{}^{\tau'}/_{\mathit{self}}] \triangleright \omega :: \phi'$ und $\tau' = \langle \phi' \rangle$.

Aufgrund von $\mathit{dom}\,(\Gamma^\star) \cap \mathit{Attribute} = \emptyset$ folgt somit wegen Lemma 3.7 $\Gamma^\star \triangleright \omega[{}^{\mathbf{object}\,(\mathit{self}:\tau')\ \omega\ \mathbf{end}}/_{\mathit{self}}] :: \phi'$. Wegen $\langle \phi' \rangle \leq \langle m : \tau; \phi \rangle$ existieren nach Lemma 3.3 $\phi'' \in \mathit{RType}$ und $\tau'' \in \mathit{Type}$, so dass gilt $\phi' = (m : \tau''; \phi'')$ mit $\tau'' \leq \tau$. Also folgt mit Typregel (SEND') $\Gamma^\star \triangleright \omega[{}^{\mathbf{object}\,(\mathit{self}:\tau')\ \omega\ \mathbf{end}}/_{\mathit{self}}] \# m :: \tau''$ und wegen $\tau'' \leq \tau$ schliesslich $\Gamma^\star \triangleright \omega[{}^{\mathbf{object}\,(\mathit{self}:\tau')\ \omega\ \mathbf{end}}/_{\mathit{self}}] \# m :: \tau$ mit Typregel (SUBSUME).

5.) $\Gamma^\star \triangleright \omega \# m :: \tau$ mit Typregel (SEND') kann nur aus $\Gamma^\star \triangleright \omega :: \langle m : \tau; \phi \rangle$ folgen. Der small step $\omega \# m \to e'$ wiederum kann nur mit einer der Regeln (SEND-ATTR), (SEND-SKIP) oder (SEND-EXEC) hergeleitet worden sein.

1.) Für $\omega \# m \to e'$ mit (SEND-ATTR) muss gelten $\omega = (\mathbf{val}\ a = v; \omega')$ und $e' = \omega'[{}^{v}/_{a}]$, also $\Gamma^\star \triangleright \mathbf{val}\ a = v; \omega' :: \langle m : \tau; \phi \rangle$. Dieses Typurteil wiederum kann nur mit Typregel (ATTR) aus Prämissen der Form $(\Gamma^\star =)^\star \triangleright v :: \tau'$ und $\Gamma[{}^{\tau'}/_{a}] \triangleright \omega' :: (m : \tau; \phi)$ folgen. Nach Lemma 3.6 folgt $\Gamma^\star \triangleright \omega'[{}^{v}/_{a}] :: (m : \tau; \phi)$ und mit Typregel (SEND') somit $\Gamma^\star \triangleright \omega'[{}^{v}/_{a}] \# m :: \tau$.

Die Beweise für (SEND-SKIP) und (SEND-EXEC) sind ebenfalls identisch zu dem Beweis der Preservation für die Programmiersprache $\mathcal{L}_o^t$.

Die übrigen Fälle verlaufen analog. □

Damit ist sichergestellt, dass die Wohlgetyptheit – und insbesondere der konkrete Typ – auch für die Programmiersprache $\mathcal{L}_o^{sub}$ während eines small steps erhalten bleibt. Für die Typsicherheit bleibt noch zu die Existenz des Übergangsschritts – das sogenannte *Progress-Theorem* – zu zeigen. Dazu beginnen wir, wie bereits beim Beweis des Progress-Theorems für die Programmiersprache $\mathcal{L}_o^t$, mit einem *Canonical Forms Lemma*, welches eine Aussage über die möglichen Formen von Werten bestimmter Typen macht. Wir beschränken uns ähnlich wie bei $\mathcal{L}_o^t$ auf die Aussagen des Lemmas, die wir im nachfolgenden Beweis des Progress-Theorems benutzen werden, um das Lemma kurz zu halten.

Lemma 3.9 (Canonical Forms)

(a) Für alle $v \in Val$ mit $[\,] \triangleright v :: \mathbf{int}$ gilt $v \in Int$.

(b) Für alle $v \in Val$ mit $[\,] \triangleright v :: \tau_1 \to \tau_2$ gilt genau eine der folgenden Aussagen:

1. *$v \in Op$*
2. *$v = op\, v_1$ mit $op \in Op$ und $v_1 \in Val$*
3. *$v = \lambda x : \tau_1'.e$ mit $x \in Var$, $\tau_1' \in Type$ und $e \in Exp$.*

(c) Für alle $v \in Val$ mit $[\,] \triangleright v :: \langle \phi \rangle$ gilt $v = \mathbf{object}\,(self : \langle \phi' \rangle)\ \omega\ \mathbf{end}$.

Beweis: Beide Beweise erfolgen durch Induktion über die Länge der Herleitung des Typurteils und Fallunterscheidung nach der zuletzt angewandten Typregel.

(a) Das Typurteil $[\,] \triangleright v :: \mathbf{int}$ kann ausschliesslich mit einer der Typregeln (CONST) oder (SUBSUME) hergeleitet worden sein, da für Werte vom Typ **int** sonst keine Typregel in Frage kommt.

1.) $[\,] \triangleright v :: \mathbf{int}$ mit Typregel (CONST) kann nur aus $v :: \mathbf{int}$ folgen. Letzteres wiederum kann nur mit (INT) folgen, also gilt $v \in Int$.

2.) Für $[\,] \triangleright v :: \mathbf{int}$ mit (SUBSUME) muss gelten $[\,] \triangleright v :: \tau$ und $\tau \leq \mathbf{int}$. Wegen Lemma 3.2 folgt $\tau = \mathbf{int}$, und mit Induktionsvoraussetzung schliesslich $v \in Int$.

(b) Das Typurteil $[\,] \triangleright v :: \tau_1 \to \tau_2$ kann nur mit den Typregeln (APP), (ABSTR), (CONST) oder (SUBSUME) hergeleitet worden sein.

1.) $[\,] \triangleright v :: \tau_1 \to \tau_2$ mit Typregel (Const) kann nur aus $v \in \mathit{Const}$ und $v :: \tau_1 \to \tau_2$ folgen. Letzteres wiederum kann aber nur mit (Aop) oder (Rop) hergeleitet worden sein, also gilt $v \in \mathit{Op}$.

2.) Falls $[\,] \triangleright v :: \tau_1 \to \tau_2$ mit Typregel (Abstr) hergeleitet worden ist, gilt unmittelbar nach Voraussetzung $v = \lambda x : \tau_1'.e$.

3.) Für $[\,] \triangleright v :: \tau_1 \to \tau_2$ mit Typregel (App) muss gelten $v = e_1\, e_2$, nach Definition 2.4 also $e_1 \in \mathit{Op}$ und $e_2 \in \mathit{Val}$.

4.) $[\,] \triangleright v :: \tau_1 \to \tau_2$ mit Typregel (Subsume) kann nur aus Prämissen der Form $[\,] \triangleright v :: \tau'$ und $\tau' \leq \tau_1 \to \tau_2$ folgen. Nach Lemma 3.3 existieren $\tau_1', \tau_2' \in \mathit{Type}$ mit $\tau' = \tau_1' \to \tau_2'$, $\tau_1 \leq \tau_1'$ und $\tau_2' \leq \tau_2$. Aus $[\,] \triangleright v :: \tau_1' \to \tau_2'$ folgt dann mit Induktionsvoraussetzung, dass v eine der im Lemma aufgeführten Formen hat.

(c) In diesem Fall ist klar, dass das Typurteil $[\,] \triangleright v :: \langle\phi\rangle$ nur mit den Typregeln (Object) oder (Subsume) hergeleitet worden sein kann.

1.) Für $[\,] \triangleright v :: \langle\phi\rangle$ mit Typregel (Object) gilt bereits nach Voraussetzung $v = \mathbf{object}\,(\mathit{self} : \langle\phi\rangle)\ \omega\ \mathbf{end}$.

2.) $[\,] \triangleright v :: \langle\phi\rangle$ mit Typregel (Subsume) kann nur aus Prämissen der Form $[\,] \triangleright v :: \tau$ und $\tau \leq \langle\phi\rangle$ folgen. Gemäß Lemma 3.3 ist τ ebenfalls ein Objekttyp, und mit Induktionsvoraussetzung folgt die Aussage. □

Satz 3.2 (Existenz des Übergangsschritts, „Progress“)

(a) $\forall e \in \mathit{Exp}, \tau \in \mathit{Type} : [\,] \triangleright e :: \tau \Rightarrow (e \in \mathit{Val} \vee \exists e' \in \mathit{Exp} : e \to e')$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma^+ \triangleright r :: \phi \Rightarrow (r \in \mathit{RVal} \vee \exists r' \in \mathit{Row} : r \to r')$

Beweis: Wie bereits für die Programmiersprache $\mathcal{L}_o^t$ erfolgt der Beweis des Progress-Theorems durch simultane Induktion über die Länge der Herleitung der Typurteile $[\,] \triangleright e :: \tau$ und $\Gamma^+ \triangleright r :: \phi$, allerdings mit Fallunterscheidung nach der zuletzt angewandten Typregel statt Fallunterscheidung nach der syntaktischen Form von e und r. Der Beweis ist im wesentlichen gleich, da das Canonical Forms Lemma bereits die meisten durch Subsumption hinzukommenden Sonderfälle abdeckt.

1.) Für (Const) und (Abstr) gilt $e \in \mathit{Val}$, und für (Empty) gilt $r \in \mathit{RVal}$.

2.) $[\,] \triangleright e :: \tau$ mit (Subsume) kann ausschliesslich aus $[\,] \triangleright e :: \tau'$ mit $\tau' \leq \tau$ folgen. Die Aussage folgt dann mit Induktionsvoraussetzung aus $[\,] \triangleright e :: \tau'$.

3.) Das Typurteil $[\,] \triangleright e_1\, e_2 :: \tau$ mit Typregel (App) bedingt $[\,] \triangleright e_1 :: \tau_2 \to \tau$ und $[\,] \triangleright e_2 :: \tau_2$. Weiterhin ist nach der Art der Teilausdrücke e_1, e_2 zu unterscheiden.

Wenn $e_1 \notin \mathit{Val}$, dann existiert nach Induktionsvoraussetzung ein $e_1' \in \mathit{Exp}$ mit $e_1 \to e_1'$. Also existiert für $e_1\, e_2$ ein small step $e_1\, e_2 \to e_1'\, e_2$ mit Regel (App-Left).

Entsprechendes gilt für den Fall $e_1 \in \mathit{Val}$ und $e_2 \notin \mathit{Val}$. Dann existiert ein small step mit Regel (App-Right).

Es bleibt noch der Fall $e_1, e_2 \in \mathit{Val}$ zu betrachten. Wegen $[\,] \triangleright e_1 :: \tau_2 \to \tau$ sind gemäß dem Canonical Forms Lemma (3.9) für e_1 drei Fälle zu unterscheiden.

1.) Wenn $e_1 \in \mathit{Op}$, dann gilt nach Definition 2.4 $e_1\, e_2 \in \mathit{Val}$.

2.) Ist $e_1 = \mathit{op}\, v_1$ für $\mathit{op} \in \mathit{Op}$ und $v_1 \in \mathit{Val}$, so folgt aus $[\,] \triangleright \mathit{op}\, v_1 :: \tau_2 \to \tau$ mit Lemma 3.8 $[\,] \triangleright \mathit{op} :: \tau_1' \to \tau_2 \to \tau$, $[\,] \triangleright v_1 :: \tau_1$ und $\tau_1 \leq \tau_1'$. Ebenfalls mit Lemma 3.8 folgt aus $[\,] \triangleright \mathit{op} :: \tau_1' \to \tau_2 \to \tau$, dass $\tau_1' = \mathbf{int}$ und $\tau_2 = \mathbf{int}$ gelten muss. Nach Lemma 3.2 somit $\tau_1 = \mathbf{int}$, das heißt $[\,] \triangleright v_1 :: \mathbf{int}$ und $[\,] \triangleright e_2 :: \mathbf{int}$. Daraus folgt mit Lemma 3.9 $v_1, e_2 \in \mathit{Int}$, also existiert ein small step $\mathit{op}\, v_1\, e_2 \to \mathit{op}^I(v_1, e_2)$ mit Regel (Op).

3.) Für $e_1 = \lambda x : \tau_2'.e_1'$ existiert ein small step $(\lambda x : \tau_2'.e_1')\, e_2 \to e_1'[{}^{e_2}/_{x}]$ mit Regel (Beta-V), da nach Voraussetzung $e_2 \in \mathit{Val}$.

4.) $[\,] \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ mit Typregel (Object) kann ausschliesslich mit $[\mathit{self} : \tau] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen. Nach Induktionsvoraussetzung ist $r \in \mathit{RVal}$ oder es existiert ein $r' \in \mathit{Row}$ mit $r \to r'$. Falls also $r \in \mathit{RVal}$, so gilt $(\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}) \in \mathit{Val}$. Anderenfalls, für $r \to r'$, existiert ein small step $\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} \to \mathbf{object}\,(\mathit{self} : \tau)\ r'\ \mathbf{end}$ mit Regel (Object-Eval).

5.) Für $[\,] \triangleright e_1 \# m :: \tau$ mit Typregel (Send) muss gelten $[\,] \triangleright e_1 :: \langle m : \tau;\, \phi \rangle$, und nach Induktionsvoraussetzung gilt entweder $e_1 \in \mathit{Val}$ oder es existiert ein $e_1' \in \mathit{Exp}$ mit $e_1 \to e_1'$.

Falls $e_1 \in \mathit{Val}$, so gilt nach Lemma 3.9 $e_1 = \mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end}$ und es existiert ein small step $\mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end} \# m \to \omega[{}^{e_1}/_{\mathit{self}}] \# m$ mit Regel (Send-Unfold).

Anderenfalls existiert ein small step $e_1 \# m \to e_1' \# m$ mit Regel (Send-Eval).

Die restlichen Fälle verlaufen ähnlich. □

Satz 3.3 (Typsicherheit, „Safety“) *Wenn $[\,] \triangleright e :: \tau$, dann bleibt die Berechnung für e nicht stecken.*

Beweis: Folgt analog wie für $\mathcal{L}_o^t$ aus den Preservation und Progress-Sätzen. □

3.4 Minimal Typing

Nach dem Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_o^{sub}$ im vorangegangenen Abschnitt wollen wir nun auf die Frage der Typeindeutigkeit zurückkommen. Wir haben bereits gesehen, dass das Typsystem nach Hinzunahme der Subsumption-Regel nicht mehr deterministisch ist. Während für die Sprache $\mathcal{L}_o^t$ ein einfacher Algorithmus zur Typüberprüfung existiert, der durch Rückwärtsanwendung von Typregeln versucht einen Typ herzuleiten, ist dies für die Sprache $\mathcal{L}_o^{sub}$ nicht mehr möglich.

Neben der (Subsume)-Regel existiert noch ein weiteres Hinderniss für eine einfache algorithmische Umsetzung: Das Regelwerk für die Subtyprelation ist ebenfalls nicht deterministisch, denn auf jede Formel der Gestalt $\tau \leq \tau'$ ist stets die (S-Trans)-Regel und häufig zumindest eine weitere Regel anwendbar.

In diesem Abschnitt wollen wir versuchen ein deterministisches Regelwerk für die Typüberprüfung einer Sprache mit Subtyping anzugeben, welches äquivalent zum Typsystem der Sprache $\mathcal{L}_o^{sub}$ ist.

3.4.1 Die Subtyprelation $\leq_m$

Nach dem Subtyping-Lemma (3.3) enthält die Subtyprelation $\leq$ nur Paare von Typen, deren syntaktische Form übereinstimmt. Das bedeutet, ein Funktionstyp kann nur in Relation zu einem anderen Funktionstyp stehen, aber niemals zu einem Objekttyp oder einem primitiven Typ. Im Folgenden definieren wir basierend auf dieser Überlegung die Relation $\leq_m$ und zeigen anschliessend, dass die Relationen $\leq$ und $\leq_m$ übereinstimmen (vgl. dazu Definition 3.1 für die Relation $\leq$).

Definition 3.3 (Subtyping-Regeln fürs Minimal Typing) Die Subtyprelation $\leq_m$ ist die kleinste Relation auf $\mathit{Type} \times \mathit{Type}$, die sich mit den folgenden Regeln herleiten läßt.

(Sm-Refl) $\quad \tau_\beta \leq_m \tau_\beta$ für alle $\tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

(Sm-Arrow) $\quad \dfrac{\tau_1' \leq_m \tau_1 \quad \tau_2 \leq_m \tau_2'}{\tau_1 \to \tau_2 \leq_m \tau_1' \to \tau_2'}$

(Sm-Object) $\quad \dfrac{\tau_i \leq_m \tau_j' \text{ für alle } i,j \text{ mit } m_i = m_j'}{\langle m_1 : \tau_1; \ldots; m_k : \tau_k\rangle \leq_m \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l'\rangle}$
falls $\{m_1', \ldots, m_l'\} \subseteq \{m_1, \ldots, m_k\}$

Intutiv ist klar, dass gemäß Lemma 3.3 die Relationen $\leq$ und $\leq_m$ identisch sind. Der wesentliche Unterschied besteht in der Regel (Sm-Refl), die nun auf primitive Typen

eingeschränkt worden ist, und in der Regel (Sm-Object), die eine Kombination der vorherigen Regeln (S-Trans), (S-Obj-Depth) und (S-Obj-Width) darstellt.

Um den Beweis der Gleichheit der beiden Relationen $\leq$ und $\leq_m$ zu erleichtern, zeigen wir zunächst, dass die Relation $\leq_m$ transitiv ist. Dazu das folgende Lemma.

Lemma 3.10 $\forall \tau_1, \tau_2, \tau_3 \in \mathit{Type} : \tau_1 \leq_m \tau_2 \ \wedge \ \tau_2 \leq_m \tau_3 \ \Rightarrow \ \tau_1 \leq_m \tau_3$

Beweis: Der Beweis erfolgt durch simultane Induktion über die Länge der Herleitungen von $\tau_1 \leq_m \tau_2$ und $\tau_2 \leq_m \tau_3$ und Fallunterscheidung nach der syntaktischen Form von τ_2.

1.) $\tau_2 \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$ bedingt, dass $\tau_1 \leq_m \tau_2$ und $\tau_2 \leq_m \tau_3$ mit Regel (Sm-Refl) folgen müssen, also $\tau_1 = \tau_2 = \tau_3$. Daraus wiederum folgt unmittelbar $\tau_1 \leq_m \tau_3$ mit Regel (Sm-Refl).

2.) Für $\tau_2 = \tau_2' \to \tau_2''$ können $\tau_1 \leq_m \tau_2$ und $\tau_2 \leq_m \tau_3$ nur Regel (Sm-Arrow) hergeleitet worden sein. Also ist $\tau_1 = \tau_1' \to \tau_1''$ und $\tau_3 = \tau_3' \to \tau_3''$, und es gilt $\tau_2' \leq_m \tau_1'$, $\tau_1'' \leq_m \tau_2''$, $\tau_3' \leq_m \tau_2'$ und $\tau_2'' \leq_m \tau_3''$. Nach Induktionsvoraussetzung folgt daraus $\tau_3' \leq_m \tau_1'$ und $\tau_1'' \leq_m \tau_3''$, und mit Regel (Sm-Arrow) schliesslich $\tau_1' \to \tau_1'' \leq_m \tau_3' \to \tau_3''$.

3.) Der Fall $\tau_2 = \langle \phi \rangle$ verläuft entsprechend, wobei dann $\tau_1 \leq_m \tau_2$ und $\tau_2 \leq_m \tau_3$ nur mit Regel (Sm-Object) folgen können. □

Mit diesem Ergebnis können wir nun den folgenden Satz vergleichsweise einfach beweisen.

Satz 3.4 $\forall \tau_1, \tau_2 \in \mathit{Type} : \tau_1 \leq_m \tau_2 \ \Leftrightarrow \ \tau_1 \leq \tau_2$

Beweis: Die Äquivalenz zeigen wir in zwei Schritten.

„⇒“ Diesen Teil des Beweises führen wir durch vollständige Induktion über die Länge der Herleitung von $\tau_1 \leq_m \tau_2$ und Fallunterscheidung nach der zuletzt angewandten Regel.

1.) Im Fall von $\tau_1 \leq_m \tau_2$ mit Regel (Sm-Refl) muss gelten $\tau_1 = \tau_2$. Also folgt $\tau_1 \leq \tau_2$ mit (S-Refl).

2.) Für (Sm-Arrow) folgt die Behauptung sofort mit Induktionsvoraussetzung, da die (Sm-Arrow)-Regel identisch ist mit der (S-Arrow)-Regel.

3.) Der (Sm-Object)-Fall folgt ebenso leicht mit Induktionsvoraussetzung und Lemma 3.3.

„⇐“ Diese Richtung beweisen wir entsprechend mittels vollständiger Induktion über die Länge der Herleitung von $\tau_1 \leq \tau_2$ und Fallunterscheidung nach der letzten Regel in der Herleitung.

1.) $\tau_1 \leq \tau_2$ mit (S-Refl) kann nur aus $\tau_1 = \tau_2$ folgen. Dann ist nach der syntaktischen Form von τ_1 zu unterscheiden.

Für $\tau_1 \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$ folgt unmittelbar $\tau_1 \leq_m \tau_2$ mit Regel (Sm-Refl).

Wenn $\tau_1 = \tau_1' \to \tau_1''$, dann gilt $\tau_1' \leq \tau_1'$ und $\tau_1'' \leq \tau_1''$ wegen (S-Refl), mit Induktionsvoraussetzung folgt daraus $\tau_1' \leq_m \tau_1'$ und $\tau_1'' \leq_m \tau_1''$, und mit Regel (Sm-Arrow) schliesslich $\tau_1' \to \tau_1'' \leq_m \tau_1' \to \tau_1''$.

Bleibt noch der Fall $\tau_1 = \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle$ zu betrachten. Wegen (S-Refl) gilt wieder $\tau_1' \leq \tau_1'$ für $i = 1, \ldots, n$ und nach Induktionsvoraussetzung gilt somit $\tau_1' \leq_m \tau_1'$ für $i = 1, \ldots, n$. Mit Regel (Sm-Object) folgt daraus $\langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle \leq_m \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle$, denn trivialerweise gilt $\{m_1, \ldots, m_n\} \subseteq \{m_1, \ldots, m_n\}$.

2.) Für $\tau_1 \leq \tau_2$ mit (S-Arrow) folgt die Behauptung unmittelbar aus der Induktionsvoraussetzung mit Regel (Sm-Arrow).

3.) Wenn $\tau_1 \leq \tau_2$ mit Regel (S-Trans) hergeleitet worden ist, existiert nach Voraussetzung ein τ_3 mit $\tau_1 \leq \tau_3$ und $\tau_3 \leq \tau_2$. Daraus folgt mit Induktionsvoraussetzung $\tau_1 \leq_m \tau_3$ und $\tau_3 \leq_m \tau_2$, und somit $\tau_1 \leq_m \tau_3$ wegen der Transitivität von $\leq_m$ (Lemma 3.10).

4.) Falls andererseits $\tau_1 \leq \tau_2$ mit Regel (S-Obj-Width) hergeleitet wurde, so muss gelten $\tau_1 = \langle m_1 : \tau_1'; \ldots; m_{n+k} : \tau_{n+k}' \rangle$ und $\tau_2 = \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle$. Es gilt also insbesondere $\{m_1, \ldots, m_n\} \subseteq \{m_1, \ldots, m_{n+k}\}$, und weiterhin $\tau_i' \leq \tau_i'$ für $i = 1, \ldots, n$ wegen (S-Refl). Daraus folgt mit Induktionsvoraussetzung $\tau_i' \leq_m \tau_i'$ für $i = 1, \ldots, n$, und schliesslicht mit Regel (Sm-Object) $\langle m_1 : \tau_1'; \ldots; m_{n+k} : \tau_{n+k}' \rangle \leq_m \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle$.

5.) Für $\tau_1 \leq \tau_2$ mit Regel (S-Obj-Depth) folgt die Behauptung ebenso problemlos mit (Sm-Object) aus der Induktionsvoraussetzung. □

Mit diesem Ergebnis brauchen wir nicht länger zwischen $\leq$ und $\leq_m$ zu unterscheiden, und können wahlweise die Regeln aus Definition 3.1 oder Definition 3.3 benutzen, um Aussagen der Gestalt $\tau_1 \leq \tau_2$ herzuleiten.

Basierend auf dem neuen Regelwerk aus Definition 3.3 läßt sich nun ein einfacher Entscheidungsalgorithmus formulieren, der für zwei Typen $\tau_1, \tau_2 \in \mathit{Type}$ überprüft, ob $\tau_1 \leq \tau_2$ gilt.

Algorithmus 3.1 (Entscheidungsalgorithmus für die Subtyprelation) Der Algorithmus erhält als Eingabe zwei Typen $\tau, \tau' \in \mathit{Type}$ und liefert „ja“, falls $\tau < \tau'$ gilt, sonst „nein“.

1.) Wenn $\tau, \tau' \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$ und $\tau = \tau'$, dann Antwort „ja“.

2.) Wenn $\tau = \tau_1 \to \tau_2$ und $\tau' = \tau_1' \to \tau_2'$, dann überprüfe rekursiv, ob $\tau_1' \leq \tau_1$ und $\tau_2 \leq \tau_2$ gelten.

3.) Wenn $\tau = \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle$ und $\tau' = \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle$ ist, wobei $\{m_1', \ldots, m_l'\} \subseteq \{m_1, \ldots, m_k\}$, dann überprüfe rekursiv, ob $\tau_i \leq \tau_j$ für alle i, j mit $m_i = m_j'$ gilt.

4.) In allen anderen Fällen ist die Antwort „nein“.

Es ist offensichtlich, dass der Algorithmus korrekt ist. Ebenso offensichtlich ist, dass der Algorithmus für jede Eingabe terminiert, es sich also tatsächlich um einen Entscheidungsalgorithmus für die Subtyprelation handelt, da ein rekursiver Schritt immer nur mit kleineren Typen durchgeführt wird.

3.4.2 Die Sprache $\mathcal{L}_o^m$

TODO: Prosa

Definition 3.4 (Suprema und Infima)

(a) Das Supremum $\tau \vee \tau'$ zweier Typen $\tau, \tau' \in \mathit{Type}$ ist wie folgt induktiv definiert:

$$\begin{array}{rcl} \tau_\beta \vee \tau_\beta & = & \tau_\beta \text{ für alle } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ \tau_1 \to \tau_2 \vee \tau_1' \to \tau_2' & = & (\tau_1 \wedge \tau_1') \to (\tau_2 \vee \tau_2') \\ \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle & & \\ \vee \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle & = & \langle m_1'' : \tau_1''; \ldots; m_n'' : \tau_m'' \rangle \\ & & \text{mit } \{m_1'', \ldots, m_n''\} = \{m_1, \ldots, m_k\} \cap \{m_1', \ldots, m_l'\} \\ & & \text{und } \tau_h'' = \tau_i \vee \tau_i' \text{ für } m_i = m_h'' = m_j' \end{array}$$

(b) Das Infimum $\tau \wedge \tau'$ zweier Typen $\tau, \tau' \in \mathit{Type}$ ist induktiv definiert durch:

$$\begin{array}{rcl} \tau_\beta \wedge \tau_\beta & = & \tau_\beta \text{ für alle } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ \tau_1 \to \tau_2 \wedge \tau_1' \to \tau_2' & = & (\tau_1 \vee \tau_1') \to (\tau_2 \wedge \tau_2') \\ \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle & & \\ \wedge \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle & = & \langle m_1'' : \tau_1''; \ldots; m_n'' : \tau_m'' \rangle \\ & & \text{mit } \{m_1'', \ldots, m_n''\} = \{m_1, \ldots, m_k\} \cup \{m_1', \ldots, m_l'\} \\ & & \text{und } \tau_h'' = \tau_i \wedge \tau_i' \text{ für } m_i = m_h'' = m_j' \end{array}$$

TODO: Prosa

Lemma 3.11 *Seien $\tau_1, \tau_2 \in Type$.*

(a) Wenn $\tau_1 \vee \tau_2$ existiert, dann gilt $\tau_1 \leq \tau_1 \vee \tau_2$ und $\tau_2 \leq \tau_1 \vee \tau_2$.

(b) Wenn $\tau_1 \wedge \tau_2$ existiert, dann gilt $\tau_1 \wedge \tau_2 \leq \tau_1$ und $\tau_1 \wedge \tau_2 \leq \tau_2$.

Beweis: Folgt unmittelbar aus Definition 3.4. □

Weiterhin benötigen wir noch kleines Lemma, welches die Existenz eines Supremums, also eines kleinsten gemeinsamen Supertyps, für zwei kompatible Typen sichert, die beide ihrerseits Subtyp eines bekannten Typs sind.

Lemma 3.12 *Seien $\tau, \tau_1, \tau_2 \in Type$. Wenn $\tau_1 \leq \tau$ und $\tau_2 \leq \tau$, dann existiert ein Typ $\tau_1 \vee \tau_2$ mit $\tau_1 \vee \tau_2 \leq \tau$.*

Beweis: Trivial. □

Wir geben nun Typregeln für das Minimal Typing an, mit denen sich gültige Typurteile $\Gamma \triangleright_m e :: \tau$ und $\Gamma \triangleright_m r :: \phi$ herleiten lassen. Allerdings beschränken wir uns dabei auf Typherleitungen für Ausdrücke, die der Programmierer benutzen kann, d.h. insbesondere es wird keine Typherleitung für Ausdrücke der Form $\omega \# m$ geben.

Die Idee hierbei ist wie folgt: Typurteile für diese speziellen Ausdrücke werden lediglich benötigt zum Beweis der Typsicherheit mit Hilfe des Kalküls. Für das Minimal Typing werden wir aber Typsicherheit nicht explizit zeigen, sondern lediglich beweisen das der Kalkül korrekt ist bezüglich des Typsystems von $\mathcal{L}_o^{sub}$, und somit die Typsicherheit der Sprache $\mathcal{L}_o^m$ aus der Typsicherheit der Sprache $\mathcal{L}_o^{sub}$ folgt. Das bedeutet wir können uns beim Minimal Typing auf die Ausdrücke beschränken, die der Programmierer im Quelltext notieren kann.

Basierend auf diesen Überlegungen können wir nun die Regeln für den Minimal Typing Kalkül definieren. Dazu übernehmen wir im wesentlichen die Typregeln der Programmiersprache $\mathcal{L}_o^{sub}$, mit Ausnahme der (Subsume)-Regel, und erweitern die Regeln (App), (Rec), (Cond), (Dupl) und (Method) um Subsumption.

Definition 3.5 (Gültige Typurteile für $\mathcal{L}_o^m$) Ein Typurteil $\Gamma \triangleright_m e :: \tau$ oder $\Gamma \triangleright_m r :: \phi$ heisst gültig für $\mathcal{L}_o^m$, wenn es sich mit den folgenden Typregeln für die funktionale Kernsprache

(Id) $\Gamma \rhd_m id :: \tau \quad \text{falls } id \in dom(\Gamma) \wedge \Gamma(id) = \tau$

(Const) $$\frac{c \;::\; \tau}{\Gamma \rhd_m c :: \tau}$$

(App-Subsume) $$\frac{\Gamma \rhd_m e_1 :: \tau_2' \to \tau \quad \Gamma \rhd e_2 :: \tau_2 \quad \tau_2 \leq \tau_2'}{\Gamma \rhd_m e_1\, e_2 :: \tau}$$

(Abstr) $$\frac{\Gamma[^{\tau}/_{x}] \rhd_m e :: \tau'}{\Gamma \rhd_m \lambda x : \tau.e :: \tau \to \tau'}$$

(Rec-Subsume) $$\frac{\Gamma[^{\tau}/_{x}] \rhd_m e :: \tau' \quad \tau' \leq \tau}{\Gamma \rhd_m \mathbf{rec}\, x : \tau.e :: \tau}$$

(Let) $$\frac{\Gamma \rhd_m e_1 :: \tau_1 \quad \Gamma[^{\tau_1}/_{x}] \rhd_m e_2 :: \tau_2}{\Gamma \rhd_m \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 :: \tau_2}$$

(Cond-Subsume) $$\frac{\Gamma \rhd_m e_0 :: \mathbf{bool} \quad \Gamma \rhd_m e_1 :: \tau_1 \quad \Gamma \rhd_m e_2 :: \tau_2}{\Gamma \rhd_m \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau_1 \vee \tau_2}$$

den folgenden Typregeln für Objekte

(Send) $$\frac{\Gamma \rhd_m e :: \langle m : \tau; \phi \rangle}{\Gamma \rhd_m e\#m :: \tau}$$

(Object) $$\frac{\Gamma^{\star}[^{\tau}/_{self}] \rhd_m r :: \phi \quad \tau = \langle \phi \rangle}{\Gamma \rhd_m \mathbf{object}\,(self : \tau)\; r\; \mathbf{end} :: \tau}$$

(Dupl-Subsume) $$\frac{\Gamma \rhd_m self :: \tau \quad \forall i = 1 \ldots n : \Gamma \rhd_m a_i :: \tau_i \wedge \Gamma \rhd_m e_i :: \tau_i' \wedge \tau_i' \leq \tau_i}{\Gamma \rhd_m \{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\} :: \tau}$$

sowie den folgenden Typregeln für Reihen

(Empty) $\Gamma \rhd_m \epsilon :: \emptyset$

(Attr) $$\frac{\Gamma^{\star} \rhd_m e :: \tau \quad \Gamma[^{\tau}/_{a}] \rhd_m r_1 :: \phi}{\Gamma \rhd_m \mathbf{val}\; a = e;\, r_1 :: \phi}$$

(Method-Subsume) $$\frac{\Gamma \rhd_m self :: \langle m : \tau; \phi' \rangle \quad \Gamma \rhd_m e :: \tau' \quad \Gamma \rhd_m r_1 :: \phi \quad \tau' \leq \tau}{\Gamma \rhd_m \mathbf{method}\; m = e;\, r_1 :: (m : \tau; \emptyset) \oplus \phi}$$

herleiten läßt.

TODO: Mehr Prosa?

Wir wollen nun zeigen, dass der Minimal Typing Kalkül aus Definition 3.5 korrekt ist, also dass die Typrelation der Programmiersprache $\mathcal{L}_o^m$ eine Teilmenge der Typrelation von $\mathcal{L}_o^{sub}$ ist. Anders ausgedrückt bedeutet das: Wenn sich mit den Regeln von $\mathcal{L}_o^m$ für einen Ausdruck e in einer Typumgebung Γ ein Typ τ herleiten läßt, so läßt sich auch mit den Regeln von $\mathcal{L}_o^{sub}$ der gleiche Typ τ herleiten (und entsprechend für Reihen).

Satz 3.5 (Korrektheit des Minimal Typing)

(a) $\forall \Gamma \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma \rhd_m e :: \tau \;\Rightarrow\; \Gamma \rhd e :: \tau$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma \rhd_m r :: \phi \;\Rightarrow\; \Gamma \rhd r :: \phi$

Intuitiv sollte im wesentlichen schon klar sein, dass dieser Satz gilt. Wir betrachten daher im folgenden Beweis lediglich die Herleitungen, die mit der Anwendung einer der fünf geänderten Regeln enden.

Beweis: Wie üblich führen wir den Beweis durch simultane Induktion über die Länge der Herleitungen der Typurteile $\Gamma \rhd_m e :: \tau$ und $\Gamma \rhd_m r :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel (aus dem Minimal Typing Kalkül). Wie bereits erwähnt betrachten wir lediglich die interessanten Fälle.

1.) $\Gamma \rhd_m e_1\, e_2 :: \tau$ mit Typregel (App-Subsume) kann nur aus Prämissen der Form $\Gamma \rhd_m e_1 :: \tau_2' \to \tau$, $\Gamma \rhd_m e_2 :: \tau_2$ und $\tau_2 \leq \tau_2'$ folgen. Nach Induktionsvoraussetzung gilt also $\Gamma \rhd e_1 :: \tau_2' \to \tau$ und $\Gamma \rhd e_2 :: \tau_2$, woraus sich das Typurteil $\Gamma \rhd e_1\, e_2 :: \tau$ wie folgt herleiten läßt:

$$\dfrac{\Gamma \rhd e_1 :: \tau_2' \to \tau \qquad \dfrac{\Gamma \rhd e_2 :: \tau_2 \qquad \tau_2 \leq \tau_2'}{\Gamma \rhd e_2 :: \tau_2'}\ \textsc{Subsume}}{\Gamma \rhd e_1\, e_2 :: \tau}\ \textsc{App}$$

2.) $\Gamma \rhd_m \mathbf{rec}\, x : \tau.e :: \tau$ mit Typregel (Rec-Subsume) bedingt $\Gamma[^{\tau}/_{x}] \rhd_m e :: \tau'$ und $\tau' \leq \tau$. Mit Induktionsvoraussetzung folgt daraus $\Gamma[^{\tau}/_{x}] \rhd e :: \tau'$, und die Behauptung läßt sich folgendermaßen herleiten:

$$\dfrac{\dfrac{\Gamma[^{\tau}/_{x}] \rhd e :: \tau' \qquad \tau' \leq \tau}{\Gamma[^{\tau}/_{x}] \rhd e :: \tau}\ \textsc{Subsume}}{\Gamma \rhd \mathbf{rec}\, x : \tau.e :: \tau}\ \textsc{Rec}$$

3.) Für $\Gamma \rhd_m \textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 :: \tau_1 \vee \tau_2$ gilt nach Voraussetzung $\Gamma \rhd_m e_0 :: \textbf{bool}$, $\Gamma \rhd_m e_1 :: \tau_1$ und $\Gamma \rhd_m e_2 :: \tau_2$. Sei $\tau = \tau_1 \vee \tau_2$, dann gilt wegen Lemma 3.11 $\tau_1 \leq \tau$ und $\tau_2 \leq \tau$, und wegen Induktionsvoraussetzung gilt $\Gamma \rhd e_0 :: \textbf{bool}$, $\Gamma \rhd e_1 :: \tau_1$ und $\Gamma \rhd e_2 :: \tau_2$. Damit läßt sich die Behauptung wie folgt herleiten:

$$\dfrac{\Gamma \rhd e_0 :: \textbf{bool} \qquad \dfrac{\dfrac{\Gamma \rhd e_1 :: \tau_1 \qquad \tau_1 \leq \tau}{\Gamma \rhd e_1 :: \tau}\,\textsc{Subsume} \qquad \dfrac{\Gamma \rhd e_2 :: \tau_2 \qquad \tau_2 \leq \tau}{\Gamma \rhd e_2 :: \tau}\,\textsc{Subsume}}{\Gamma \rhd \textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 :: \tau}\,\textsc{Cond}$$

4.) $\Gamma \rhd_m \{\langle a_1 = e_1; \ldots; a_n = e_n\rangle\} :: \tau$ mit Typregel (Dupl-Subsume) kann ausschliesslich aus Prämissen der Form

$$\Gamma \rhd_m \mathit{self} :: \tau$$

sowie

$$\forall i = 1, \ldots, n : \Gamma \rhd_m a_i :: \tau_i \ \wedge\ \Gamma \rhd_m e_i :: \tau_i' \ \wedge\ \tau_i' \leq \tau_i$$

folgen. Nach Induktionsvoraussetzung gilt

$$\Gamma \rhd \mathit{self} :: \tau$$

und

$$\forall i = 1, \ldots, n : \Gamma \rhd a_i :: \tau_i \ \wedge\ \Gamma \rhd e_i :: \tau_i' \ \wedge\ \tau_i' \leq \tau_i,$$

und mit Typregel (Subsume) folgt

$$\forall i = 1, \ldots, n : \Gamma \rhd a_i :: \tau_i \ \wedge\ \Gamma \rhd e_i :: \tau_i.$$

Daraus schliesslich folgt $\Gamma \rhd \{\langle a_1 = e_1; \ldots; a_n = e_n\rangle\} :: \tau$ mit Typregel (Dupl).

5.) Im Fall von $\Gamma \rhd_m \textbf{method}\ m = e_1;\ r_1 :: \phi$ mit Typregel (Method-Subsume) gilt nach Voraussetzung $\Gamma \rhd_m \mathit{self} :: \langle m : \tau;\ \phi''\rangle$, $\Gamma \rhd_m e_1 :: \tau'$, $\Gamma \rhd_m r_1 :: \phi'$ und $\tau' \leq \tau$, wobei $\phi = (m : \tau; \emptyset) \oplus \phi'$. Nach Induktionsvoraussetzung folgt somit $\Gamma \rhd e_1 :: \tau'$ und $\Gamma \rhd r_1 :: \phi'$. Wegen Typregel (Subsume) gilt ebenfalls $\Gamma \rhd e_1 :: \tau$, und daraus schliesslich folgt mit (Method) $\Gamma \rhd \textbf{method}\ m = e_1;\ r_1 :: \phi$.

Die übrigen Fälle folgen trivialerweise aus der Übereinstimmung der Regeln (teilweise mit Induktionsvoraussetzung). □

Wie bereits angedeutet, folgt aus der Korrektheit des Minimal Typing Kalküls die Typsicherheit der Programmiersprache $\mathcal{L}_o^m$.

Satz 3.6 (Typsicherheit, „Safety“) *Wenn* $[\,] \triangleright_m e :: \tau$*, dann bleibt die Berechnung für* e *nicht stecken.*

Beweis: Folgt mit Satz 3.5 unmittelbar aus der Typsicherheit von $\mathcal{L}_o^{sub}$ (Satz 3.3), denn es gilt

$$\begin{array}{ll} & [\,] \triangleright_m e :: \tau \\ \overset{\text{Satz 3.5}}{\Longrightarrow} & [\,] \triangleright e :: \tau \\ \overset{\text{Satz 3.3}}{\Longrightarrow} & \text{die Berechnung von } e \text{ bleibt nicht stecken.} \end{array}$$ □

TODO: Überleitung zur Vollständigkeit

Definition 3.6 $Exp_P := Exp \setminus \{\omega \# m \mid \omega \in Val, m \in Method\}$

Exp_P enthält genau die Ausdrücke, die der Programmierer im Quelltext von Programmen notieren kann.
TODO: Diese Definition ist Unfug, da nach dieser Definition $\omega \# m$ immer noch innerhalb von Ausdrücken vorkommen kann.

TODO: Insbesondere kann also eine Reihe niemals ausserhalb eines Objekts erscheinen, da dies sonst nur durch Ausdrücke der Form $\omega \# m$ möglich war. Entsprechend müssen Typumgebungen für Reihen in gültigen Typurteilen stets einen Typ für *self* enthalten. Dazu definieren wir eine Menge $TEnv_\phi \subseteq TEnv$, die alle Typumgebungen enthält, in denen ein Typ für *self* eingetragen ist, der länger ist als $\langle\phi\rangle$.

Definition 3.7 $TEnv_\phi := \{\Gamma \in TEnv \mid \exists \phi', \phi'' \in RType : \Gamma(self) = \langle\phi'\rangle \wedge \phi' = \phi \oplus \phi''\}$

In Worten bedeutet $\phi' = \phi \oplus \phi''$, dass ϕ' eine Verlängerung von ϕ ist, also dass ϕ' mindestens alle Methodennamen von ϕ enthält, und ϕ' und ϕ auf den gemeinsamen Methodentypen übereinstimmen.

Definition 3.8 (Typumgebungen und Subtyping) Seien $\Gamma, \Gamma' \in TEnv$. Dann definieren eine Subtyprelation auf $TEnv$ wie folgt komponentenweise:

$$\Gamma' \leq \Gamma \quad :\Leftrightarrow \quad dom\,(\Gamma) = dom\,(\Gamma') \wedge \forall id \in dom\,(\Gamma) : \Gamma(id) \leq \Gamma'(id)$$

TODO: Bla blub

Lemma 3.13 *Seien* $\Gamma_1, \Gamma_2 \in \mathit{TEnv}$. *Dann gilt:*

(a) $\Gamma_1 \leq \Gamma_2 \;\Rightarrow\; \Gamma_1^\star \leq \Gamma_2^\star$

(b) $\forall \tau \in \mathit{Type} : \forall id \in \mathit{Id} : \Gamma_1 \leq \Gamma_2 \;\Rightarrow\; \Gamma_1[^\tau/_{id}] \leq \Gamma_2[^\tau/_{id}]$

(c) $\forall \tau_1, \tau_2 \in \mathit{Type} : \forall id \in \mathit{Id} : \Gamma_1 \leq \Gamma_2 \wedge \tau_1 \leq \tau_2 \;\Rightarrow\; \Gamma_1[^{\tau_1}/_{id}] \leq \Gamma_2[^{\tau_2}/_{id}]$

Beweis:

(a) Folgt trivialerweise, denn es gilt

$$\mathit{dom}\,(\Gamma_1^\star) = \mathit{dom}\,(\Gamma_1) \cap \mathit{Var} = \mathit{dom}\,(\Gamma_2) \cap \mathit{Var} = \mathit{dom}\,(\Gamma_2^\star)$$

und

$$\Gamma_1^\star(id) = \Gamma_1(id) \leq \Gamma_2(id) = \Gamma_2^\star(id)$$

für alle $id \in \mathit{dom}\,(\Gamma_1^\star)$.

(b) Ist lediglich ein Spezialfall von (c), denn es gilt $\tau \leq \tau$ wegen Subtyping-Regel (S-Refl).

(c) Folgt ebenso einfach, denn es gilt

$$\mathit{dom}\,(\Gamma_1[^{\tau_1}/_{id}]) = \mathit{dom}\,(\Gamma_1) \cup \{id\} = \mathit{dom}\,(\Gamma_2) \cup \{id\} = \mathit{dom}\,(\Gamma_2[^{\tau_1}/_{id}])$$

und

$$\Gamma_1[^{\tau_1}/_{id}](id) = \tau_1 \leq \tau_2 = \Gamma_2[^{\tau_2}/_{id}](id),$$

sowie

$$\forall id' \in \mathit{dom}\,(\Gamma_1) : \Gamma_1(id') \leq \Gamma_2(id'),$$

nach Voraussetzung, zusammenfassend also gilt $\Gamma_1[^{\tau_1}/_{id}] \leq \Gamma_2[^{\tau_2}/_{id}]$. □

TODO: Prosa

Satz 3.7 (Vollständigkeit des Minimal Typing)

(a) $\forall \tau \in \mathit{Type} : \forall \Gamma, \Gamma' \in \mathit{TEnv} : \forall e \in \mathit{Exp}_P :$
$\Gamma \rhd e :: \tau \wedge \Gamma' \leq \Gamma \wedge \Gamma' =_{dom(\Gamma) \cap \mathit{Attribute}} \Gamma \;\Rightarrow\; \exists \tau' \in \mathit{Type} : \Gamma' \rhd_m e :: \tau' \wedge \tau' \leq \tau$

(b) $\forall \phi \in \mathit{RType} : \forall \Gamma, \Gamma' \in \mathit{TEnv}_\phi : \forall r \in \mathit{Row} :$
$\Gamma \rhd r :: \phi \wedge \Gamma' \leq \Gamma \;\Rightarrow\; \Gamma' \rhd_m r :: \phi$

Die Voraussetzungen scheinen auf den ersten Blick etwas viele zu sein. Wir werden jedoch im Verlauf des Beweises sehen, warum jede einzelne dieser Voraussetzungen notwendig ist. Die Beschränkung auf Ausdrücke in Exp_P, also Ausdrücke, die nicht von der Form $\omega \# m$ sind, sollte sofort ersichtlich sein, da diese nur während der Auswertung eines Programms entstehen können und dem Programmierer nicht zur Verfügung stehen, und folglich nicht durch den Minimal Typing Kalkül abgedeckt werden. Die Einschränkungen sind auch weniger restriktiv als man zunächst annehmen könnte, wie wir im Anschluss an den Beweis sehen werden.

Beweis: Der Beweis der Vollständigkeit erfolgt durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma \rhd e :: \tau$ und $\Gamma \rhd r :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel (der Programmiersprache $\mathcal{L}_o^{sub}$). Wir betrachten dazu exemplarisch die folgenden Fälle.

1.) $\Gamma \rhd c :: \tau$ mit Typregel (Const) kann nur mit $c :: \tau$ folgen. Also folgt $\Gamma' \rhd_m c :: \tau$ mit Typregel (Const) der Programmiersprache $\mathcal{L}_o^m$.

2.) Im Fall von $\Gamma \rhd \mathit{id} :: \tau$ mit Typregel (Id) gilt nach Voraussetzung $\mathit{id} \in dom(\Gamma)$ und $\Gamma(\mathit{id}) = \tau$. Also gilt nach Definition 3.8 $\mathit{id} \in dom(\Gamma')$ und $\Gamma'(\mathit{id}) = \tau' \leq \tau$. Mit Typregel (Id) aus dem Minimal Typing Kalkül folgt somit $\Gamma' \rhd_m \mathit{id} :: \tau'$.

3.) Für $\Gamma \rhd e :: \tau$ mit Typregel (Subsume) existiert nach Voraussetzung ein $\tau' \in \mathit{Type}$ mit $\tau' \leq \tau$ und $\Gamma \rhd e :: \tau'$. Nach Induktionsvoraussetzung existiert ein $\tau'' \leq \tau'$, so dass $\Gamma' \rhd_m e :: \tau''$ gilt. Wegen (S-Trans) schliesslich folgt $\tau'' \leq \tau$.

4.) $\Gamma \rhd \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 :: \tau$ mit Typregel (Let) erfordert $\Gamma \rhd e_1 :: \tau_1$ und $\Gamma[\tau_1/_x] \rhd e_2 :: \tau$. Nach Induktionsvoraussetzung existiert ein $\tau_1' \in \mathit{Type}$ mit $\tau_1' \leq \tau_1$, so dass

$$\Gamma' \rhd_m e_1 :: \tau_1'$$

gilt. Nach Lemma 3.13 ist $\Gamma'[\tau_1'/_x] \leq \Gamma[\tau_1/_x]$, also existiert wiederum nach Induktionsvoraussetzung ein τ' mit $\tau' \leq \tau$, so dass gilt

$$\Gamma'[\tau_1'/_x] \rhd_m e_2 :: \tau',$$

woraus dann mit Typregel (LET) von $\mathcal{L}_o^m$ schliesslich

$$\Gamma' \rhd_m \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 :: \tau'$$

folgt.

5.) Für $\Gamma \rhd \lambda x : \tau_1.e :: \tau_1 \to \tau_2$ mit Typregel (ABSTR) muss gelten $\Gamma[^{\tau_1}/_x] \rhd e :: \tau_2$ gelten. Nach Lemma 3.13 ist $\Gamma'[^{\tau_1}/_x] \leq \Gamma[^{\tau_1}/_x]$, also existiert nach Induktionsvoraussetzung ein $\tau_2' \in \mathit{Type}$ mit $\tau_2' \leq \tau_2$ und

$$\Gamma'[^{\tau_1}/_x] \rhd_m e :: \tau_2',$$

und mit Typregel (ABSTR) des Minimal Typing Kalküls folgt schliesslich

$$\Gamma' \rhd_m \lambda x : \tau_1.e :: \tau_1 \to \tau_2'.$$

Es bleibt zu zeigen, dass $\tau_1 \to \tau_2' \leq \tau_1 \to \tau_2$ gilt. Dies folgt einfach mit Subtyping-Regeln (S-ARROW), (S-REFL) und der Voraussetzung $\tau_2' \leq \tau_2$.

6.) $\Gamma \rhd \textbf{if}\, e_0 \,\textbf{then}\, e_1 \,\textbf{else}\, e_2 :: \tau$ kann mit Typregel (COND) nur aus Prämissen der Form $\Gamma \rhd e_0 :: \textbf{bool}$, $\Gamma \rhd e_1 :: \tau$ und $\Gamma \rhd e_2 :: \tau$ folgen. Nach Induktionsvoraussetzung gilt also

$$\Gamma' \rhd_m e_0 :: \textbf{bool},$$

denn nach dem Subtyping-Lemma (Lemma 3.3) ist **bool** selbst der einzige Subtyp von **bool**. Weiterhin existiert nach Induktionsvoraussetzung ein $\tau_1 \in \mathit{Type}$ mit $\tau_1 \leq \tau$ und

$$\Gamma' \rhd_m e_1 :: \tau_1,$$

sowie ein $\tau_2 \in \mathit{Type}$ mit $\tau_2 \leq \tau$ und

$$\Gamma' \rhd_m e_2 :: \tau_2.$$

Gemäß Lemma 3.12 ist also das Supremum $\tau_1 \vee \tau_2$ definiert und es gilt $\tau_1 \vee \tau_2 \leq \tau$. Also folgt insgesamt

$$\Gamma' \rhd_m \textbf{if}\, e_0 \,\textbf{then}\, e_1 \,\textbf{else}\, e_2 :: \tau_1 \vee \tau_2$$

mit Typregel (COND-SUBSUME).

7.) $\Gamma \triangleright \mathbf{rec}\, x : \tau.e :: \tau$ mit Typregel (REC) bedingt $\Gamma[^{\tau}/_{x}] \triangleright e :: \tau$. Wegen Lemma 3.13 gilt $\Gamma'[^{\tau}/_{x}] \leq \Gamma[^{\tau}/_{x}]$, also existiert nach Induktionsvoraussetzung ein $\tau' \in \mathit{Type}$ mit $\tau' \leq \tau$ und

$$\Gamma'[^{\tau}/_{x}] \triangleright_m e :: \tau'.$$

Daraus folgt

$$\Gamma' \triangleright_m \mathbf{rec}\, x : \tau.e :: \tau$$

mit Typregel (REC-SUBSUME), und trivialerweise gilt $\tau \leq \tau$.

8.) Das Typurteil $\Gamma \triangleright \{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\} :: \tau$ kann mit Typregel (DUPL) nur aus Prämissen der Form $\Gamma \triangleright \mathit{self} :: \tau$ sowie $\Gamma(a_i) = \tau_i$ und $\Gamma \triangleright e_i :: \tau_i$ für $i = 1, \ldots, n$ folgen. Nach Induktionsvoraussetzung existiert also ein $\tau' \in \mathit{Type}$ mit $\tau' \leq \tau$ und

$$\Gamma' \triangleright_m \mathit{self} :: \tau'.$$

Darüberhinaus existieren $\tau'_1, \ldots, \tau'_n \in \mathit{Type}$ mit $\tau'_i \leq \tau_i$ und

$$\Gamma' \triangleright_m e_i :: \tau'_i$$

für $i = 1, \ldots, n$. Wegen $\Gamma' =_{dom\,(\Gamma) \cap Attribute} \Gamma$ gilt nach wie vor[4]

$$\Gamma'(a_i) = \tau_i$$

für $i = 1, \ldots, n$ und somit folgt

$$\Gamma' \triangleright_m \{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\} :: \tau'$$

mit Typregel (DUPL-SUBSUME) der Programmiersprache $\mathcal{L}_o^m$.

9.) $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ mit Typregel (OBJECT) kann nur aus Prämissen der Form $\Gamma^{\star}[^{\tau}/_{\mathit{self}}] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen. Mit Lemma 3.13 folgt aus $\Gamma \leq \Gamma'$, dass gilt

$$\Gamma^{\star}[^{\tau}/_{\mathit{self}}] \leq \Gamma'^{\star}[^{\tau}/_{\mathit{self}}],$$

[4] An dieser Stelle wird deutlich, warum die Voraussetzung $\Gamma' =_{dom\,(\Gamma) \cap Attribute} \Gamma$ unbedingt notwendig ist.

also folgt nach Induktionsvoraussetzung $\Gamma'^{\star}[^{\tau}/_{self}] \rhd_m r :: \phi$, denn

$$\Gamma'^{\star}[^{\tau}/_{self}](self) = \tau \leq \langle \phi \rangle$$

gilt wegen (S-Refl) und somit ist $\Gamma'^{\star}[^{\tau}/_{self}] \in TEnv_{\phi}$. Daraus schliesslich folgt

$$\Gamma' \rhd_m \textbf{object}\,(self : \tau)\ r\ \textbf{end} :: \tau$$

mit der Typregel (Object) aus dem Minimal Typing Kalkül, und $\tau \leq \tau$ gilt trivialerweise wegen Subtyping-Regel (S-Refl).

10.) Falls das Typurteil $\Gamma \rhd e\#m :: \tau$ mit Typregel (Send) hergeleitet wurde, muss gelten $\Gamma \rhd e :: \langle m : \tau; \phi \rangle$. Nach Induktionsvoraussetzung existiert dann ein $\tau_m \in Type$ mit $\tau_m \leq \langle m : \tau; \phi \rangle$, so dass

$$\Gamma' \rhd_m e :: \tau_m$$

gilt. Gemäß dem Subtyping-Lemma (3.3) existieren dann $\tau' \in Type, \phi' \in RType$ mit $\tau_m = \langle m : \tau'; \phi' \rangle$, $\tau' \leq \tau$ und $\langle \phi' \rangle \leq \langle \phi \rangle$, und es gilt

$$\Gamma' \rhd_m e\#m :: \tau'$$

wegen Typregel (Send) der Programmiersprache $\mathcal{L}_o^m$.

11.) $\Gamma_\phi \rhd \textbf{method}\ m = e'; r' :: \phi$ mit Typregel (Method) bedingt $\Gamma_\phi \rhd e' :: \tau'$ und $\Gamma_\phi \rhd r' :: \phi'$, wobei $\phi = (m : \tau'; \emptyset) \oplus \phi'$. Nach Induktionsvoraussetzung existiert ein $\tau \in Type$ mit $\tau \leq \tau'$, so dass gilt

$$\Gamma'_\phi \rhd_m e' :: \tau$$

und

$$\Gamma'_\phi \rhd_m r' :: \phi',$$

denn ϕ' ist eine Verkürzung von ϕ, also ist $\Gamma_\phi \in TEnv_{\phi'}$. Weiterhin existiert nach Definition 3.7 ein $\phi'' \in RType$ mit $\Gamma'_\phi(self) = \langle (m : \tau'; \emptyset) \oplus \phi' \oplus \phi'' \rangle$, und wegen Typregel (Id) folgt daraus

$$\Gamma'_\phi \rhd_m self :: \langle (m : \tau'; \emptyset) \oplus \phi' \oplus \phi'' \rangle.$$

Insgesamt folgt also

$$\Gamma'_\phi \rhd_m \textbf{method}\ m = e'; r' :: \phi$$

mit der Typregel (Method-Subsume) aus dem Minimal Typing Kalkül.

12.) Wenn $\Gamma_\phi \triangleright \textbf{val}\ a = e'; r' :: \phi$ mit Typregel (Attr) hergeleitet worden ist, gilt nach Voraussetzung $\Gamma^\star_\phi \triangleright e' :: \tau$ und $\Gamma_\phi[^\tau/_a] \triangleright r' :: \phi$. Nach Induktionsvoraussetzung existiert also ein $\tau' \in \mathit{Type}$ mit $\tau' \leq \tau$ und

$$\Gamma'^\star_\phi \triangleright_m e' :: \tau',$$

denn aus $\Gamma'_\phi \leq \Gamma_\phi$ folgt $\Gamma'^\star_\phi \leq \Gamma^\star_\phi$ mit Lemma 3.13. Ebenso gilt $\Gamma'_\phi[^{\tau'}/_a] \leq \Gamma_\phi[^\tau/_a]$ wegen Lemma 3.13 und es folgt

$$\Gamma'_\phi[^{\tau'}/_x] \triangleright_m r' :: \phi$$

mit Induktionsvoraussetzung, denn $\Gamma'_\phi[^{\tau'}/_x] \in \mathit{TEnv}_\phi$. Zusammenfassend folgt also

$$\Gamma'_\phi \triangleright_m \textbf{val}\ a = e'; r' :: \phi$$

mit Typregel (Attr) aus dem Minimal Typing Kalkül.

Die restlichen Fälle verlaufen ähnlich. □

Wie bereits angesprochen, ist damit die Vollständigkeit des Minimal Typing Kalküls gezeigt. Insbesondere gilt das folgende Korollar, welches die wesentliche Aussage des Satzes noch einmal hervorhebt.

Korollar 3.3 *Sei $e \in \mathit{Exp}_P$ ein abgeschlossener Ausdruck. Dann gilt:*

$$\forall \tau \in \mathit{Type} : [\,] \triangleright e :: \tau \ \Rightarrow\ \exists \tau' \in \mathit{Type} : [\,] \triangleright_m e :: \tau' \wedge \tau' \leq \tau$$

Vereinfacht gesprochen gibt es zu jedem Typurteil der Programmiersprache $\mathcal{L}^{sub}_o$, ein äquivalentes in der Programmiersprache $\mathcal{L}^m_o$, wobei aber statt einem der möglichen Typen stets der kleinst-mögliche Typ hergeleitet wird.

Beweis: Klar. □

Kommen wir abschliessend noch einmal auf die eigentliche Motivation für das Minimal Typing zurück: Ein deterministisches Regelwerk für ein Typsystem mit Subsumption. Die wesentlichen Hindernisse waren zuvor die Tatsache, dass die (Subsume)-Regel auf jeden Ausdruck anwendbar ist, und dass bei der (Subsume)-Regel der Supertyp τ, zu dem mit der Anwendung dieser Regel übergegangen wird, durch den Typechecker „geraten“ werden müsste. Intuitiv lößt das Typsystem für die Programmiersprache $\mathcal{L}^m_o$, welches in Definition 3.5 beschrieben ist, diese beiden Probleme, so dass wir nun in der Lage sind den folgenden Satz zu beweisen.

Satz 3.8 (Eindeutigkeit des minimalen Typs)

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

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

Statt beliebige Ausdrücke aus *Exp* zuzulassen, könnten wir uns hierbei ebenfalls auf Exp_P beschränken, also auf Ausdrücke, die der Programmierer im Quelltext des Programms notieren kann. Aber diese Einschränkung wäre unnötig, da wir lediglich zeigen, dass höchstens ein Typ existiert und für Ausdrücke der Form $\omega\#m$ existiert nach Definition kein Typ.

Beweis: Die Behauptung wird durch simultane Induktion über die Grösse von e und r, und Fallunterscheidung nach der syntaktischen Form von e und r bewiesen. Dazu ist für jeden Fall zu zeigen, dass höchstens eine Typregel in Frage kommt, und der Typ eindeutig bestimmt ist durch die Typumgebung Γ und den Ausdruck e bzw. die Reihe r. Wir schon im Beweis der Typeindeutigkeit der Programmiersprache $\mathcal{L}_o^t$ (Satz 2.3) überspringen wir die Details, da es sich um eine triviale Induktion handelt. □

Korollar 3.4 *Sei* Γ *eine Typumgebung.*

(a) Ist $e \in Exp$ *wohlgetypt in* Γ*, dann existiert genau ein* $\tau \in Type$ *mit* $\Gamma \triangleright_m e :: \tau$.

(b) Ist $r \in Row$ *wohlgetypt in* Γ*, dann existiert genau ein* $\phi \in RType$ *mit* $\Gamma \triangleright_m r :: \phi$.

Beweis: Folgt unmittelbar aus Satz 3.8. □

TODO: Basierend auf dem Regelwerk aus Definition 3.5 läßt sich nun ein Algorithmus für die Programmiersprache mit Subtyping angeben.

3.5 Coercions

Neben dem im vorangegangenen Abschnitt beschriebenen Minimal Typing Kalkül existieren noch weitere Ansätze, ein Typsystem mit Subtyping entscheidbar zu machen. In diesem Abschnitt beschreiben wir kurz den in [RV98] und [Rém02] für OCaml vorgestellten Ansatz. Dabei wird die Syntax und Semantik der Programmiersprache um sogenannte *Coercions* erweitert, welche es dem Programmierer ermöglichen für einen bestimmten Ausdruck im Typechecker zu einem grösseren Typ überzugehen.

In einem einfachen Typsystem genügt es die kontextfreie Grammatik von Ausdrücken um die Produktion

$$e \;::=\; (e_1 : \tau')$$

zu erweitern, während für ein Typsystem mit Typinferenz (vgl. [Pie02, S.317ff]), wie zum Beispiel das OCaml Typsystem, die allgemeinere Form

$$e \;::=\; (e_1 : \tau <: \tau')$$

notwendig ist. Wir verwenden im folgenden die allgemeinere Form.

Für die Programmiersprache $\mathcal{L}_o^{sub}$ existiert eine Subsumption-Regel, die es ermöglicht an beliebiger Stelle in einer Typherleitung zu einem Supertyp überzugehen. Gerade diese (Subsume)-Regel war aber der Grund für die Uneindeutigkeit des Typsystems von $\mathcal{L}_o^{sub}$. Statt der (Subsume)-Regel nimmt man deshalb die sogenannte (Coerce)-Regel zum Typsystem hinzu.

$$(\text{Coerce}) \quad \frac{\tau \leq \tau' \quad \Gamma \rhd e :: \tau}{\Gamma \rhd (e : \tau <: \tau') :: \tau'}$$

Hierdurch kann während der Typherleitung ebenfalls zu einem Supertyp übergegangen werden, allerdings immer nur an den Stellen, die der Programmierer explizit dafür vorgesehen hat. Somit erhält man auf recht einfache Weise ein eindeutiges Typsystem mit Subtyping, welches sich im Gegensatz zum Typsystem der Programmiersprache $\mathcal{L}_o^m$ auch um Typinferenz und ML-Polymorphie erweiteren läßt.

Dazu muss dann allerdings auch die Semantik erweitert werden, um eine Regel für den neuen Typ von Ausdruck. Wie bisher führen wir keinerlei Typüberprüfung zur Laufzeit durch, so dass die small step Regel entsprechend einfach aussieht:

$$(\text{Coerce}) \quad (e : \tau <: \tau') \rightarrow e$$

Die im Ausdruck enthaltenen Typinformationen werden verworfen, und es erfolgt eine Auswertung des Ausdrucks mit dem bisherigen small step Regelwerk.

Es ist ziemlich offensichtlich, dass eine solche Programmiersprache typsicher ist, denn für jede Typherleitung im Typsystem dieser Programmiersprache lässt sich eine äquivalente Herleitung im Typsystem der Sprache $\mathcal{L}_o^{sub}$ konstruieren, indem man

(a) alle coercions aus dem Programmtext entfernt, und

(b) an allen Stellen, an denen in der ursprünglichen Herleitung die (Coerce)-Regel angewendet wurde, die (Subsume)-Regel anwendet mit dem bekannten Supertyp.

Das bedeutet, es gilt

$$
\begin{array}{ll}
 & [\,] \triangleright e :: \tau \quad \text{im (Coerce)-Typsystem} \\
\Rightarrow & [\,] \triangleright e :: \tau \quad \text{im Typsystem von } \mathcal{L}_o^{sub} \\
\Rightarrow & e \text{ divergiert oder terminiert mit einem Wert (Satz 3.3).}
\end{array}
$$

Coercion-Ausdrücke sind eingeschränkt vergleichbar mit dem `static_cast` Konstrukt in C++ (vgl. [Str00, S. 440f]), insofern als dass ein `static_cast` ausschliesslich zur Compile-Zeit durch den Typechecker überprüft wird. Da jedoch in C++ für beliebige Zeigertypen ein `static_cast` von und nach `void*` möglich ist, kann hier nicht von einem typsicheren Cast die Rede sein.

Seien beispielsweise `A` und `B` Klassen, wobei `B` von `A` erbt, was in C++ impliziert, dass $\texttt{B} \leq \texttt{A}$ gilt, und sei `a` eine Variable von Typ `A*`, die auf eine gültige Instanz der Klasse `A` zeigt. Dann ist die folgende Zeile ein gültiges C++-Statement:

```
B* b = static_cast<B*> (static_cast<void*> (a));
```

Diese sogenannten Upcasts erfordern in C++ eigentlich einen `reinterpret_cast`, der als generell unsicher deklariert ist. Wie man jedoch sieht, läßt sich auch der scheinbar sichere `static_cast` Operator, der zur Compile-Zeit durch den Typechecker geprüft wird, mit einem einfachen Trick aushebeln. Grund dafür ist, dass `void*` im C++-Typsystem für explizite Typkonversionen zugleich größter, wie auch kleinster Zeigertyp ist[5].

Für implizite Typkonversionen – das C++-Typsystem beinhaltet eine abgewandelte Form der Subsumption-Regel – gilt diese Eigenschaft nicht, so dass hier nur *sinnvolle* Casts erlaubt sind.

[5] Eine der vielen Stellen, in der sich das unliebsame C-Erbe bemerkbar macht.

4 Rekursive Typen

Im vorangegangenen Kapitel haben wir das einfache objekt-orientierte Typsystem aus Kapitel 2 um Subsumption erweitert, so dass in der Sprache $\mathcal{L}_o^{sub}$ deutlich mehr Ausdrücke wohlgetypt sind, als noch in der Programmiersprache $\mathcal{L}_o^t$. Dabei haben wir für $\mathcal{L}_o^{sub}$ gezeigt, dass diese neu hinzugewonnen wohlgetypten Ausdrücke zu Unrecht durch das Typsystem von $\mathcal{L}_o^t$ abgelehnt wurden, denn deren Berechnung bleibt nicht stecken (vgl. Satz 3.3).

Allerdings ist das Typsystem der Sprache $\mathcal{L}_o^{sub}$ immer noch zu eingeschränkt, um sinnvoll mit funktionalen Objekten programmieren zu können. Zum Beispiel ist es nicht möglich den folgenden Ausdruck so mit Typinformationen zu versehen, dass mit den Typregeln von $\mathcal{L}_o^{sub}$ ein Typ herleitbar wäre.

$$
\begin{array}{l}
\textbf{let}\, point = \\
\quad \textbf{object}\, (self) \\
\qquad \textbf{val}\ x = 1; \\
\qquad \textbf{val}\ y = 2; \\
\qquad \textbf{method}\ move = \lambda dx.\lambda dy.\{\langle x = x + dx; y = y + dy\rangle\}; \\
\quad \textbf{end} \\
\textbf{in}\, point\#move\, 2\, 1
\end{array}
$$

Offensichtlich würde die Berechnung dieses Ausdrucks nicht stecken bleiben, sondern stattdessen mit einem Punktobjekt terminieren, mit den Koordinaten $x = 3$ und $y = 3$. Die Methode *move* liefert ein neues Punktobjekt, welches sich lediglich in den Werten der Attribute x und y unterscheidet. Insbesondere besitzt aber dieses neue Punktobjekt ebenfalls eine Methode *move*.

Für das Typsystem bedeutet das, dass ein Typ für *point* „sich selbst enthalten“ müsste, und zwar als Ergebnistyp der Methode *move*. Oder anders ausgedrückt, gesucht ist ein Typ τ_{point} für den gilt:

$$\tau_{point} = \langle move : \textbf{int} \to \textbf{int} \to \tau_{point};\, \emptyset \rangle$$

Es ist leicht zu sehen, dass im Typsystem der Sprache $\mathcal{L}_o^{sub}$ kein solcher Typ τ_{point} existiert, der diese (rekursive) Gleichung erfüllt. Um also Objekten, die sich selbst oder ein Duplikat von sich selbst als Ergebnis eines Methodenaufrufs liefern, Typen zuordnen zu

können, ist es notwendig das Typsystem um sogenannte *rekursive Typen* (engl.: *recursive types*) zu erweitern.

TODO: Darstellung als unendlicher Baum, und Überleitung zur endlichen Darstellung dieser unendlichen Struktur (μ-types)

4.1 Die Sprache $\mathcal{L}_o^{rt}$

Die im diesem Abschnitt betrachtete Sprache $\mathcal{L}_o^{rt}$ stellt eine Erweiterung des Typsystems der Sprache $\mathcal{L}_o^t$ um rekursive Typen dar. Die Semantik der Sprache $\mathcal{L}_o^t$ wird bis auf die folgenden syntaktischen Änderungen unverändert übernommen.

Definition 4.1 (Syntax der Sprache $\mathcal{L}_o^{rt}$) Vorgegeben sei eine Menge *TName* von *Typnamen* t.

(a) Die Menge Type^{raw} aller syntaktisch herleitbaren Typen τ^{raw} ist durch die kontextfreie Grammatik

$$\begin{array}{rcl} \tau^{raw} & ::= & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit} \\ & \mid & \tau_1^{raw} \to \tau_2^{raw} \\ & \mid & \mu t.\tau_1^{raw} \\ & \mid & \langle \phi^{raw} \rangle \end{array}$$

und die Menge RType^{raw} aller syntaktisch herleitbaren Reihentypen ϕ^{raw} ist durch

$$\begin{array}{rcl} \phi^{raw} & ::= & \emptyset \\ & \mid & m : \tau^{raw};\ \phi_1^{raw} \end{array}$$

definiert.

(b) Die Menge $\mathit{free}\,(\tau^{raw})$ aller im Typ $\tau^{raw} \in \mathit{Type}^{raw}$ frei vorkommenden Typnamen ist induktiv durch

$$\begin{array}{rcl} \mathit{free}\,(\tau_\beta) & = & \emptyset \text{ für } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ \mathit{free}\,(\tau_1^{raw} \to \tau_2^{raw}) & = & \mathit{free}\,(\tau_1^{raw}) \cup \mathit{free}\,(\tau_2^{raw}) \\ \mathit{free}\,(\mu t.\tau_1^{raw}) & = & \mathit{free}\,(\tau_1^{raw}) \setminus \{t\} \\ \mathit{free}\,(\langle \phi^{raw} \rangle) & = & \mathit{free}\,(\phi^{raw}) \end{array}$$

und die Menge $\mathit{free}\,(\phi^{raw})$ aller im Reihentyp $\phi^{raw} \in \mathit{RType}^{raw}$ frei vorkommenden Typnamen ist induktiv durch

$$\begin{array}{rcl} \mathit{free}\,(\emptyset) & = & \emptyset \\ \mathit{free}\,(m : \tau^{raw};\ \phi_1^{raw}) & = & \mathit{free}\,(\tau^{raw}) \cup \mathit{free}\,(\phi_1^{raw}) \end{array}$$

definiert.

(c) Die Mengen $\mathit{Type} \subseteq \mathit{Type}^{raw}$ aller gültigen Typen τ und $\mathit{RType}^{raw} \subseteq \mathit{RType}$ aller gültigen Reihentypen ϕ der Programmiersprache $\mathcal{L}_o^{rt}$ sind wie folgt definiert

$$\begin{array}{rcl} \mathit{Type} & = & \{\tau \in \mathit{Type}^{raw} \mid \mathit{free}\,(\tau) = \emptyset \wedge \forall t, t_1, \ldots, t_n \in \mathit{TName} : \mu t.\mu t_1. \ldots \mu t_n.t \notin \mathcal{P}(\tau)\} \\ \mathit{RType} & = & \{\phi \in \mathit{RType}^{raw} \mid \mathit{free}\,(\phi) = \emptyset\}, \end{array}$$

wobei $\mathcal{P}(\tau)$ die Menge aller im Typ τ vorkommenden Typen bezeichnet.

Rekursive Typen sind hierbei analog zu rekursiven Ausdrücken zu lesen, d.h. mit $\mu t.\tau$ bezeichnen wir den Typ, der die Gleichung

$$\mu t.\tau = \tau[{}^{\mu t.\tau}/_{t}]$$

erfüllt. Allerdings handelt es sich hierbei nicht um die syntaktische Gleichheit der beiden Typen, sondern vielmehr die übereinstimmende Bedeutung.

TODO: Erläuterungen zur Definition

Bevor wir uns jedoch genauer mit dem Begriff der Gleichheit von rekursiven Typen beschäftigen, kommen wir zunächst auf das Beispiel mit dem Punktobjekt zurück. In der Sprache $\mathcal{L}_o^{rt}$ lässt sich das Beispiel nun wie folgt mit Typinformationen versehen.

$$\begin{array}{l} \textbf{let}\, \mathit{point} = \\ \quad \textbf{object}\,(\mathit{self} : \mu t.\langle \mathit{move} : \textbf{int} \to \textbf{int} \to t;\, \emptyset\rangle) \\ \quad\quad \textbf{val}\ x = 1; \\ \quad\quad \textbf{val}\ y = 2; \\ \quad\quad \textbf{method}\ \mathit{move} = \lambda dx : \textbf{int}.\lambda dy : \textbf{int}.\{\langle x = x + dx; y = y + dy\rangle\}; \\ \quad \textbf{end} \\ \textbf{in}\, \mathit{point}\#\mathit{move}\, 2\, 1 \end{array}$$

Intuitiv sollte sich für diesen Ausdruck der Typ des Punktobjekts herleiten lassen, also

$$\mu t.\langle \mathit{move} : \textbf{int} \to \textbf{int} \to t;\, \emptyset\rangle,$$

was dem zu erwartenden Ergebnis entsprechen würde.

Definition 4.2 Die Menge *TCons* der Typkonstruktoren sei definiert durch

$$\mathit{TCons} \;:=\; \{\textbf{bool}, \textbf{int}, \textbf{unit}, \to\} \cup \{\langle m_1; \ldots; m_n\rangle \mid n \in \mathbb{N},\ m_1, \ldots, m_n \in \mathit{Method}\},$$

wobei die Methodennamen in $\langle m_1; \ldots; m_n\rangle$ ähnlich wie bei den Typen als disjunkt angenommen werden.

Definition 4.3

(a) Sei $root : Type \to TCons$ die induktiv wie folgt definierte Funktion:

$$\begin{aligned} root\,(\tau_\beta) &= \tau_\beta \quad \text{für } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ root\,(\tau_1 \to \tau_2) &= \to \\ root\,(\langle m_i : {\tau_i}^{i=1\ldots n}\rangle) &= \langle m_i^{i=1\ldots n}\rangle \\ root\,(\mu t.\tau) &= root\,(\tau[{}^{\mu t.\tau}/_t]) \end{aligned}$$

(b) Die Funktion $arity : Type \to \mathbb{N}$ sei induktiv definiert durch:

$$\begin{aligned} arity\,(\tau_\beta) &= 0 \quad \text{für } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ arity\,(\tau_1 \to \tau_2) &= 2 \\ arity\,(\langle m_i : {\tau_i}^{i=1\ldots n}\rangle) &= n \\ arity\,(\mu t.\tau) &= arity\,(\tau[{}^{\mu t.\tau}/_t]) \end{aligned}$$

(c) Die partielle Funktion $child : Type \times \mathbb{N} \rightharpoonup Type$ ist induktiv definiert durch:

$$\begin{aligned} child\,(\tau_1 \to \tau_2, i) &= \begin{cases} \tau_i & \text{falls } 1 \le i \le 2 \\ \uparrow & \text{sonst} \end{cases} \\ child\,(\langle m_j : {\tau_j}^{j=1\ldots n}\rangle, i) &= \begin{cases} \tau_i & \text{falls } 1 \le i \le n \\ \uparrow & \text{sonst} \end{cases} \\ child\,(\mu t.\tau, i) &= child\,(\tau[{}^{\mu t.\tau}/_t], i) \end{aligned}$$

Statt $child(\tau, i)$ schreiben wir kurz $child_i\,(\tau)$.

TODO: Prosa... hiermit klar, warum *Type* wie oben definiert werden muss. Es ist klar, dass $child_i\,(\tau)$ stets definiert ist, solange $1 \le i \le arity\,(\tau)$.

Korollar 4.1 $\forall \tau, \tau' \in Type :\ root\,(\tau) = root\,(\tau') \ \Rightarrow\ arity\,(\tau) = arity\,(\tau')$

Beweis: Trivial. □

Definition 4.4 Die Relationen $\sim_n$ seien wie folgt induktiv definiert

$$\begin{aligned} \sim_0 &= Type^2 \\ \sim_{n+1} &= \{(\tau, \tau') \in Type^2 \mid root\,(\tau) = root\,(\tau') \\ &\qquad \wedge\, \forall 1 \le i \le arity\,(\tau) :\ child_i\,(\tau) \sim_n child_i\,(\tau')\}, \end{aligned}$$

und die Relation $\sim$ ist durch

$$\sim \ = \ \bigcap_{n \in \mathbb{N}} \sim_n$$

definiert, also als Durchschnitt der Relationen $\sim_n$.

TODO: Überleitung

Lemma 4.1 *Die Relation $\sim$ ist eine Äquivalenzrelation.*

Beweis: Hierzu genügt es zu zeigen, dass jede Relation $\sim_n$ eine Äquivalenzrelation ist, denn der Schnitt von Äquivalenzrelationen ist wiederum eine Äquivalenzrelation. Wir zeigen also durch Induktion über n, dass alle $\sim_n$ reflexiv, transitiv und symmetrisch sind.

- $n = 0$

  Da $\sim_0 = \mathit{Type}^2$ ist klar, dass $\sim_0$ reflexiv, transitiv und symmetrisch ist.

- $n \rightsquigarrow n+1$

  Nach Induktionsvoraussetzung ist $\sim_n$ reflexiv, transitiv und symmetrisch.

  Reflexivität: Da $\sim_n$ reflexiv ist, folgt trivialerweise, dass auch $\sim_{n+1}$ reflexiv ist.

  Transivität: Seien $(\tau_1, \tau_2), (\tau_2, \tau_3) \in \sim_{n+1}$. Dann gilt $\mathit{root}(\tau_1) = \mathit{root}(\tau_2) = \mathit{root}(\tau_3)$, $\mathit{child}_i(\tau_1) \sim_n \mathit{child}_i(\tau_2)$ und $\mathit{child}_i(\tau_2) \sim_n \mathit{child}_i(\tau_3)$ für alle $i = 1, \ldots, \mathit{arity}(\tau_2)$. Da $\sim_n$ transitiv ist, folgt $\mathit{child}_i(\tau_1) \sim_n \mathit{child}_i(\tau_3)$ für alle $i = 1, \ldots, \mathit{arity}(\tau_1)$, und somit $(\tau_1, \tau_3) \in \sim_{n+1}$.

  Symmetrie: Sei $(\tau, \tau') \in \sim_{n+1}$. Dann ist nach Definition $\mathit{root}(\tau) = \mathit{root}(\tau')$ und es gilt $\mathit{child}_i(\tau) \sim_n \mathit{child}_i(\tau')$ für alle $i = 1, \ldots, \mathit{arity}(\tau)$. Da $\sim_n$ symmetrisch ist, folgt $\mathit{child}_i(\tau') \sim_n \mathit{child}_i(\tau)$, also $(\tau, \tau') \in \sim_{n+1}$.

Also ist $\sim$ eine Äquivalenzrelation auf *Type*. □

TODO: Fixpunktsatz schon für $\sim$ notwendig, wird unten implizit verwendet (z.B. im Lemma über die Umkehrung der Typrelation)

TODO: Prosa

Definition 4.5 (Gültige Typurteile für $\mathcal{L}_o^{rt}$) Ein Typurteil $\Gamma \triangleright_e e :: \tau$ oder $\Gamma \triangleright_r r :: \phi$ heisst *gültig* für $\mathcal{L}_o^{rt}$, wenn es sich mit den Typregeln der Programmiersprache $\mathcal{L}_o^t$ aus Definition 2.20 mit Ausnahme der (Object)-Regel, sowie den folgenden neuen Typregeln

$$\text{(Object')} \quad \frac{\Gamma^\star[\tau/_{self}] \triangleright_r r :: \phi \quad \tau \sim \langle\phi\rangle}{\Gamma \triangleright_e \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau}$$

$$\text{(Equiv)} \quad \frac{\Gamma \triangleright_e e :: \tau' \quad \tau' \sim \tau}{\Gamma \triangleright_e e :: \tau}$$

herleiten lässt.

TODO: Beispiel für Typherleitung
TODO: Überleitung

4.1.1 Typsicherheit

TODO: Zunächst die obligatorischen Lemmata

Lemma 4.2 (Typumgebungen und frei vorkommende Namen)

(a) $\forall \Gamma \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma \triangleright e :: \tau \Rightarrow \mathit{free}(e) \subseteq \mathit{dom}(\Gamma)$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma \triangleright r :: \phi \Rightarrow \mathit{free}(r) \subseteq \mathit{dom}(\Gamma)$

Beweis: Entspricht dem Beweis von Lemma 2.5 für die Programmiersprache $\mathcal{L}_o^t$, wobei der neue Fall für die (Equiv)-Regel direkt mit Induktionsvoraussetzung folgt. □

Lemma 4.3 (Koinzidenzlemma)

(a) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma_1 \triangleright e :: \tau \wedge \Gamma_1 =_{\mathit{free}(e)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright e :: \tau$

(b) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma_1 \triangleright r :: \phi \wedge \Gamma_1 =_{\mathit{free}(r)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright r :: \phi$

Beweis: Wiederum wie im Beweis von Lemma 2.6 für die Programmiersprache $\mathcal{L}_o^t$, und der Fall für die (Equiv)-Regel folgt ebenfalls direkt mit Induktionsvoraussetzung. □

Lemma 4.4 (Typurteile und Substitution) *Sei* $id \in \mathit{Attribute} \cup \mathit{Var}$, $\Gamma \in \mathit{TEnv}$, $\tau \in \mathit{Type}$ *und* $e \in \mathit{Exp}$. *Dann gilt:*

(a) $\forall e' \in \mathit{Exp} : \forall \tau' \in \mathit{Type} : \Gamma[{}^{\tau}/_{id}] \triangleright e' :: \tau' \wedge \Gamma^{\star} \triangleright e :: \tau \Rightarrow \Gamma \triangleright e'[{}^{e}/_{id}] :: \tau'$

(b) $\forall r \in \mathit{Row} : \forall \phi \in \mathit{RType} : \Gamma[{}^{\tau}/_{id}] \triangleright r :: \phi \wedge \Gamma^{\star} \triangleright e :: \tau \Rightarrow \Gamma \triangleright r[{}^{e}/_{id}] :: \phi$

Beweis: Wie im Beweis von Lemma 2.7 mittels simultaner Induktion über die Länge der Herleitung der Typurteile $\Gamma[{}^{\tau}/_{id}] \triangleright e' :: \tau'$ und $\Gamma[{}^{\tau}/_{id}] \triangleright r :: \phi$, und Fallunterscheidung nach der letzten angewandten Typregel. Wir betrachten nur die Fälle der (Equiv)- und (Object')-Regeln.

1.) $\Gamma[{}^{\tau}/_{id}] \triangleright e' :: \tau'$ mit Typregel (Equiv).

Nach Voraussetzung existiert ein $\tau'' \in \mathit{Type}$ mit $\Gamma[{}^{\tau}/_{id}] \triangleright e' :: \tau''$ und $\tau'' \sim \tau'$. Mit Induktionsvoraussetzung folgt dann

$$\Gamma \triangleright e'[{}^{e}/_{id}] :: \tau'',$$

und somit wegen

$$\tau'' \sim \tau'$$

mit Typregel (EQUIV) das erwartete Ergebnis $\Gamma \triangleright e'[^{e}/_{id}] :: \tau'$.

2.) $\Gamma[^{\tau}/_{id}] \triangleright \mathbf{object}\,(\mathit{self} : \tau')\ r\ \mathbf{end} :: \tau'$ mit Typregel (OBJECT').

Dann existiert ein $\phi \in \mathit{RType}$ mit $(\Gamma[^{\tau}/_{id}])^{\star}[^{\tau'}/_{self}] \triangleright r :: \phi$ und $\langle\phi\rangle \sim \tau'$.

Falls $id \in \mathit{Var}$, dann gilt nach Definition $(\Gamma[^{\tau}/_{id}])^{\star} = \Gamma^{\star}[^{\tau}/_{id}]$. Andererseits gilt für $id \in \mathit{Attribute}$, dass $(\Gamma[^{\tau}/_{id}])^{\star} = \Gamma^{\star}$, also nach Lemma 4.2 $id \notin \mathit{free}\,(r)$. Demzufolge gilt $\Gamma^{\star} =_{\mathit{free}\,(r)} \Gamma^{\star}[^{\tau}/_{id}]$. Zusammenfassend folgt also

$$\Gamma^{\star}[^{\tau}/_{id}][^{\tau'}/_{self}] \triangleright r :: \phi,$$

und da $\mathit{self} \notin \mathit{Attribute} \cup \mathit{Var}$ folgt weiter

$$\Gamma^{\star}[^{\tau'}/_{self}][^{\tau}/_{id}] \triangleright r :: \phi.$$

Darauf lässt sich nun die Induktionsvoraussetzung anwendung und wir erhalten

$$\Gamma^{\star}[^{\tau'}/_{self}] \triangleright r[^{e}/_{id}] :: \phi,$$

woraus sich wegen $\langle\phi\rangle \sim \tau'$ mit Typregel (OBJECT') schliesslich

$$\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau')\ r[^{e}/_{id}]\ \mathbf{end} :: \tau'$$

ergibt, was nach Definition der Substitution (2.8) dem Typurteil

$$\Gamma \triangleright (\mathbf{object}\,(\mathit{self} : \tau')\ r\ \mathbf{end})[^{e}/_{id}] :: \tau'$$

entspricht.

Die übrigen Fälle verlaufen entsprechend wie im Beweis des Lemmas für die Programmiersprache $\mathcal{L}_o^t$. □

Zusätzlich zum Substitutionslemma für Attribute und Variablen (4.4) benötigen wir wieder ein spezielles Lemma, welches die Typerhaltung der *self*-Substitution sichert (siehe Lemma 2.9).

Lemma 4.5 (Typurteile und *self*-Substitution) *Sei* $\Gamma \in \mathit{TEnv}$, $\mathit{self} \in \mathit{Self}$, $\tau \in \mathit{Type}$ *und* $r \in \mathit{Row}$. *Dann gilt:*

(a) Für alle $e \in \mathit{Exp}$ *und* $\tau' \in \mathit{Type}$:

(1) $\Gamma[\tau/_{\mathit{self}}] \triangleright e :: \tau'$

(2) $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$

(3) $\forall a \in \mathit{dom}(\Gamma) \cap \mathit{Attribute}, \tau_a \in \mathit{Type} : \Gamma^\star \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$

$\Rightarrow \Gamma \triangleright e[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$

(b) Für alle $r' \in \mathit{Row}$ *und* $\phi \in \mathit{RType}$:

(1) $\Gamma[\tau/_{\mathit{self}}] \triangleright r' :: \phi$

(2) $\Gamma^\star \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r \oplus r'\ \mathbf{end} :: \tau$

(3) $\forall a \in \mathit{dom}(\Gamma) \cap \mathit{Attribute}, \tau_a \in \mathit{Type} : \Gamma^\star \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$

$\Rightarrow \Gamma \triangleright r'[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\oplus r'\ \mathbf{end}}/_{\mathit{self}}] :: \phi$

Beweis: Der Beweis entspricht im wesentlichen dem Beweis von Lemma 2.9 für die Programmiersprache $\mathcal{L}_o^t$. Wie zuvor folgt gemäß Lemma 4.2, dass die Substitution stets definiert ist. Der Beweis erfolgt dann durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[\tau/_{\mathit{self}}] \triangleright e :: \tau'$ und $\Gamma[\tau/_{\mathit{self}}] \triangleright r' :: \phi$, und Fallunterscheidung nach der letzten Typregel in der Herleitung. Wir betrachten hier lediglich die Fälle der (Equiv)- und (Object')-Regeln.

1.) $\Gamma[\tau/_{\mathit{self}}] \triangleright e :: \tau'$ mit Typregel (Equiv).

Nach Voraussetzung existiert ein $\tau'' \in \mathit{Type}$ mit $\Gamma[\tau/_{\mathit{self}}] \triangleright e :: \tau''$ und $\tau'' \sim \tau'$. Darauf lässt sich direkt die Induktionsvoraussetzung anwenden, und es folgt

$$\Gamma \triangleright e[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau''.$$

Wegen $\tau'' \sim \tau'$ folgt unmittelbar

$$\Gamma \triangleright e[^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}] :: \tau'$$

mit Typregel (Equiv).

2.) $\Gamma[^{\tau}/_{self}] \triangleright \mathbf{object}\,(self : \tau')\ r''\ \mathbf{end} :: \tau'$ mit Typregel (OBJECT').

In diesem Fall folgt die Behauptung trivialerweise, da nach Definition 2.8 gilt

$$(\mathbf{object}\,(self : \tau')\ r''\ \mathbf{end})[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] \quad = \quad \mathbf{object}\,(self : \tau')\ r''\ \mathbf{end},$$

und es gilt $self \notin free\,(\mathbf{object}\,(self : \tau')\ r''\ \mathbf{end})$ gemäß Definition 2.5. Mit Lemma 4.3 folgt dann

$$\Gamma \triangleright \mathbf{object}\,(self : \tau')\ r''\ \mathbf{end} :: \tau',$$

was zu zeigen war.

Die restlichen Fälle verlaufen wie im Beweis von Lemma 2.9. □

Bevor wir nun das Preservation-Theorem für die Programmiersprache $\mathcal{L}_o^{rt}$ formulieren und beweisen können, benötigen wir ähnlich wie im Fall der Typerhaltung der Programmiersprache $\mathcal{L}_o^{sub}$ ein Lemma, welches im wesentlichen einer Umkehrung der Typrelation entspricht. Dieses Lemma ermöglicht eine einfachere Argumentation in den Beweisen der Preservation- und Progress-Theoreme. Wie zuvor beschränken wir uns auf die Aussagen, die wir in den nachfolgenden Beweisen benötigen, statt das Lemma in seiner allgemeinsten Form anzugeben.

Lemma 4.6 (Umkehrung der Typrelation)

(a) Wenn $\Gamma \triangleright op :: \tau$*, dann gilt* $\tau \sim \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ *für* $op \in \{+, -, *\}$ *oder* $\tau \sim \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ *für* $op \in \{\leq, \geq, <, >, =\}$.

(b) Wenn $\Gamma \triangleright \lambda x : \tau_2.e :: \tau'$*, dann gilt* $\Gamma[^{\tau_2}/_{x}] \triangleright e :: \tau$ *mit* $\tau' \sim \tau_2 \to \tau$.

(c) Wenn $\Gamma \triangleright e_1\, e_2 :: \tau$*, dann gilt* $\Gamma \triangleright e_1 :: \tau_2 \to \tau$ *und* $\Gamma \triangleright e_2 :: \tau_2$.

(d) Wenn $\Gamma \triangleright \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau'$*, dann gilt* $\Gamma^{\star}[^{\tau}/_{self}] \triangleright r :: \phi$ *mit* $\langle\phi\rangle \sim \tau' \sim \tau$.

Beweis: Die Beweise führen wir jeweils durch vollständige Induktion über die Länge der Herleitung der Typurteile, mit Fallunterscheidung nach der zuletzt angewandten Typregel. Offensichtlich kommen in jedem Fall nur exakt zwei Typregeln in Frage.

(a) $\Gamma \triangleright op :: \tau$ kann nur mit einer der Typregeln (CONST) oder (EQUIV) hergeleitet worden sein.

1.) Für $\Gamma \triangleright op :: \tau$ mit Typregel (CONST) folgt die Behauptung unmittelbar nach Definition, denn $\sim$ ist reflexiv.

2.) Wenn $\Gamma \triangleright op :: \tau$ mit Typregel (EQUIV) hergeleitet wurde, existiert nach Voraussetzung ein $\tau' \in \mathit{Type}$ mit $\Gamma \triangleright op :: \tau'$ und $\tau' \sim \tau$. Da $\sim$ transitiv ist, folgt mit Induktionsvoraussetzung die Behauptung.

(b) Eine Herleitung des Typurteils $\Gamma \triangleright \lambda x : \tau_2.e :: \tau'$ kann ausschliesslich mit Anwendungen von (ABSTR) oder (EQUIV) enden.

1.) Endet die Herleitung mit einer Anwendung von (ABSTR), so gilt nach Voraussetzung $\Gamma[^{\tau_2}/_x] \triangleright e :: \tau$ und $\tau' = \tau_2 \to \tau$. Also $\tau' \sim \tau_2 \to \tau$, da $\sim$ reflexiv ist.

2.) Im Fall von (EQUIV) existiert wieder ein $\tau'' \in \mathit{Type}$, so dass $\Gamma \triangleright \lambda x : \tau_2.e :: \tau''$ mit $\tau'' \sim \tau'$ gilt. Nach Induktionsvoraussetzung folgt $\Gamma[^{\tau_2}/_x] \triangleright e :: \tau$ mit $\tau'' \sim \tau_2 \to \tau$. Wegen der Transivität von $\sim$ folgt also die Behauptung.

(c) Für $\Gamma \triangleright e_1\, e_2 :: \tau$ kommen nur die Typregeln (APP) und (EQUIV) in Frage.

1.) Im Fall der (APP)-Regel folgt die Behauptung unmittelbar aus den Voraussetzungen.

2.) Wurde andererseits $\Gamma \triangleright e_1\, e_2 :: \tau$ mit Typregel (EQUIV) hergeleitet, so existiert ein $\tau' \in \mathit{Type}$ mit $\Gamma \triangleright e_1\, e_2 :: \tau'$ und $\tau' \sim \tau$. Nach Induktionsvoraussetzung existiert dann ein $\tau_2 \in \mathit{Type}$ mit $\Gamma \triangleright e_1 :: \tau_2 \to \tau'$ und $\Gamma \triangleright e_2 :: \tau_2$. Wegen $\tau' \sim \tau$ gilt ebenfalls $\tau_2 \to \tau' \sim \tau_2 \to \tau$, und mit Anwendung der (EQUIV)-Regel folgt schliesslich $\Gamma \triangleright e_1 :: \tau_2 \to \tau$.

(d) Das Typurteil $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ kann nur mit einer der Typregeln (EQUIV) oder (OBJECT') hergeleitet worden sein.

1.) Für (OBJECT') folgt die Behauptung wegen der Reflexivität von $\sim$ bereits aus den Voraussetzung der Regel.

2.) Im Fall von (EQUIV) folgt die Behauptung wegen der Transitivität von $\sim$ unmittelbar aus der Induktionsvoraussetzung. □

Dank dieses Lemmas können wir im Folgenden Anwendungen der (EQUIV)-Regel im wesentlichen ignorieren, so dass sich die Beweise deutlich vereinfachen.

Nach diesen Vorbereitungen können wir nun endlich das Preservation-Theorem für die Programmiersprache $\mathcal{L}_o^{rt}$ formulieren und beweisen.

Satz 4.1 (Typerhaltung, „Preservation“)

(a) $\forall \Gamma \in \mathit{TEnv} : \forall e, e' \in \mathit{Exp} : \forall \tau \in \mathit{Type} :\ \Gamma^\star \triangleright e :: \tau \ \wedge\ e \to e' \ \Rightarrow\ \Gamma^\star \triangleright e' :: \tau$

(b) $\forall \Gamma \in \mathit{TEnv} : \forall r, r' \in \mathit{Row} : \forall \phi \in \mathit{RType} : \Gamma \triangleright r :: \phi \wedge r \to r' \;\Rightarrow\; \Gamma \triangleright r' :: \phi$

Beweis: Wie bereits im Beweis des Preservation-Theorems für die Programmiersprache $\mathcal{L}_o^{sub}$ (Satz 3.1) erfolgt der Beweis durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma^\star \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi$, mit Fallunterscheidung nach der letzten Typregel in der Herleitung. Dies ist notwendig, da die Typregeln der Programmiersprache $\mathcal{L}_o^{rt}$, ebenso wie die der Sprache $\mathcal{L}_o^{sub}$, nicht eindeutigkeit rückwärtsanwendbar sind[1].

Wir betrachten exemplarisch die folgenden Fälle.

1.) Für (Const), (Id), (Abstr) und (Empty) gilt gemäß Lemma 2.3 $e \not\to$ bzw. $r \not\to$.

2.) $\Gamma^\star \triangleright e :: \tau$ mit Typregel (Equiv).

Nach Voraussetzung existiert ein $\tau' \in \mathit{Type}$, so dass $\Gamma^\star \triangleright e :: \tau'$ mit $\tau' \sim \tau$ gilt. Mit Induktionsvoraussetzung folgt daraus

$$\Gamma^\star \triangleright e' :: \tau',$$

woraus dann wegen $\tau' \sim \tau$ mit Typregel (Equiv) das erwartete Ergebnis

$$\Gamma^\star \triangleright e' :: \tau$$

folgt.

3.) $\Gamma^\star \triangleright e_1\, e_2 :: \tau$ mit Typregel (App).

Nach Voraussetzung gilt $\Gamma^\star \triangleright e_1 :: \tau_2 \to \tau$ und $\Gamma^\star \triangleright e_2 :: \tau_2$. Der small step $e_1\, e_2 \to e'$ kann nur mit einer der small step Regeln (App-Left), (App-Right), (Beta-V) oder (Op) hergeleitet worden sein. Entsprechend unterscheiden wir nach der zuletzt angewendeten small step Regel:

1.) $e_1\, e_2 \to e'$ mit (App-Left) impliziert $e' = e_1'\, e_2$ und $e_1 \to e_1'$. Nach Induktionsvoraussetzung folgt $\Gamma^\star \triangleright e_1' :: \tau_2 \to \tau$, also $\Gamma^\star \triangleright e_1'\, e_2 :: \tau$ mit Typregel (App).

2.) $e_1\, e_2 \to e'$ mit small step Regel (App-Right) kann nur aus $e_1 \in \mathit{Val}$, $e_2 \to e_2'$ und $e' = e_1\, e_2'$ folgen. Die Behauptung folgt dann mit Induktionsvoraussetzung und Typregel (App).

[1] Genauergesagt, kann auf jeden Ausdrucks nicht nur die eigentlich für diesen Ausdruck vorgesehene Typregel, sondern ebenfalls die (Equiv)-Regel, angewandt werden.

3.) Wenn $e_1\, e_2 \to e'$ mit (Beta-V) hergeleitet wurde, gilt $e_1 = \lambda x : \tau_1.e_1'$ und $e_2 \in \mathit{Val}$, sowie $e' = e_1'[{}^{e_2}/_x]$. Nach Lemma 4.6 gilt

$$\Gamma^\star[{}^{\tau_1}/_x] \rhd e_1' :: \tau'$$

mit $\tau_1 \to \tau' \sim \tau_2 \to \tau$. Nach Definition von $\sim$ (4.4) also insbesondere $\tau_1 \sim \tau_2$ und $\tau' \sim \tau$. Mit Typregel (Equiv) folgt aus $\Gamma^\star \rhd e_2 :: \tau_2$ und $\tau_2 \sim \tau_1$, dass auch

$$\Gamma^\star \rhd e_2 :: \tau_1$$

gilt. Nach Substitutionslemma (4.4) folgt somit

$$\Gamma^\star \rhd e_1'[{}^{e_2}/_x] :: \tau',$$

und wegen $\tau' \sim \tau$ schliesslich

$$\Gamma^\star \rhd e_1'[{}^{e_2}/_x] :: \tau$$

mit Typregel (Equiv).

4.) $e_1\, e_2 \to e'$ mit small step Regel (Op) bedingt $e_1 = \mathit{op}\, v_1$, $e_2 = v_2 \in \mathit{Val}$ und $e' = \mathit{op}^I(v_1, v_2)$. Mit Lemma 4.6 folgt, dass

$$\Gamma^\star \rhd \mathit{op} :: \tau_1 \to \tau_2 \to \tau$$

und

$$\Gamma^\star \rhd v_1 :: \tau_1$$

gilt. Nach Lemma 4.6 folgt weiterhin, dass $\tau_1 \sim \mathbf{int}$, $\tau_2 \sim \mathbf{int}$, und $\tau \sim \mathbf{int}$ für $\mathit{op} \in \{+, -, *\}$ bzw. $\tau \sim \mathbf{bool}$ für $\mathit{op} \in \{\leq, \geq, <, >, =\}$.
Sei also $\mathit{op} \in \{+, -, *\}$, dann gilt $\mathit{op}^I(v_1, v_2) \in \mathit{Int}$. Mit Typregel (Const) folgt

$$\Gamma^\star \rhd \mathit{op}^I(v_1, v_2) :: \mathbf{int},$$

und wegen $\tau \sim \mathbf{int}$ folgt schliesslich

$$\Gamma^\star \rhd \mathit{op}^I(v_1, v_2) :: \tau$$

mit Typregel (Equiv).
Für $\mathit{op} \in \{\leq, \geq, <, >, =\}$ folgt die Behauptung analog.

4.) $\Gamma^\star \rhd \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ mit Typregel (Object').

Der small step $e \to e'$ kann nur mit der small step Regel (OBJECT-EVAL) aus $r \to r'$ folgen, und es gilt $e' = \textbf{object}\,(\mathit{self} : \tau)\ r'\ \textbf{end}$. Nach Voraussetzung existiert ein $\phi \in \mathit{RType}$ mit

$$\Gamma^\star[^\tau/_{\mathit{self}}] \triangleright r :: \phi$$

und $\langle\phi\rangle \sim \tau$. Mit Induktionsvoraussetzung folgt daraus

$$\Gamma^\star[^\tau/_{\mathit{self}}] \triangleright r' :: \phi,$$

also mit Typregel (OBJECT') die Behauptung.

5.) $\Gamma^\star \triangleright e_1 \# m :: \tau$ mit Typregel (SEND).

Dann gilt $\Gamma^\star \triangleright e_1 :: \langle m : \tau;\ \phi\rangle$. Der small step $e_1 \# m \to e'$ kann nur mit den small step Regeln (SEND-EVAL) oder (SEND-UNFOLD) hergeleitet worden sein.

1.) Der small step $e_1 \# m \to e_1' \# m$ mit Regel (SEND-EVAL) kann nur aus $e_1 \to e_1'$ folgen. Nach Induktionsvoraussetzung gilt

$$\Gamma^\star \triangleright e_1' :: \langle m : \tau;\ \phi\rangle,$$

also folgt

$$\Gamma^\star \triangleright e_1' \# m :: \tau$$

mit Typregel (SEND).

2.) $e_1 \# m \to e'$ mit (SEND-UNFOLD) bedingt $e_1 = \textbf{object}\,(\mathit{self} : \tau_1)\ \omega\ \textbf{end}$ und $e' = \omega[^{e_1}/_{\mathit{self}}]\#m$. Nach Lemma 4.6 gilt

$$\Gamma^\star[^{\tau_1}/_{\mathit{self}}] \triangleright \omega :: \phi_1$$

mit $\langle\phi_1\rangle \sim \langle m : \tau;\ \phi\rangle \sim \tau_1$. Also existieren nach Definition 4.4 $\tau' \in \mathit{Type}$ und $\phi_1' \in \mathit{RType}$ mit $\phi_1 = (m : \tau';\ \phi_1')$ und $\tau' \sim \tau$. Nach Voraussetzung gilt schon

$$\Gamma^\star \triangleright \textbf{object}\,(\mathit{self} : \tau_1)\ \omega\ \textbf{end} :: \langle m : \tau;\ \phi\rangle,$$

und nach Definition 2.18 gilt $\Gamma^\star = (\Gamma^\star)^\star$, somit also

$$(\Gamma^\star)^\star \triangleright \textbf{object}\,(\mathit{self} : \tau_1)\ \omega\ \textbf{end} :: \langle m : \tau;\ \phi\rangle.$$

Wegen $\langle m : \tau;\ \phi\rangle \sim \tau_1$ folgt daraus

$$(\Gamma^\star)^\star \triangleright \textbf{object}\,(\mathit{self} : \tau_1)\ \omega\ \textbf{end} :: \tau_1$$

mit Typregel (EQUIV). Darauf läßt sich nun das spezielle *self*-Substitutions-Lemma (4.5) anwenden, und wir erhalten

$$\Gamma^\star \rhd \omega\big[^{\mathbf{object}\,(self:\tau_1)\ \omega\ \mathbf{end}}/_{self}\big] :: \phi_1.$$

Da $\phi_1 = (m : \tau'; \phi_1')$ folgt weiter mit Typregel (SEND')

$$\Gamma^\star \rhd \omega\big[^{\mathbf{object}\,(self:\tau_1)\ \omega\ \mathbf{end}}/_{self}\big]\#m :: \tau',$$

und wegen $\tau' \sim \tau$ folgt schliesslich mit Typregel (EQUIV) die Behauptung.

Die übrigen Fälle verlaufen analog. □

Damit ist sichergestellt, dass auch im neuen Typsystem mit rekursiven Typen die small step Semantik typerhaltend ist. Zum Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_o^{rt}$ fehlt damit nur noch das Progress-Theorem. Dazu formulieren wir zunächst eine entsprechende Version des *Canonical Forms Lemma* für die Sprache $\mathcal{L}_o^{rt}$. Wie im Fall der Sprache $\mathcal{L}_o^{sub}$ (siehe Lemma 3.9) beschränken wir uns auf die Aussagen, die wir im Beweis des nachfolgenden Progress-Theorem benutzen werden.

Lemma 4.7 (Canonical Forms) *Sei $v \in Val$, $\tau \in Type$, und gelte $[\,] \rhd v :: \tau$.*

*(a) Wenn $\tau \sim$ **int**, dann gilt $v \in Int$.*

(b) Wenn $\tau \sim \tau_1 \to \tau_2$, dann gilt eine der folgenden Aussagen:

1.) $v \in Op$

2.) $v = op\, v_1$ mit $op \in Op$ und $v_1 \in Val$

3.) $v = \lambda x : \tau_1'.e$ mit $x \in Var$, $\tau_1' \in Type$ und $e \in Exp$

*(c) Wenn $\tau \sim \langle\phi\rangle$, dann gilt $v =$ **object** $(self : \tau')\ \omega$ **end**.*

Beweis: Die Beweise der einzelnen Teile des Lemmas erfolgen jeweils durch vollständige Induktion über die Länge der Herleitung des Typurteils mit Fallunterscheidung nach der letzten Typregel in der Herleitung. Wir betrachten lediglich den Beweis des ersten Teils des Lemmas.

(a) Das Typurteil $[\,] \rhd v :: \tau$ mit $\tau \sim$ **int** kann nur mit den Typregeln (CONST) und (EQUIV) hergeleitet worden sein, da sich mit den übrigen, auf Werte anwendbaren Typregeln, wie zum Beispiel (ABSTR), kein zu **int** äquivalenter Typ herleiten läßt. Wir haben also lediglich die Fälle für die (CONST)- und (EQUIV)-Regeln zu betrachten.

1.) Für $[\,] \triangleright v :: \tau$ mit (Const) gilt nach Voraussetzung $v :: \tau$ und insbesondere $v \in \mathit{Const}$. Die einzige in Frage kommende Regel hier ist (Int), denn von den möglichen Typen von Konstanten ist einzig **int** äquivalent zu **int**. Die Regel (Int) wiederum impliziert $v \in \mathit{Int}$, was zu zeigen war.

2.) Im Fall von (Equiv) als letzter Typregel folgt die Behauptung unmittelbar mit Induktionsvoraussetzung.

Die übrigen Teile des Lemmas lassen sich ähnlich einfach beweisen. □

Damit sind nun alle Voraussetzungen gegeben, um das Progress-Theorem formulieren und beweisen zu können. Der Beweis ist im wesentlichen identisch zum Beweis des Progress-Theorems für die Programmiersprache $\mathcal{L}_o^{sub}$ (Satz 3.2), da die wesentlichen Details, in denen sich die Sprachen unterscheiden, bereits in den vorangegangenen Lemmata behandelt werden.

Satz 4.2 (Existenz des Übergangsschritts, „Progress“)

(a) $\forall e \in \mathit{Exp}, \tau \in \mathit{Type} : [\,] \triangleright e :: \tau \Rightarrow (e \in \mathit{Val} \vee \exists e' \in \mathit{Exp} : e \to e')$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma^+ \triangleright r :: \phi \Rightarrow (r \in \mathit{RVal} \vee \exists r' \in \mathit{Row} : r \to r')$

Beweis: Es sollte keine große Überraschung darstellen, dass der Beweis durch simultane Induktion über die Länge der Herleitung der Typurteile $[\,] \triangleright e :: \tau$ und $\Gamma^+ \triangleright r :: \phi$ erfolgt, wobei jeweils nach der zuletzt angewandten Typregel unterschieden wird. Wir betrachten dazu die folgenden Fälle.

1.) Für (Const) und (Abstr) gilt $e \in \mathit{Val}$, und für (Empty) gilt $r \in \mathit{RVal}$.

2.) $[\,] \triangleright e :: \tau$ mit Typregel (Equiv).

Nach Voraussetzung existiert ein $\tau' \in \mathit{Type}$, so dass gilt $[\,] \triangleright e :: \tau'$ mit $\tau' \sim \tau$. Die Behauptung folgt dann unmittelbar mit Induktionsvoraussetzung.

3.) $[\,] \triangleright e_1\, e_2 :: \tau$ mit Typregel (App).

Dann muss gelten $[\,] \triangleright e_1 :: \tau_2 \to \tau$ und $[\,] \triangleright e_2 :: \tau_2$, und es ist nach der Form der Teilausdrücke e_1, e_2 zu unterscheiden.

Falls $e_1 \notin \mathit{Val}$, so existiert nach Induktionsvoraussetzung ein $e_1' \in \mathit{Exp}$ mit $e_1 \to e_1'$, und somit ein small step $e_1\, e_2 \to e_1'\, e_2$ mit small step Regel (App-Left).

Entsprechend existiert für $e_1 \in \mathit{Val}$ und $e_2 \notin \mathit{Val}$ ein small step mit Regel (App-Right).

Somit bleibt noch der Fall $e_1, e_2 \in \mathit{Val}$ zu betrachten. Nach Canonical Forms Lemma (4.7) sind wegen $[\,] \triangleright e_1 :: \tau_2 \to \tau$ und der Reflexivität von $\sim$ für e_1 die folgenden drei Fälle zu unterscheiden:

1.) $e_1 \in \mathit{Op}$

Dann gilt nach Definition 2.4 $e_1\, e_2 \in \mathit{Val}$.

2.) $e_1 = \mathit{op}\, v_1$ mit $\mathit{op} \in \mathit{Op}$ und $v_1 \in \mathit{Val}$

Nach Lemma 4.6 existiert ein $\tau_1 \in \mathit{Type}$ mit $[\,] \triangleright \mathit{op} :: \tau_1 \to \tau_2 \to \tau$ und $[\,] \triangleright v_1 :: \tau_1$. Nach Definition 2.17 sowie Definition 4.4 folgt $\tau_1 \sim \mathbf{int}$ und $\tau_2 \sim \mathbf{int}$, also $v_1, e_2 \in \mathit{Int}$ nach Lemma 4.7. Somit existiert ein small step $\mathit{op}\, v_1\, e_2 \to \mathit{op}^I(v_1, e_2)$ mit small step Regel (Op).

3.) Für $e_1 = \lambda x : \tau_2'.e_1'$ existiert ein small step $(\lambda x : \tau_2'.e_1')\, e_2 \to e_1'[{}^{e_2}/_{x}]$ mit Regel (Beta-V), da nach Voraussetzung $e_2 \in \mathit{Val}$.

4.) $[\,] \triangleright e_1 \# m :: \tau$ mit Typregel (Send).

Das kann nur aus $[\,] \triangleright e_1 :: \langle m : \tau;\, \phi \rangle$ folgen, und nach Induktionsvoraussetzung gilt entweder $e_1 \in \mathit{Val}$ oder es existiert ein $e_1' \in \mathit{Exp}$ mit $e_1 \to e_1'$.

Sei also $e_1 \in \mathit{Val}$, dann gilt nach Lemma 4.7, dass $e_1 = \mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end}$. Also existiert ein small step

$$(\mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end})\#m \to \omega[{}^{e_1}/_{\mathit{self}}]\#m$$

mit small step Regel (Send-Unfold).

Ist andererseits $e_1 \notin \mathit{Val}$, so existiert ein small step

$$e_1\#m \to e_1'\#m$$

mit small step Regel (Send-Eval).

5.) $\Gamma^+ \triangleright \mathbf{val}\ a = e';\ r' :: \phi$ mit Typregel (Attr).

Das Typurteil kann nur aus Prämissen der Form $(\Gamma^+)^\star \triangleright e' :: \tau'$ und $\Gamma^+ \triangleright r' :: \phi'$ folgen, wobei zu beachten gilt, dass $(\Gamma^+)^\star = [\,]$. Nach Induktionsvoraussetzung gilt dann entweder $e' \in \mathit{Val}$ oder es existiert ein $e'' \in \mathit{Exp}$ mit $e' \to e''$. Ebenso gilt entweder $r' \in \mathit{RVal}$ oder existiert ein $r'' \in \mathit{Row}$ mit $r' \to r''$.

Seien $e' \in \mathit{Val}$ und $r' \in \mathit{RVal}$, dann gilt $(\mathbf{val}\ a = e';\ r') \in \mathit{RVal}$ nach Definition 2.4.

Falls $e' \notin \mathit{Val}$, also $e' \to e''$, so existiert ein small step

$$\mathbf{val}\ a = e';\ r' \to \mathbf{val}\ a = e'';\ r'$$

mit small step Regel (Attr-Left).

Analog existiert für $e' \in \mathit{Val}$ und $r' \notin \mathit{RVal}$, also $r' \to r''$, ein small step

$$\mathbf{val}\ a = e';\ r' \to \mathbf{val}\ a = e';\ r''$$

mit Regel (Attr-Right).

Die restlichen Fälle verlaufen analog. □

Mit diesen Ergebnissen können wir nun wiederum leicht die Typsicherheit der Programmiersprache $\mathcal{L}_o^{rt}$ zeigen, da sich der Beweis des nachfolgenden Satzes analog zum Beweis des Safety-Theorems der Sprache $\mathcal{L}_o^t$ (Satz 2.7) unmittelbar aus dem Preservation- und Progress-Theorem ergibt.

Satz 4.3 (Typsicherheit, „Safety“) *Wenn* $[] \triangleright e :: \tau$*, dann bleibt die Berechnung für* e *nicht stecken.*

Beweis: Folgt analog wie für $\mathcal{L}_o^t$ aus den Preservation und Progress-Sätzen. □

TODO: Algorithmus!

4.2 Die Sprache $\mathcal{L}_o^{srt}$

Abschliessend wollen wir betrachten, in wie weit sich die beiden bisher betrachteten Erweiterungen des Typsystems – Subtyping und rekursive Typen – in einer Programmiersprache kombinieren lassen.

TODO: Mehr Prosa

Definition 4.6 Die Relationen $\lesssim_n$ seien wie folgt induktiv definiert

$$\begin{array}{rcl}
\lesssim_0 & = & \mathit{Type}^2 \\
\lesssim_{n+1} & = & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}(\tau) = \mathit{root}(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}\} \\
& \cup & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}(\tau) = \mathit{root}(\tau') = \to \\
& & \quad \wedge\, \mathit{child}_1(\tau') \lesssim_n \mathit{child}_1(\tau) \wedge \mathit{child}_2(\tau) \lesssim_n \mathit{child}_2(\tau')\} \\
& \cup & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}(\tau) = \langle m_1;\ldots;m_{k+l}\rangle \wedge \mathit{root}(\tau') = \langle m_1;\ldots;m_k\rangle \\
& & \quad \wedge \forall 1 \le i \le k :\ \mathit{child}_i(\tau) \lesssim_n \mathit{child}_i(\tau')\}
\end{array}$$

und die Relation $\lesssim$ ist durch

$$\lesssim \ = \ \bigcap_{n\in\mathbb{N}} \lesssim_n$$

definiert, also als Durchschnitt aller Relationen $\lesssim_n$.

TODO: Überleitung, Vergleich mit Minimal Typing Subtypregeln (Sm-Refl, Sm-Arrow und Sm-Object)

Lemma 4.8 *Die Relation $\lesssim$ ist eine Quasiordnung.*

Im Gegensatz zur Subtyprelation in Kapitel 3 (vgl. Definition 3.1) begnügen wir uns in diesem Fall mit einer Quasiordnung, da wir die Eigenschaft der Antisymmetrie nicht zwingend benötigen zum Beweis der Typsicherheit. Stattdessen formulieren wir später ein Lemma, welches den Zusammenhang zwischen den Relationen $\sim$ und $\lesssim$ darstellt. Diese Eigenschaft entspricht im weiteren Sinne einer Antisymmetrie für die Relation $\lesssim$.

Beweis: Analog zum Beweis von Lemma 4.1 genügt es zu zeigen, dass jede Relation $\lesssim_n$ eine Quasiordnung ist, denn der Schnitt von Quasiordnungen ist wiederum eine Quasiordnung. Dazu zeigen wir durch vollständige Induktion über n, dass alle $\lesssim_n$ reflexiv und transitiv sind.

- $n = 0$

  Da $\lesssim_0 = \mathit{Type}^2$ ist klar, dass $\lesssim_0$ reflexiv und transitiv ist.

- $n \rightsquigarrow n + 1$

  Nach Induktionsvoraussetzung ist $\lesssim_n$ reflexiv und transitiv. Damit zeigen wir, dass auch $\lesssim_{n+1}$ reflexiv und transitiv ist. Konkret zeigen wir lediglich die Transitivität von $\lesssim_{n+1}$, da die Reflexivität trivialerweise erfüllt ist.

  Seien also $(\tau_1, \tau_2) \in \lesssim_{n+1}$ und $(\tau_2, \tau_3) \in \lesssim_{n+1}$. Dann unterscheiden wir nach der Form von $\mathit{root}\,(\tau_2)$:

  1.) $\mathit{root}\,(\tau_2) \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$
  Dann gilt $\mathit{root}\,(\tau_1) = \mathit{root}\,(\tau_2) = \mathit{root}\,(\tau_3)$, also auch $(\tau_1, \tau_3) \in \lesssim_{n+1}$.

  2.) $\mathit{root}\,(\tau_2) = \rightarrow$
  Dann gilt wieder $\mathit{root}\,(\tau_1) = \mathit{root}\,(\tau_2) = \mathit{root}\,(\tau_3)$. Weiterhin gelten die folgenden Ungleichungen:

  $$\begin{array}{lcllcl} \mathit{child}_1\,(\tau_2) & \lesssim_n & \mathit{child}_1\,(\tau_1) & \mathit{child}_2\,(\tau_1) & \lesssim_n & \mathit{child}_2\,(\tau_2) \\ \mathit{child}_1\,(\tau_3) & \lesssim_n & \mathit{child}_1\,(\tau_2) & \mathit{child}_2\,(\tau_2) & \lesssim_n & \mathit{child}_2\,(\tau_3) \end{array}$$

  Wegen der nach Induktionsvoraussetzung geltenden Transitivität von $\lesssim_n$ folgt damit, dass auch die Ungleichungen

  $$\begin{array}{lcl} \mathit{child}_1\,(\tau_3) & \lesssim_n & \mathit{child}_1\,(\tau_1) \\ \mathit{child}_2\,(\tau_1) & \lesssim_n & \mathit{child}_2\,(\tau_3) \end{array}$$

  gelten, und somit $(\tau_1, \tau_3) \in \lesssim_{n+1}$ nach Definition 4.6.

3.) $root\,(\tau_2) = \langle m_1; \ldots; m_k \rangle$
Dann existieren für τ_1 und τ_3 nach Definition $j, k \in \mathbb{N}$ mit $j \leq k \leq l$, so dass $root\,(\tau_1) = \langle m_1; \ldots; m_l \rangle$ und $root\,(\tau_3) = \langle m_1; \ldots; m_j \rangle$. Weiterhin gilt

$$\forall 1 \leq i \leq k: \ child_i\,(\tau_1) \lesssim_n child_i\,(\tau_2)$$

und

$$\forall 1 \leq i \leq j: \ child_i\,(\tau_2) \lesssim_n child_i\,(\tau_3).$$

Da $j \leq k$ und $\lesssim_n$ als transitiv angenommen wurde, folgt

$$\forall 1 \leq i \leq j: \ child_i\,(\tau_1) \lesssim_n child_i\,(\tau_3).$$

und somit unmittelbar $(\tau_1, \tau_3) \in \lesssim_{n+1}$.

Damit ist gezeigt, dass die Menge *Type* durch die Relation $\lesssim$, sowie die Relationen $\lesssim_n$, quasi geordnet ist. □

Damit kommen wir nun zum bereits angesprochenen Lemma über den Zusammenhang zwischen Typgleichheit und Subtyping bei rekursiven Typen. Wir werden zeigen, dass die Relation $\lesssim$ bezüglich der Relation $\sim$ antisymmetrisch ist.

Lemma 4.9 $\forall \tau, \tau' \in Type: \ \tau \lesssim \tau' \wedge \tau' \lesssim \tau \ \Leftrightarrow \ \tau \sim \tau'$

Beweis: Es genügt zu zeigen, dass die Aussage

$$\forall \tau, \tau' \in Type: \ \tau \lesssim_n \tau' \wedge \tau' \lesssim_n \tau \ \Leftrightarrow \ \tau \sim_n \tau'$$

für alle $n \in \mathbb{N}$ gilt, da die Relationen $\sim$ und $\lesssim$ beide als Schnitt der induktiv definierten Relationen $\sim_n$ und $\lesssim_n$ definiert sind.

„$\Leftarrow$“ Wir beweisen die Behauptung durch vollständige Induktion über n.

- $n = 0$
  Klar, denn es gilt $\sim_0 = Type^2 = \lesssim_0$.
- $n \rightsquigarrow n + 1$
  Für $\tau \sim_{n+1} \tau'$ muss nach Voraussetzung

  $$root\,(\tau) = root\,(\tau')$$

  und

  $$\forall 1 \leq i \leq arity\,(\tau): \ child_i\,(\tau) \sim_n child_i\,(\tau')$$

gelten. Nach Induktionsvoraussetzung folgt daraus, dass

$$\forall 1 \leq i \leq \mathit{arity}\,(\tau):\ \mathit{child}_i\,(\tau) \lesssim_n \mathit{child}_i\,(\tau') \wedge \mathit{child}_i\,(\tau') \lesssim_n \mathit{child}_i\,(\tau).$$

gilt. Damit folgt insgesamt die Behauptung mit einer einfachen Fallunterscheidung nach der Form von $\mathit{root}\,(\tau)$ und Anwenden der Definition von $\lesssim_{n+1}$ (4.6).

„⇒“ **TODO:** Ebenfalls Induktion über n □

Genaugenommen ist das Lemma sogar eine stärkere Aussage. Für die eigentliche Antisymmetrie würde eine Implikation, wie im folgenden Korollar dargestellt, genügen.

Korollar 4.2 $\forall \tau, \tau' \in \mathit{Type}:\ \tau \lesssim \tau' \wedge \tau' \lesssim \tau \ \Rightarrow\ \tau \sim \tau'$

Beweis: Folgt direkt aus Lemma 4.9. □

TODO: Prosa

Definition 4.7 Sei $\mathcal{S} : \mathit{TypeRel} \to \mathit{TypeRel}$ die wie folgt definierte totale Abbildung:

$$\begin{aligned}
\mathcal{S}(R) \;=\;& \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \mathit{root}\,(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}\} \\
\cup\;& \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \mathit{root}\,(\tau') = \to \\
& \quad \wedge\, (\mathit{child}_1\,(\tau'), \mathit{child}_1\,(\tau)) \in R \wedge\, (\mathit{child}_2\,(\tau), \mathit{child}_2\,(\tau')) \in R\} \\
\cup\;& \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \langle m_1; \ldots; m_{k+l}\rangle \wedge\, \mathit{root}\,(\tau') = \langle m_1; \ldots; m_k\rangle \\
& \quad \wedge \forall 1 \leq i \leq k:\ (\mathit{child}_i\,(\tau), \mathit{child}_i\,(\tau')) \in R\},
\end{aligned}$$

wobei *TypeRel* die Menge aller binären Relationen auf *Type*, also $\mathit{TypeRel} = \mathcal{P}(\mathit{Type}^2)$, bezeichnet.

Lemma 4.10

(a) $(\mathit{TypeRel}, \supseteq)$ *ist ein semantischer Bereich mit kleinstem Element* $\mathit{Type} \times \mathit{Type}$.

(b) $\mathcal{S}$ *ist eine stetige Abbildung auf* $(\mathit{TypeRel}, \supseteq)$.

Beweis:

(a) Klar, denn für jede Folge $R_0 \supseteq R_1 \supseteq R_2 \supseteq \ldots$ mit $R_n \in \mathit{TypeRel}$ für $n \in \mathbb{N}$ existiert mit $\bigcap_{n\in\mathbb{N}} R_n$ ein Supremum, und $\mathit{Type} \times \mathit{Type}$ ist bezüglich $\supseteq$ das kleinste Element[2] in *TypeRel*.

[2] Natürlich ist $\mathit{Type} \times \mathit{Type}$ intuitiv das größte Element in *TypeRel*, aber zur Vermeidung von Verwirrung halten wir uns hier strikt an die Termini aus Abschnitt 1.1.1.

(b) Sei $R_0 \supseteq R_1 \supseteq R_2 \supseteq \ldots$ eine nicht-leere Folge in *TypeRel*, also $R_n \in \mathit{TypeRel}$ für $n \in \mathbb{N}$. Gemäß Definition 1.5 haben wir zu zeigen, dass das Supremum von $\{\mathcal{S}(R_n) \mid n \in \mathbb{N}\}$ existiert und $\bigsqcup\{\mathcal{S}(R_n) \mid n \in \mathbb{N}\} = \mathcal{S}\big(\bigsqcup\{R_n \mid n \in \mathbb{N}\}\big)$ gilt.

Es ist ziemlich offensichtlich, dass $\bigcap_{n\in\mathbb{N}} S(R_n)$ bezüglich $\supseteq$ die kleinste obere Schranke von $\{\mathcal{S}(R_n) \mid n \in \mathbb{N}\}$ ist.

Es bleibt zu zeigen, dass die Abbildung $\mathcal{S}$ das Supremum erhält, das bedeutet wir haben zu zeigen, dass $\bigcap_{n\in\mathbb{N}} \mathcal{S}(R_n) = \mathcal{S}(\bigcap_{n\in\mathbb{N}} R_n)$ gilt:

„$\supseteq$“ Sei $(\tau, \tau') \in \mathcal{S}(\bigcap_{n\in\mathbb{N}} R_n)$, dann gilt einer der folgenden Fälle:

1.) $\mathit{root}(\tau) = \mathit{root}(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$
In diesem Fall gilt $(\tau, \tau') \in \mathcal{S}(R)$ für alle $R \in \mathit{TypeRel}$. Also gilt insbesondere $(\tau, \tau') \in \bigcap_{n\in\mathbb{N}} \mathcal{S}(R_n)$.

2.) $\mathit{root}(\tau) = \mathit{root}(\tau') = \rightarrow$
Seien $\tau_i = \mathit{child}_i(\tau)$ und $\tau_i' = \mathit{child}_i(\tau')$ für $i = 1, 2$. Nach Definition 4.7 gilt $(\tau_1', \tau_1), (\tau_2, \tau_2') \in \bigcap_{n\in\mathbb{N}} R_n$. Das bedeutet $(\tau_1', \tau_1), (\tau_2, \tau_2') \in R_n$ für alle $n \in \mathbb{N}$. Also folgt wiederum nach Definition von $\mathcal{S}$, dass $(\tau, \tau') \in \mathcal{S}(R_n)$ für alle $n \in \mathbb{N}$, und somit $(\tau, \tau') \in \bigcap_{n\in\mathbb{N}} \mathcal{S}(R_n)$.

3.) $\mathit{root}(\tau) = \langle m_1; \ldots; m_{k+l} \rangle$, $\mathit{root}(\tau') = \langle m_1; \ldots; m_k \rangle$
Folgt analog.

„$\subseteq$“ Sei $(\tau, \tau') \in \bigcap_{n\in\mathbb{N}} \mathcal{S}(R_n)$, also $(\tau, \tau') \in \mathcal{S}(R_n)$ für alle $n \in \mathbb{N}$. Dann sind wieder die folgenden Fälle zu unterscheiden.

1.) $\mathit{root}(\tau) = \mathit{root}(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$
Klar.

2.) $\mathit{root}(\tau) = \mathit{root}(\tau') = \rightarrow$
Seien wieder $\tau_i = \mathit{child}_i(\tau)$ und $\tau_i' = \mathit{child}_i(\tau')$ für $i = 1, 2$, dann gilt $(\tau_1', \tau_1), (\tau_2, \tau_2') \in R_n$ nach Definition von $\mathcal{S}$. Das bedeutet weiter, dass $(\tau_1', \tau_1), (\tau_2, \tau_2') \in \bigcap_{n\in\mathbb{N}} R_n$. Die Behauptung folgt dann unmittelbar nach Definition 4.7.

3.) $\mathit{root}(\tau) = \langle m_1; \ldots; m_{k+l} \rangle$, $\mathit{root}(\tau') = \langle m_1; \ldots; m_k \rangle$
Folgt analog. □

Damit läßt sich nun das zentrale Lemma über die Relation $\lesssim$ formulieren und beweisen. Der Leser sei an dieser Stelle nochmals auf Abschnitt 1.1.1 verwiesen, in dem die notwendigen mathematischen Grundlagen zur Fixpunkttheorie dargestellt werden.

Lemma 4.11 (Fixpunkt-Lemma) *$\lesssim$ ist der kleinste Fixpunkt von $\mathcal{S}$.*

Beweis: Nach Lemma 4.10 ist $(\mathit{TypeRel}, \supseteq)$ ein semantischer Bereich und die totale Abbildung $\mathcal{S}$ ist stetig. Damit läßt sich der Fixpunktsatz von Kleene (Satz 1.1) anwenden, nach dem das Supremum $\bigsqcup_{n\in\mathbb{N}} \mathcal{S}^n(\mathit{Type} \times \mathit{Type})$ existiert und kleinster Fixpunkt von $\mathcal{S}$ ist. Es bleibt zu zeigen, dass $\lesssim = \bigsqcup_{n\in\mathbb{N}} \mathcal{S}^n(\mathit{Type} \times \mathit{Type})$.

Dazu zeigen wir zunächst durch vollständige Induktion, dass $\lesssim_n = \mathcal{S}^n(\mathit{Type} \times \mathit{Type})$ gilt:

- $n = 0$

  Klar, da $\lesssim_0 = \mathit{Type} \times \mathit{Type} = \mathcal{S}^0(\mathit{Type} \times \mathit{Type})$.

- $n \rightsquigarrow n + 1$

  Nach Induktionsvoraussetzung gilt $\lesssim_n = \mathcal{S}^n(\mathit{Type} \times \mathit{Type})$. Damit folgt trivialerweise die Behauptung, da die Definition von $\lesssim_{n+1}$ unter dieser Voraussetzung mit der Definition von $\mathcal{S}$ übereinstimmt.

Damit erhalten wir $\lesssim = \bigcap_{n\in\mathbb{N}} \mathcal{S}^n(\mathit{Type} \times \mathit{Type}) = \bigsqcup_{n\in\mathbb{N}} \mathcal{S}^n(\mathit{Type} \times \mathit{Type})$, was zu zeigen war. □

Korollar 4.3 *$\lesssim$ ist bezüglich $\subseteq$ der größte Fixpunkt von $\mathcal{S}$.*

Beweis: Klar. □

Das vorangegangene Fixpunkt-Lemma läßt sich auch in einer vertrauteren Form darstellen, die wir bereits für die Programmiersprache $\mathcal{L}_o^{sub}$ kennengelernt haben (siehe Lemma 3.3).

Lemma 4.12 (Subtyping-Lemma) *$\tau \lesssim \tau'$ gilt genau dann, wenn eine der folgenden Aussagen zutrifft:*

(a) $\tau \sim \tau' \sim \tau_\beta$ *mit* $\tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

(b) $\tau \sim \tau_1 \to \tau_2$ *und* $\tau' \sim \tau_1' \to \tau_2'$ *mit* $\tau_1' \lesssim \tau_1$ *und* $\tau_2 \lesssim \tau_2'$

(c) $\tau \sim \langle m_1 : \tau_1; \ldots; m_{k+l} : \tau_{k+l}\rangle$ *und* $\tau' \sim \langle m_1 : \tau_1'; \ldots; m_k : \tau_k'\rangle$ *mit* $\tau_i \lesssim \tau_i'$ *für alle* $i = 1, \ldots, k$

Beweis: Folgt unmittelbar aus der Tatsache, dass $\lesssim$ Fixpunkt von $\mathcal{S}$ ist (Lemma 4.11), denn die Aussagen des Lemmas entsprechen der Definition von $\mathcal{S}$. □

Bevor wir nun die Typsicherheit der Programmiersprache $\mathcal{L}_o^{srt}$ betrachten, müssen wir noch die Typregeln für das Typsystem festlegen. Im wesentlichen handelt es sich beim Typsystem von $\mathcal{L}_o^{srt}$ um eine Kombination der Typsysteme der Programmiersprachen $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$. Entsprechend sieht die Definition der gültigen Typurteile für $\mathcal{L}_o^{srt}$ bekannt aus:

Definition 4.8 (Gültige Typurteile für $\mathcal{L}_o^{srt}$) Ein Typurteil $\Gamma \rhd_e e :: \tau$ oder $\Gamma \rhd_r r :: \phi$ heisst *gültig* für $\mathcal{L}_o^{srt}$, wenn es sich mit den Typregeln der Programmiersprache $\mathcal{L}_o^{rt}$ aus Definition 4.5 mit Ausnahme der (EQUIV)-Regel, sowie der folgenden neuen Typregel

$$(\text{SUBSUME'}) \quad \frac{\Gamma \rhd_e e :: \tau' \quad \tau' \lesssim \tau}{\Gamma \rhd_e e :: \tau}$$

herleiten lässt.

4.2.1 Typsicherheit

Intuitiv lässt sich bereits vermuten, dass die Programmiersprache $\mathcal{L}_o^{srt}$ ebenfalls typsicher ist, da das Typsystem im wesentlichen durch Zusammenführung der Typsysteme der Programmiersprachen $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$ entstanden ist. Allerdings ist diese Eigenschaft deshalb noch nicht zwingend trivial ersichtlich, denn es existieren Programme, die zwar in $\mathcal{L}_o^{srt}$ wohlgetypt sind, jedoch weder in $\mathcal{L}_o^{sub}$ noch in $\mathcal{L}_o^{rt}$.

Also beweisen wir in diesem Abschnitt zunächst wiederum die Preservation- und Progress-Sätze, für die Programmiersprache $\mathcal{L}_o^{srt}$, womit dann gezeigt wäre, dass auch die kombinierte Programmiersprache noch immer typsicher ist. Dazu beginnen wir wie gehabt mit einigen technischen Lemmata, wobei wir die Beweise, sofern sie keine neuen Erkenntnisse zu Tage fördern, im wesentlichen überspringen.

Lemma 4.13 (Typumgebungen und frei vorkommende Namen)

(a) $\forall \Gamma \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma \rhd e :: \tau \Rightarrow \mathit{free}(e) \subseteq \mathit{dom}(\Gamma)$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma \rhd r :: \phi \Rightarrow \mathit{free}(r) \subseteq \mathit{dom}(\Gamma)$

Beweis: Wie zuvor verläuft der Beweis durch Induktion über die Länge der Herleitung der Typurteile. Die bekannten Fälle folgen wie gehabt, der neue Fall der (SUBSUME')-Regel folgt unmittelbar mit Induktionsvoraussetzung. □

Lemma 4.14 (Koinzidenzlemma)

(a) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma_1 \rhd e :: \tau \wedge \Gamma_1 =_{\mathit{free}(e)} \Gamma_2 \Rightarrow \Gamma_2 \rhd e :: \tau$

(b) $\forall \Gamma_1, \Gamma_2 \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma_1 \rhd r :: \phi \wedge \Gamma_1 =_{\mathit{free}(r)} \Gamma_2 \Rightarrow \Gamma_2 \rhd r :: \phi$

Beweis: Ebenfalls wie gehabt, wobei der Fall der (SUBSUME')-Regel direkt mit Induktionsvoraussetzung folgt. □

Wie zuvor benötigen wir wieder zwei Substitutionslemmata, ein allgemeines für Attribute und Variablen, und ein spezielles für die *self*-Substitution.

Lemma 4.15 (Typurteile und Substitution) *Sei* $id \in Attribute \cup Var$, $\Gamma \in TEnv$, $\tau \in Type$ *und* $e \in Exp$. *Dann gilt:*

(a) $\forall e' \in Exp : \forall \tau' \in Type : \Gamma[^{\tau}/_{id}] \rhd e' :: \tau' \wedge \Gamma^{\star} \rhd e :: \tau \Rightarrow \Gamma \rhd e'[^{e}/_{id}] :: \tau'$

(b) $\forall r \in Row : \forall \phi \in RType : \Gamma[^{\tau}/_{id}] \rhd r :: \phi \wedge \Gamma^{\star} \rhd e :: \tau \Rightarrow \Gamma \rhd r[^{e}/_{id}] :: \phi$

Beweis: Ähnlich wie im Beweis des Lemmas für die Programmiersprache $\mathcal{L}_o^{rt}$ (4.4), wobei der Fall der (SUBSUME')-Regel ebenfalls trivialerweise mit Induktionsvoraussetzung folgt. □

Lemma 4.16 (Typurteile und *self*-Substitution) *Sei* $\Gamma \in TEnv$, $self \in Self$, $\tau \in Type$ *und* $r \in Row$. *Dann gilt:*

(a) Für alle $e \in Exp$ *und* $\tau' \in Type$*:*

(1) $\Gamma[^{\tau}/_{self}] \rhd e :: \tau'$

(2) $\Gamma^{\star} \rhd \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau$

(3) $\forall a \in dom\,(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^{\star} \rhd r(a) :: \tau_a \Leftrightarrow \Gamma \rhd a :: \tau_a$

$\Rightarrow \Gamma \rhd e[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau'$

(b) Für alle $r' \in Row$ *und* $\phi \in RType$*:*

(1) $\Gamma[^{\tau}/_{self}] \rhd r' :: \phi$

(2) $\Gamma^{\star} \rhd \mathbf{object}\,(self : \tau)\ r \oplus r'\ \mathbf{end} :: \tau$

(3) $\forall a \in dom\,(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^{\star} \rhd r(a) :: \tau_a \Leftrightarrow \Gamma \rhd a :: \tau_a$

$\Rightarrow \Gamma \rhd r'[^{\mathbf{object}\,(self:\tau)\ r\oplus r'\ \mathbf{end}}/_{self}] :: \phi$

Beweis: Der Beweis entspricht ebenfalls im wesentlichen den Beweisen des Lemmas für zuvor behandelte Programmiersprachen. Wie früher folgt gemäß Lemma 4.13, dass die Substitution in jedem Fall definiert ist. Der eigentliche Beweis erfolgt dann durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[^{\tau}/_{self}] \rhd e :: \tau'$ und $\Gamma[^{\tau}/_{self}] \rhd r' :: \phi$, mit Fallunterscheidung nach der zuletzt angewendeten Typregel in der Herleitung. Wir betrachten dazu lediglich den Fall der neuen Typregel (SUBSUME').

1.) $\Gamma[^{\tau}/_{self}] \triangleright e :: \tau'$ mit Typregel (SUBSUME').

Dann existiert nach Voraussetzung ein $\tau'' \in \mathit{Type}$ mit $\Gamma[^{\tau}/_{self}] \triangleright e :: \tau''$ und $\tau'' \lesssim \tau'$. Darauf lässt sich unmittelbar die Induktionsvoraussetzung anwenden, und es gilt

$$\Gamma \triangleright e[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau''.$$

Wegen $\tau'' \lesssim \tau'$ folgt dann direkt die Behauptung

$$\Gamma \triangleright e[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau'$$

mit Typregel (SUBSUME').

Die übrigen Fälle verlaufen wie in den früheren Beweisen. □

Wie zuvor für $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$ benötigen wir zum Beweis der Preservation- und Progress-Theoreme neben den obigen Lemmata noch ein weiteres Lemmas, welches im wesentlichen einer Umkehrung der Typrelation entspricht. Wie gehabt beschränken wir uns auf die Aussagen, die in den späteren Beweisen benutzt werden, anstatt das Lemma in seiner vollständigen Form anzugeben.

Lemma 4.17 (Umkehrung der Typrelation)

(a) Wenn $\Gamma \triangleright op :: \tau$, *dann gilt* $\tau \sim \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ *für* $op \in \{+, -, *\}$ *oder* $\tau \sim \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ *für* $op \in \{\leq, \geq, <, >, =\}$.

(b) Wenn $\Gamma \triangleright \lambda x : \tau_1.e :: \tau$, *dann gilt* $\Gamma[^{\tau_1}/_{x}] \triangleright e :: \tau_2$ *und* $\tau_1 \to \tau_2 \lesssim \tau$.

(c) Wenn $\Gamma \triangleright e_1\, e_2 :: \tau$, *dann gilt* $\Gamma \triangleright e_1 :: \tau_2' \to \tau$, $\Gamma \triangleright e_2 :: \tau_2$ *und* $\tau_2 \lesssim \tau_2'$.

(d) Wenn $\Gamma \triangleright \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau'$, *dann gilt* $\Gamma^{\star}[^{\tau}/_{self}] \triangleright r :: \phi$ *mit* $\langle\phi\rangle \sim \tau$ *und* $\tau \lesssim \tau'$.

Beweis: Wir beweisen die einzelnen Aussagen jeweils durch Induktion über die Länge der Typherleitung, mit Fallunterscheidung nach der zuletzt angewandten Typregel in der Herleitung. In jedem Fall kommen wieder nur exakt zwei Typregeln in Frage.

(a) $\Gamma \triangleright op :: \tau$ kann nur mit einer der Typregeln (CONST) oder (SUBSUME') hergeleitet worden sein:

1.) Falls $\Gamma \triangleright op :: \tau$ mit Typregel (CONST) hergeleitet wurde, folgt die Behauptung unmittelbar nach Definition, denn $\sim$ ist reflexiv.

2.) Sollte die Herleitung von $\Gamma \triangleright op :: \tau$ mit einer Anwendung von Typregel (SUBSUME') enden, so existiert nach Voraussetzung ein $\tau' \in \mathit{Type}$ mit $\Gamma \triangleright op :: \tau'$ und $\tau' \lesssim \tau$. Nach Induktionsvoraussetzung gilt $\tau' \sim \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ für $op \in \{+, -, *\}$ oder $\tau' \sim \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ für $op \in \{\leq, \geq, <, >, =\}$.
Wir betrachten ausschliesslich den Fall $\tau' \sim \mathbf{int} \to \mathbf{int} \to \mathbf{int}$, der Fall $\tau' \sim \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ folgt dann analog. Wegen Lemma 4.9 folgt $\mathbf{int} \to \mathbf{int} \to \mathbf{int} \lesssim \tau'$ und wegen der Transitivität von $\lesssim$ folgt $\mathbf{int} \to \mathbf{int} \to \mathbf{int} \lesssim \tau$. Nach Subtyping-Lemma (4.12) existeren dann $\tau_1, \tau_2 \in \mathit{Type}$ mit $\tau \sim \tau_1 \to \tau_2$, $\tau_1 \lesssim \mathbf{int}$ und $\mathbf{int} \to \mathbf{int} \lesssim \tau_2$. Nach erneuter Anwendung des Lemmas erhalten wir $\tau_2', \tau_2'' \in \mathit{Type}$ mit $\tau_2' \lesssim \mathbf{int}$ und $\mathbf{int} \lesssim \tau_2''$. Wiederum mit Lemma 4.12 folgt schliesslich $\tau_1 \sim \tau_2' \sim \tau_2'' \sim \mathbf{int}$. Somit gilt schlussendlich $\tau \sim \mathbf{int} \to \mathbf{int} \to \mathbf{int}$, was zu zeigen war.

(b) Das Typurteil $\Gamma \triangleright \lambda x : \tau_1.e :: \tau$ läßt sich nur mit einer der Typregeln (ABSTR) oder (SUBSUME') herleiten:

1.) Im Fall von $\Gamma \triangleright \lambda x : \tau_1.e :: \tau$ mit Typregel (ABSTR) gilt nach Voraussetzung $\Gamma[^{\tau_1}/_x] \triangleright e :: \tau_2$ und $\tau = \tau_1 \to \tau_2$. Da $\lesssim$ reflexiv ist, folgt $\tau_1 \to \tau_2 \lesssim \tau$.

2.) Falls $\Gamma \triangleright \lambda x : \tau_1.e :: \tau$ mit Typregel (SUBSUME') hergeleitet worden ist, existiert nach Voraussetzung ein $\tau' \in \mathit{Type}$ mit $\Gamma \triangleright \lambda x : \tau_1.e :: \tau'$ und $\tau' \lesssim \tau$. Nach Induktionsvoraussetzung gilt also $\Gamma[^{\tau_1}/_x] \triangleright e :: \tau_2$ mit $\tau_1 \to \tau_2 \lesssim \tau'$. Wegen der Transitivität von $\lesssim$ folgt damit $\tau_1 \to \tau_2 \lesssim \tau$, was zu zeigen war.

(c) Die Herleitung von $\Gamma \triangleright e_1\, e_2 :: \tau$ kann nur mit einer Anwendung einer der Typregeln (APP) oder (SUBSUME') enden, entsprechend sind nur diese Fälle zu betrachten:

1.) Für $\Gamma \triangleright e_1\, e_2 :: \tau$ mit Typregel (APP) muss gelten $\Gamma \triangleright e_1 :: \tau_2 \to \tau$ und $\Gamma \triangleright e_2 :: \tau_2$. Da $\lesssim$ reflexiv ist, gilt $\tau_2 \lesssim \tau_2$.

2.) $\Gamma \triangleright e_1\, e_2 :: \tau$ mit Typregel (SUBSUME') kann nur aus Prämissen der Form $\Gamma \triangleright e_1\, e_2 :: \tau'$ und $\tau' \lesssim \tau$ folgen. Nach Induktionsvoraussetzung gilt dann $\Gamma \triangleright e_1 :: \tau_2' \to \tau'$, $\Gamma \triangleright e_2 :: \tau_2$ und $\tau_2 \lesssim \tau_2'$. Wegen $\tau' \lesssim \tau$ folgt somit schliesslich $\Gamma \triangleright e_1 :: \tau_2' \to \tau$ mit Typregel (SUBSUME') aus $\Gamma \triangleright e_1 :: \tau_2' \to \tau'$.

(d) Wiederum ist klar, dass das Typurteil $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ ausschliesslich mit den Typregeln (OBJECT') und (SUBSUME') hergeleitet worden sein kann, und somit nur diese Fälle zu betrachten sind:

1.) Falls $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ mit einer Anwendung der Typregel (OBJECT') hergeleitet wurde, so gilt nach Voraussetzung $\Gamma^\star[^{\tau}/_{\mathit{self}}] \triangleright r :: \phi$ mit $\langle\phi\rangle \sim \tau$ und $\tau = \tau'$, woraus unmittelbar $\tau \lesssim \tau'$ folgt.

2.) Sollte andererseits $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ mit Typregel (SUBSUME') hergeleitet worden sein, so existiert nach Voraussetzung ein $\tau'' \in \mathit{Type}$ mit $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau''$ und $\tau'' \lesssim \tau'$. Darauf lässt sich nun die Induktionsvoraussetzung anwenden und wir erhalten $\Gamma^\star[{}^{\tau}/_{\mathit{self}}] \triangleright r :: \phi$ mit $\langle\phi\rangle \sim \tau$ und $\tau \lesssim \tau''$. Wegen der Transitivität von $\lesssim$ folgt schliesslich $\tau \lesssim \tau'$.□

Wie bereits im Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$ können wir dank des vorangegangenen Lemmas die Anwendungen der Typregel (SUBSUME') im wesentlichen ignorieren, und somit das Preservation-Theorem relativ einfach beweisen. Die Formulierung des Satzes ist, wie nicht anders zu erwarten, die gleiche wie zuvor.

Satz 4.4 (Typerhaltung, „Preservation“)

(a) $\forall \Gamma \in \mathit{TEnv} : \forall e, e' \in \mathit{Exp} : \forall \tau \in \mathit{Type} : \Gamma^\star \triangleright e :: \tau \ \wedge\ e \to e' \ \Rightarrow\ \Gamma^\star \triangleright e' :: \tau$

(b) $\forall \Gamma \in \mathit{TEnv} : \forall r, r' \in \mathit{Row} : \forall \phi \in \mathit{RType} : \Gamma \triangleright r :: \phi \ \wedge\ r \to r' \ \Rightarrow\ \Gamma \triangleright r' :: \phi$

Beweis: Der Beweis erfolgt selbstverständlich wieder durch simultane Induktion über die Länge der Typherleitungen von $\Gamma^\star \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi$, mit Fallunterscheidung nach der zuletzt angewendeten Typregel. Wir betrachten lediglich die folgenden Fälle.

1.) Für (CONST), (ID), (ABSTR) und (EMPTY) gilt gemäß Lemma 2.3 $e \not\to$ bzw. $r \not\to$.

2.) $\Gamma^\star \triangleright e :: \tau$ mit Typregel (SUBSUME').

Dann gilt nach Voraussetzung, dass ein $\tau' \in \mathit{Type}$ existiert, mit $\Gamma^\star \triangleright e :: \tau'$ und $\tau' \lesssim \tau$. Nach Induktionsvoraussetzung folgt

$$\Gamma^\star \triangleright e' :: \tau',$$

und wegen $\tau' \lesssim \tau$ folgt daraus mit Anwendung von Typregel (SUBSUME') unmittelbar

$$\Gamma^\star \triangleright e' :: \tau,$$

was zu zeigen war.

3.) $\Gamma^\star \triangleright e_1\, e_2 :: \tau$ mit Typregel (APP).

Dann muss gelten $\Gamma^\star \triangleright e_1 :: \tau_2 \to \tau$ und $\Gamma^\star \triangleright e_2 :: \tau_2$. Der small step $e_1\, e_2 \to e'$ kann wiederum nur mit einer der small step Regel (APP-LEFT), (APP-RIGHT), (BETA-V) oder (OP) hergeleitet worden sein. Entsprechend unterscheiden wir nach der letzten Regel in der Herleitung des small steps:

1.) $e_1\, e_2 \to e'$ mit (App-Left) impliziert $e' = e_1'\, e_2$ und $e_1 \to e_1'$. Darauf lässt sich die Induktionsvoraussetzung anwenden, und wir erhalten

$$\Gamma^\star \triangleright e_1' :: \tau_2 \to \tau,$$

woraus dann mit Typregel (App) das erwartete Ergebnis

$$\Gamma^\star \triangleright e_1'\, e_2 :: \tau$$

folgt.

2.) $e_1\, e_2 \to e'$ mit small step Regel (App-Right) kann nur aus $e_1 \in \mathit{Val}$, $e_2 \to e_2'$ und $e' = e_1\, e_2'$ folgen. Die Behauptung folgt dann mit Induktionsvoraussetzung und Typregel (App).

3.) Falls $e_1\, e_2 \to e'$ mit (Beta-V) hergeleitet wurde, gilt $e_1 = \lambda x : \tau_1.e_1'$ und $e_2 \in \mathit{Val}$, sowie $e' = e_1'[{}^{e_2}/_x]$. Nach dem vorangegangenen Lemma (4.17) gilt also

$$\Gamma^\star[{}^{\tau_1}/_x] \triangleright e_1' :: \tau'$$

mit $\tau_1 \to \tau' \lesssim \tau_2 \to \tau$. Nach Subtyping-Lemma (4.12) gilt also $\tau_2 \lesssim \tau_1$ und $\tau' \lesssim \tau$, das bedeutet wegen $\Gamma^\star \triangleright e_2 :: \tau_2$ folgt mit Typregel (Subsume'), dass auch

$$\Gamma^\star \triangleright e_2 :: \tau_1$$

gilt. Damit lässt sich nun auf $\Gamma^\star[{}^{\tau_1}/_x] \triangleright e_1' :: \tau'$ das Substitutions-Lemma (4.15) anwenden, und wir erhalten

$$\Gamma^\star \triangleright e_1'[{}^{e_2}/_x] :: \tau',$$

woraus dann wegen $\tau' \lesssim \tau$ mit Typregel (Subsume') schliesslich

$$\Gamma^\star \triangleright e_1'[{}^{e_2}/_x] :: \tau$$

folgt, was zu zeigen war.

4.) Für $e_1\, e_2 \to e'$ mit small step Regel (Op) muss gelten $e_1 = op\, v_1$, $e_2 = v_2 \in \mathit{Val}$ und $e' = op^I(v_1, v_2)$. Gemäß Lemma 4.17 folgt daraus, dass

$$\Gamma^\star \triangleright op :: \tau_1 \to \tau_2 \to \tau$$

und

$$\Gamma^\star \triangleright v_1 :: \tau_1$$

gilt. Weiterhin folgt nach Lemma 4.17, dass $\tau_1 \sim \mathbf{int}$, $\tau_2 \sim \mathbf{int}$, und $\tau \sim \mathbf{int}$ für $op \in \{+, -, *\}$ oder $\tau \sim \mathbf{bool}$ für $op \in \{\leq, \geq, <, >, =\}$.

Wir betrachten wieder nur den Fall, dass $op \in \{+, -, *\}$. Für relationale Operatoren folgt die Behauptung analog.
Sei also $op \in \{+, -, *\}$, dann gilt nach Definition $op^I(v_1, v_2) \in Int$. Daraus folgt mit Typregel (CONST), dass

$$\Gamma^\star \triangleright op^I(v_1, v_2) :: \mathbf{int}$$

gilt, und wegen $\tau \sim \mathbf{int}$ gilt nach Lemma 4.9 auch $\mathbf{int} \lesssim \tau$, also folgt schliesslich

$$\Gamma^\star \triangleright op^I(v_1, v_2) :: \tau$$

mit Typregel (SUBSUME').

4.) $\Gamma^\star \triangleright \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau$ mit Typregel (OBJECT').

Das kann nur aus Prämissen der Form $(\Gamma^\star)^\star[^\tau/_{self}] \triangleright r :: \phi$ und $\langle\phi\rangle \sim \tau$ folgen. Der small step $\mathbf{object}\,(self : \tau)\ r\ \mathbf{end} \rightarrow e'$ kann andererseits nur mit der small step Regel (OBJECT-EVAL) hergeleitet worden sein, wobei nach Voraussetzung ein small step $r \rightarrow r'$ existiert und $e' = \mathbf{object}\,(self : \tau)\ r'\ \mathbf{end}$. Nach Induktionsvoraussetzung gilt also

$$(\Gamma^\star)^\star[^\tau/_{self}] \triangleright r' :: \phi,$$

und mit Typregel (OBJECT') folgt dann unmittelbar die Behauptung.

5.) $\Gamma^\star \triangleright e_1\#m :: \tau$ mit Typregel (SEND).

Dann gilt nach Voraussetzung $\Gamma^\star \triangleright e_1 :: \langle m : \tau; \phi\rangle$. Der small step $e_1\#m \rightarrow e'$ kann nur mit (SEND-EVAL) oder (SEND-UNFOLD) hergeleitet worden sein:

1.) Für $e_1\#m \rightarrow e'$ mit (SEND-EVAL) existiert dann nach Voraussetzung ein small step $e_1 \rightarrow e_1'$ und es gilt $e' = e_1'\#m$. Das bedeutet, die Behauptung folgt direkt mit Induktionsvoraussetzung und Typregel (SEND).

2.) Falls $e_1\#m \rightarrow e'$ mit small step Regel (SEND-UNFOLD) hergeleitet worden ist, so muss gelten $e_1 = \mathbf{object}\,(self : \tau')\ \omega\ \mathbf{end}$ und $e' = \omega[^{e_1}/_{self}]\#m$. Nach dem vorangegangenen Lemma (4.17) gilt

$$(\Gamma^\star)^\star[^{\tau'}/_{self}] \triangleright \omega :: \phi'$$

mit $\langle\phi'\rangle \sim \tau'$ und $\tau' \lesssim \langle m : \tau; \phi\rangle$. Das wiederum bedeutet, dass mit Typregel (SUBSUME')

$$\Gamma^\star \triangleright \mathbf{object}\,(self : \tau')\ \omega\ \mathbf{end} :: \tau'$$

folgt. Damit sind alle Voraussetzungen für die Anwendung des speziellen Substitutions-Lemmas (4.16) erfüllt, und wir erhalten

$$\Gamma^\star \triangleright \omega\big[^{\mathbf{object}\,(self:\tau')\ \omega\ \mathbf{end}}/_{self}\big] :: \phi'.$$

Da $\langle\phi'\rangle \sim \tau'$ und $\tau' \lesssim \langle m : \tau; \phi\rangle$ existieren also nach Subtyping-Lemma (4.12) $\tau'' \in \mathit{Type}$ und $\phi'' \in \mathit{RType}$ mit $\tau'' \lesssim \tau$ und $\langle\phi''\rangle \lesssim \langle\phi'\rangle$, so dass $\phi' = (m : \tau''; \phi'')$. Damit erhalten wir schliesslich mit Typregel (Send')

$$\Gamma^\star \triangleright \omega\big[^{\mathbf{object}\,(self:\tau')\ \omega\ \mathbf{end}}/_{self}\big]\#m :: \tau'',$$

denn $(\Gamma^\star)^\star = \Gamma^\star$, und wegen $\tau'' \lesssim \tau$ folgt schliesslich mit Typregel (Subsume') die Behauptung.

Die restlichen Fälle verlaufen ähnlich. □

Wir haben also gezeigt, dass auch für das Typsystem der Programmiersprache $\mathcal{L}_o^{srt}$, small steps typerhaltend sind. Es bleibt wie zuvor noch das Progress-Theorem zu zeigen. Dazu formulieren wir wieder wie gehabt ein *Canonical Forms Lemma* für die Sprache $\mathcal{L}_o^{srt}$, wobei wir uns wieder auf die im nachfolgenden Beweis verwendeten Aussagen beschränken, statt das Lemma in seiner allgemeinsten Form anzugeben.

Lemma 4.18 (Canonical Forms) *Sei $v \in \mathit{Val}$, $\tau \in \mathit{Type}$, und gelte $[\,] \triangleright v :: \tau$.*

(a) Wenn $\tau \lesssim \mathbf{int}$, dann gilt $v \in \mathit{Int}$.

(b) Wenn $\tau \lesssim \tau_1 \to \tau_2$, dann gilt eine der folgenden Aussagen:

1.) $v \in \mathit{Op}$

2.) $v = op\, v_1$ mit $op \in \mathit{Op}$ und $v_1 \in \mathit{Val}$

3.) $v = \lambda x : \tau_1'.e$ mit $x \in \mathit{Var}$, $\tau_1' \in \mathit{Type}$ und $e \in \mathit{Exp}$

(c) Wenn $\tau \lesssim \langle\phi\rangle$, dann gilt $v = \mathbf{object}\,(self : \tau')\ \omega\ \mathbf{end}$.

Beweis: Wir überspringen den Beweis an dieser Stelle, da es sich bis auf Kleinigkeiten um exakt den gleichen Beweis handelt, wie im Canonical Forms Lemma für die Programmiersprache $\mathcal{L}_o^{rt}$ (4.18). □

Somit können wir schliesslich das Progress-Theorem beweisen und formulieren. Wie nicht anders zu erwarten, stimmt die Formulierung des Theorems mit den vorangegangenen Formulierungen überein (vgl. Satz 3.2 und Satz 4.2), und auch der Beweis verläuft sehr ähnlich.

Satz 4.5 (Existenz des Übergangsschritts, „Progress“)

(a) $\forall e \in \mathit{Exp}, \tau \in \mathit{Type} : [\,] \triangleright e :: \tau \Rightarrow (e \in \mathit{Val} \vee \exists e' \in \mathit{Exp} : e \to e')$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma^+ \triangleright r :: \phi \Rightarrow (r \in \mathit{RVal} \vee \exists r' \in \mathit{Row} : r \to r')$

Beweis: Wir führen den Beweis wieder durch simultane Induktion über die Länge der Herleitung der Typurteile $[\,] \triangleright e :: \tau$ und $\Gamma^+ \triangleright r :: \phi$, mit Fallunterscheidung nach der jeweils zuletzt angewandten Typregel. Dazu betrachten wir exemplarisch die folgenden Fälle.

1.) Für (Const) und (Abstr) gilt $e \in \mathit{Val}$, während für (Empty) gilt $r \in \mathit{RVal}$.

2.) $[\,] \triangleright e :: \tau$ mit Typregel (Subsume').

Nach Voraussetzung existiert ein $\tau' \in \mathit{Type}$ mit $[\,] \triangleright e :: \tau'$ und $\tau' \lesssim \tau$. Also folgt die Behauptung unmittelbar mir Induktionsvoraussetzung.

3.) $[\,] \triangleright e_1\, e_2 :: \tau$ mit Typregel (App).

Das kann nur aus Prämissen der Form $[\,] \triangleright e_1 :: \tau_2 \to \tau$ und $[\,] \triangleright e_2 :: \tau_2$ folgen. Dann zeigen wir je nach Form der Teilausdrücke e_1 und e_2, dass entweder die Applikation bereits ein Wert ist, oder ein small step existiert.

Für $e_1 \notin \mathit{Val}$ lässt sich wie früher leicht zeigen, dass ein small step mit (App-Left) existiert. Ebenso einfach zeigt man, dass für $e_1 \in \mathit{Val}$ und $e_2 \notin \mathit{Val}$ ein small step mit (App-Right) existiert.

Es bleibt also der Fall $e_1, e_2 \in \mathit{Val}$ zu betrachten. Da $\lesssim$ reflexiv ist, lässt sich das Canonical Forms Lemma (4.18) anwenden, demzufolge drei Fälle zu unterscheiden sind:

1.) $e_1 \in \mathit{Op}$
Trivial.

2.) $e_1 = \mathit{op}\, v_1$ mit $\mathit{op} \in \mathit{Op}$ und $v_1 \in \mathit{Val}$
Dann existiert nach Lemma 4.17 ein $\tau_1 \in \mathit{Type}$ mit $[\,] \triangleright \mathit{op} :: \tau_1 \to \tau_2 \to \tau$ und $[\,] \triangleright v_1 :: \tau_1$. Wie früher folgt daraus leicht, dass $v_1, e_2 \in \mathit{Int}$, und somit ein small step $\mathit{op}\, v_1\, e_2 \to \mathit{op}^I(v_1, e_2)$ mit small step Regel (Op) existiert.

3.) $e_1 = \lambda x : \tau_2'.e_1'$ mit $x \in \mathit{Var}$, $\tau_2' \in \mathit{Type}$ und $e_1' \in \mathit{Exp}$
Dann ist trivialerweise ersichtlich, dass ein small step mit (Beta-V) möglich ist.

4.) $[\,] \triangleright e_1 \# m :: \tau$ mit Typregel (SEND).

Nach Voraussetzung gilt $[\,] \triangleright e_1 :: \langle m : \tau; \phi \rangle$. Mit Induktionsvoraussetzung folgt, dass entweder $e_1 \in \mathit{Val}$ gilt oder ein small step $e_1 \to e_1'$ existiert.

Sei also $e_1 \in \mathit{Val}$, dann gilt nach Lemma 4.18, dass $e_1 = \mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end}$. Also existiert ein small steo

$$(\mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end}) \# m \to \omega[{}^{e_1}/_{\mathit{self}}] \# m$$

mit small step Regel (SEND-UNFOLD).

Ist andererseits $e_1 \notin \mathit{Val}$, so existiert ein small step

$$e_1 \# m \to e_1' \# m$$

mit small step Regel (SEND-EVAL).

Die restlichen Fälle verlaufen analog. □

Wie zuvor folgt dann mit Hilfe der Preservation- und Progress-Theoreme die Typsicherheit der Programmiersprache $\mathcal{L}_o^{srt}$. Der Vollständigkeithalber wiederholen wir den Satz aber gerne ein weiteres Mal.

Satz 4.6 (Typsicherheit, „Safety“) *Wenn $[\,] \triangleright e :: \tau$, dann bleibt die Berechnung für e nicht stecken.*

Beweis: Folgt analog wie für $\mathcal{L}_o^{rt}$ aus den Preservation und Progress-Sätzen. □

4.2.2 Typalgorithmus

Es ist offensichtlich, dass sich für die Programmiersprache $\mathcal{L}_o^{srt}$, ähnlich wie dies zuvor für die Sprache $\mathcal{L}_o^{sub}$ vorgeführt wurde, ein Minimal Typing Kalkül angeben lässt. Aus Abschnitt 4.1 wissen wir darüber hinaus, dass die Relation $\sim$ entscheidbar. Es bleibt uns also zu zeigen, dass auch die Relation $\lesssim$ entscheidbar ist.

TODO

4.3 Beispiele

4.3.1 Punktobjekte

TODO: Vollständiges Beispiel.

4.3.2 Rekursive Datenstrukturen

TODO: Listenbeispiel aus den Unterlagen.

5 Vererbung

TODO: Einleitung

5.1 Syntax der Sprache $\mathcal{L}_c$

TODO: Prosa

Definition 5.1

(a) Vorgegeben sei eine unendliche Menge *Super* von *Basisklassennamen* z.

(b) Die Menge *Id* aller *Namen* von $\mathcal{L}_c$ wird wie folgt erweitert:

$$Id = Var \cup Attribute \cup Self \cup \{z\#m \mid z \in Super, m \in Method\}$$

Die einzelnen Bestandteile der Menge *Id* werden wie gehabt als disjunkt angenommen.

Definition 5.2 (Abstrakte Syntax von $\mathcal{L}_c$)

(a) Die Menge *Body* aller *Klassenrümpfe* b von $\mathcal{L}_c$ ist durch die folgende kontextfreie Grammatik definiert:

$$\begin{array}{rcl} b & ::= & \textbf{inherit}\ a_1, \ldots, a_k; m_1, \ldots, m_l\ \textbf{from}\ e\ \textbf{as}\ z;\ b_1 \\ & \mid & r \end{array}$$

Für $a_1, \ldots, a_k$ schreiben wir kurz A, und für $m_1, \ldots, m_l$ schreiben wir kurz M.

(b) Die Menge *Exp* aller *Ausdrücke* e von $\mathcal{L}_c$ erhält man durch folgende Erweiterung der kontextfreien Grammatik von $\mathcal{L}_o$.

$$\begin{array}{rcl} e & ::= & \textbf{class}\,(\mathit{self})\ b\ \textbf{end} \\ & \mid & \textbf{new}\, e_1 \end{array}$$

5.2 Operationelle Semantik der Sprache $\mathcal{L}_c$

TODO: Prosa.

Definition 5.3 (Werte und Reihenwerte) Die Menge $\mathit{Val} \subseteq \mathit{Exp}$ aller *Werte* v von $\mathcal{L}_c$ erhält man, indem man die Produktion

$$v \quad ::= \quad \textbf{class}\,(\mathit{self})\ r\ \textbf{end}$$

zur kontextfreien Grammatik von $\mathcal{L}_o$ (vgl. Definition 2.4) hinzunimmt.

5.2.1 Frei vorkommende Namen und Substitution

Definition 5.4 (Frei vorkommende Namen)

(a) Die Menge $\mathit{free}\,(e)$ aller im Ausdruck $e \in \mathit{Exp}$ *frei vorkommenden Namen* erhält man durch folgende Verallgemeinerung von Definition 2.5.

$$\begin{aligned} \mathit{free}\,(\textbf{class}\,(\mathit{self})\ b\ \textbf{end}) &= \mathit{free}\,(b) \setminus \{\mathit{self}\} \\ \mathit{free}\,(\textbf{new}\,e) &= \mathit{free}\,(e) \end{aligned}$$

(b) Die Menge $\mathit{free}\,(b)$ aller im Klassenrumpf $b \in \mathit{Body}$ *frei vorkommenden Namen* ist wie folgt definiert:

$$\begin{aligned} \mathit{free}\,(\textbf{inherit}\ A; M\ \textbf{from}\ e\ \textbf{as}\ z;\ b) &= \{\mathit{self}\} \cup \mathit{free}\,(e) \\ &\cup\ \mathit{free}\,(b) \setminus (A \cup \{z\#m \mid m \in M\}) \end{aligned}$$

TODO: Konvention: alle Attribute eindeutig in einer Reihe, das beinhaltet auch **inherit**'s, also kein mehrfaches Erben der gleichen Klasse möglich (lässt sich umgehen, z.B. durch geeignete Umbenennung wie Präfixing mit Supername z, würde aber die Betrachtung deutlich aufwendiger machen).

Die Menge $\mathit{Exp}^\star$ ist nach wie vor so definiert, dass nur Ausdrücke aus dieser Menge nur freie Vorkommen von Variablennamen enthalten dürfen, jedoch keine freien Vorkommen von Attribute- und Objektnamen, und neuerdings auch keine freien Vorkommen von Basismethodennamen $z\#m$ (vgl. Definition 2.6).

Die Reiheneinsetzung ist wie früher definiert (Definition 2.7). Entsprechend müssen wir nur die Definition der Substitution für die Programmiersprache $\mathcal{L}_c$ erweitern.

Definition 5.5 (Substitution)

(a) Für $e' \in \mathit{Exp}$, $e \in \mathit{Exp}^\star$ und $\mathit{id} \in \mathit{Id}$ erhält man den Ausdruck $e'[{}^{e}/_{\mathit{id}}]$, der aus e' durch *Substitution* von e für id entsteht, durch folgende Erweiterung von Definition 2.8:

$$\begin{aligned} \mathbf{class}\,(\mathit{self} : \tau)\ b\ \mathbf{end}[{}^{e}/_{\mathit{id}}] &= \begin{cases} \mathbf{class}\,(\mathit{self})\ b\ \mathbf{end} & \text{falls } \mathit{id} = \mathit{self} \\ \mathbf{class}\,(\mathit{self})\ b[{}^{e}/_{\mathit{id}}]\ \mathbf{end} & \text{sonst} \end{cases} \\ (\mathbf{new}\, e_1)[{}^{e}/_{\mathit{id}}] &= \mathbf{new}\, e_1[{}^{e}/_{\mathit{id}}] \end{aligned}$$

(b) Für $b \in \mathit{Body}$, $e \in \mathit{Exp}^\star$ und $\mathit{id} \in \mathit{Id}$ ist die Reihen $b[{}^{e}/_{\mathit{id}}]$, die aus r durch *Substitution* von e für id entsteht, wie folgt definiert:

$$\begin{aligned} &(\mathbf{inherit}\ A; M\ \mathbf{from}\ e_1\ \mathbf{as}\ z;\ b_1)[{}^{e}/_{\mathit{id}}] \\ &= \begin{cases} \mathbf{inherit}\ A; M\ \mathbf{from}\ e_1[{}^{e}/_{\mathit{id}}]\ \mathbf{as}\ z;\ b_1 & \text{falls } \mathit{id} \in A \cup \{z\#m \mid m \in M\} \\ \mathbf{inherit}\ A; M\ \mathbf{from}\ e_1[{}^{e}/_{\mathit{id}}]\ \mathbf{as}\ z;\ b_1[{}^{e}/_{\mathit{id}}] & \text{sonst} \end{cases} \end{aligned}$$

5.2.2 Small step Semantik

TODO: Einleitung

Definition 5.6 (Gültige small steps für $\mathcal{L}_c$) Ein small step, $e \to_e e'$ mit $e, e' \in \mathit{Exp}$, $b \to_b b'$ mit $b, b' \in \mathit{Body}$ oder $r \to_r r'$ mit $r, r' \in \mathit{Row}$, heißt *gültig* für $\mathcal{L}_c$, wenn er sich mit den small step Regeln von $\mathcal{L}_o$ (vgl. Definition 2.10), sowie den folgenden zusätzlichen small step Regeln für Ausdrücke

$$\text{(Class-Eval)}\quad \frac{b \to_b b'}{\mathbf{class}\,(\mathit{self})\ b\ \mathbf{end} \to_e \mathbf{class}\,(\mathit{self})\ b'\ \mathbf{end}}$$

$$\text{(New-Eval)}\quad \frac{e \to_e e'}{\mathbf{new}\, e \to_e \mathbf{new}\, e'}$$

$$\text{(New-Exec)}\quad \mathbf{new}\,(\mathbf{class}\,(\mathit{self})\ r\ \mathbf{end}) \to_e \mathbf{object}\,(\mathit{self})\ r\ \mathbf{end}$$

und den neuen small step Regeln für Klassenrümpfe

$$\text{(Inherit-Right)}\quad \frac{b \to_b b'}{\mathbf{inherit}\ A; M\ \mathbf{from}\ e\ \mathbf{as}\ z;\ b \to_b \mathbf{inherit}\ A; M\ \mathbf{from}\ e\ \mathbf{as}\ z;\ b'}$$

$$\text{(Inherit-Left)}\quad \frac{e \to_e e'}{\mathbf{inherit}\ A; M\ \mathbf{from}\ e\ \mathbf{as}\ z;\ r \to_b \mathbf{inherit}\ A; M\ \mathbf{from}\ e'\ \mathbf{as}\ z;\ r}$$

$$\text{(Inherit-Exec)}\quad \mathbf{inherit}\ A; M\ \mathbf{from}\ (\mathbf{class}\,(\mathit{self})\ r_1\ \mathbf{end})\ \mathbf{as}\ z;\ r_2 \to_b r_1 \oplus r_2[{}^{m(r_1)}/_{z\#m}]_{m \in M}$$

herleiten lässt.

A Big step Interpreter

TODO: Ocaml Version des big steppers.

Index

T

V

W

Literaturverzeichnis

[Kin05] Kindler, Ekkart: *Semantik*. `http://wwwcs.uni-paderborn.de/cs/kindler/Lehre/WS04/Semantik/PDF/Skriptentwurf.pdf`. Version: 2005, Abruf: 1. Juli 2007. – Skript zur Vorlesung „Semantik von Programmiersprachen“

[Pie02] Pierce, Benjamin C.: *Types and Programming Languages*. MIT Press, 2002. – ISBN 0–262–16209–1

[Rém02] Rémy, Didier: Using, Understanding, and Unraveling the OCaml Language. In: Barthe, Gilles (Hrsg.): *Applied Semantics. Advanced Lectures. LNCS 2395.* Springer Verlag, 2002. – ISBN 3–540–44044–5, S. 413–537

[RV97] Rémy, Didier ; Vouillon, Jérôme: Objective ML: A simple object-oriented extension of ML. In: *Proceedings of the 24th ACM Conference on Principles of Programming Languages*. Paris, France, January 1997, S. 40–53

[RV98] Rémy, Didier ; Vouillon, Jérôme: Objective ML: An effective object-oriented extension to ML. In: *Theory And Practice of Object Systems* 4 (1998), Nr. 1, S. 27–50. – A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997

[Sie04] Sieber, Kurt: *Theorie der Programmierung I+II*. 2004. – Vorlesungsmitschrift

[Sie06] Sieber, Kurt: *Theorie der Programmierung I*. 2006. – Vorlesungsmitschrift

[Sie07] Sieber, Kurt: *Theorie der Programmierung III*. 2007. – Vorlesungsmitschrift

[Spr92] Spreen, Dieter: *Berechenbarkeit und Komplexitätstheorie*. 1992. – Vorlesungsskript

[Str00] Stroustrup, Bjarne: *Die C++ Programmiersprache*. 4. Auflage. München : Addison-Wesley Verlag, 2000. – ISBN 3–8273–1660–X

[Tar55] Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. In: *Pacific J. Math.* 5 (1955), S. 285–309

[Wag94] Wagner, Klaus W.: *Einführung in die Theoretische Informatik*. Springer Verlag, 1994. – ISBN 3–540–58139–1

[Weg93] Wegener, Ingo: *Theoretische Informatik*. Teubner Verlag, 1993. – ISBN 3–519–02123–4

Erklärung

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

Siegen, den 10. Juli 2007 _____

(Benedikt Meurer)