

Diplomarbeit

Theoretische Grundlagen der Objektorientierung

Benedikt Meurer

21. Juli 2007

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

Inhaltsverzeichnis

Einleitung	5
1 Grundlagen	7
1.1 Mathematische Grundlagen	7
1.2 Die funktionale Programmiersprache \mathcal{L}_f	11
2 Funktionale Objekte	21
2.1 Syntax der Sprache \mathcal{L}_o	21
2.2 Operationelle Semantik der Sprache \mathcal{L}_o	23
2.3 Ein einfaches Typsystem	38
3 Subtyping	59
3.1 Motivation	59
3.2 Die Subtyprelation	59
3.3 Die Sprache \mathcal{L}_o^{sub}	66
3.4 Minimal Typing	78
3.5 Coercions	94
4 Rekursive Typen	97
4.1 Die Sprache \mathcal{L}_o^{rt}	98
4.2 Die Sprache \mathcal{L}_o^{srt}	116
4.3 Beispiele	132
5 Vererbung	133
5.1 Syntax der Sprache \mathcal{L}_c	133
5.2 Operationelle Semantik der Sprache \mathcal{L}_c	133
5.3 Typsystem	136
A Big step Interpreter	159
Index	161

Literaturverzeichnis	163
Erklärung	165

Einleitung

Narf, [Rém02, 50f].

1 Grundlagen

1.1 Mathematische Grundlagen

Im folgenden geben wir eine kurze Einführung in die in diesem Dokument verwendete Mathematik, welche im wesentlichen auf den Ausführungen in [DP88] basiert. Für den Leser soll dieser Abschnitt lediglich als Rückblick oder Gedächtnisstütze dienen.

TODO: Potenzmenge definieren.

Definition 1.1 (Eigenschaften von Relationen) Sei M eine beliebige Menge und $R \subseteq M \times M$ eine binäre Relation in M .

- (a) R heisst *reflexiv*, wenn $(x, x) \in R$ für alle $x \in M$ gilt.
- (b) R heisst *transitiv*, wenn für alle $x, y, z \in M$ aus $(x, y) \in R$ und $(y, z) \in R$ folgt, dass auch $(x, z) \in R$ gilt.
- (c) R heisst *symmetrisch*, wenn für alle $x, y \in M$ aus $(x, y) \in R$ folgt, dass auch $(y, x) \in R$ gilt.
- (d) R heisst *antisymmetrisch*, wenn für alle $x, y \in M$ aus $(x, y) \in R$ und $(y, x) \in R$ folgt, dass $y = x$ gilt.

Definition 1.2 (Ordnungsrelationen) Sei D eine beliebige Menge und $\sqsubseteq \subseteq D \times D$ eine binäre Relation in D .

- (a) (D, \sqsubseteq) heisst *Quasiordnung*, wenn \sqsubseteq reflexiv und transitiv ist.
- (b) Eine Quasiordnung (D, \sqsubseteq) heisst *partielle Ordnung* (engl.: *partial order*, kurz *po*), wenn \sqsubseteq zusätzlich antisymmetrisch ist.
- (c) Eine partielle Ordnung (D, \sqsubseteq) heisst *total*, wenn $x \sqsubseteq y$ oder $y \sqsubseteq x$ für alle $x, y \in D$ gilt.
- (d) Eine Quasiordnung (D, \sqsubseteq) heisst *Äquivalenzrelation*, wenn \sqsubseteq zusätzlich symmetrisch ist.

Wenn aus dem Kontext ersichtlich ist, auf welche Menge Bezug genommen wird, lässt man auch die Menge weg und schreibt nur die Relation, also \sqsubseteq statt (D, \sqsubseteq) . Entsprechend schreibt man auch lediglich D , wenn klar ist, wie die Menge geordnet wird.

Definition 1.3 TODO: Schreibweise für Abbildungen in diesem Dokument. Seien A, B Mengen, $a \in A, b \in B$ und $f : A \rightarrow B$ eine partielle Funktion. Dann bezeichnet $f[b/a]$ die partielle Funktion $g : A \rightarrow B$ mit den folgenden Eigenschaften.

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

1.1.1 Bereichstheorie

Dieser Abschnitt bietet eine kurze Einführung in die wichtigsten Begriffe der Bereichstheorie, 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 1.4 Sei (D, \sqsubseteq) eine partielle Ordnung und $S \subseteq D$.

- (a) $d \in S$ heisst *kleinstes Element* von S , wenn $d \sqsubseteq s$ für alle $s \in S$.
- (b) $d \in D$ heisst *obere Schranke* von S , wenn $s \sqsubseteq d$ für alle $s \in S$.
- (c) $d \in S$ heisst *kleinste obere Schranke* oder *Supremum* von S , wenn d kleinstes Element der Menge aller oberen Schranken von S ist.

Analog zur oberen Schranke definiert man den Begriff der *unteren Schranke* und das *Infimum* als grösstes Element der Menge aller unteren Schranken. Wenn D selbst ein kleinstes Element besitzt, so bezeichnet man dieses Element mit \perp oder \perp_D . Entsprechend bezeichnet man ein grösstes Element mit \top oder \top_D . Das Supremum einer Menge $S \subseteq D$ bezeichnet man mit $\bigsqcup S$ oder $\bigsqcup_D S$, das Infimum mit $\bigsqcap S$ oder $\bigsqcap_D S$.

Definition 1.5 (Gerichtete Ordnungen) Sei (D, \sqsubseteq) eine partielle Ordnung.

- (a) Eine Menge $\Delta \subseteq D$ heisst *gerichtet* (engl.: *directed*), wenn $\Delta \neq \emptyset$ und zu je zwei Elementen $d_1, d_2 \in \Delta$ existiert ein $d \in \Delta$ mit $d_1 \sqsubseteq d$ und $d_2 \sqsubseteq d$.
- (b) (D, \sqsubseteq) heisst *gerichtet vollständig* (engl.: *directed complete partial order*, kurz *dcpo*), wenn jede gerichtete Teilmenge von D ein Supremum besitzt.

Lemma 1.1 *Sei M eine beliebige Menge. Dann gilt:*

- (a) $(\mathcal{P}(M), \subseteq)$ ist eine dcpo mit kleinstem Element \emptyset .
- (b) $(\mathcal{P}(M), \supseteq)$ ist eine dcpo mit kleinstem Element M .

Beweis:

- (a) Es ist ziemlich offensichtlich, dass \emptyset bezüglich \subseteq das kleinste Element jeder Potenzmenge ist, ebenso wie es trivialerweise ersichtlich ist, dass jede Potenzmenge durch \subseteq partiell geordnet wird. Damit bleibt zu zeigen, dass jede gerichtete Teilmenge von $\mathcal{P}(M)$ ein Supremum besitzt.

Sei also $\Delta \subseteq \mathcal{P}(M)$ gerichtet. Dann ist $\bigcup \Delta$ eine obere Schranke von Δ , denn es gilt $\delta \subseteq \bigcup \Delta$ für alle $\delta \in \Delta$. Es ist ziemlich offensichtlich, dass keine kleinere obere Schranke von Δ existiert, also ist $\bigcup \Delta$ das Supremum von Δ .

- (b) Folgt analog zur ersten Aussage des Lemmas. □

Definition 1.6 (Monotone und stetige Abbildungen) Seien (D, \sqsubseteq_D) und (E, \sqsubseteq_E) dcpos.

- (a) Eine (totale) Abbildung $f : D \rightarrow E$ heisst *monoton*, wenn für alle $d_1, d_2 \in D$ gilt:

$$d_1 \sqsubseteq_D d_2 \Rightarrow f(d_1) \sqsubseteq_E f(d_2)$$

- (b) Eine monotone Abbildung $f : D \rightarrow E$ heisst *stetig*, wenn für jede gerichtete Teilmenge $\Delta \subseteq D$ gilt:

$$f(\bigsqcup_D \Delta) = \bigsqcup_E f(\Delta)$$

Korollar 1.1 *Seien D, E dcpos und sei $f : D \rightarrow E$ eine monotone Abbildung. f ist genau dann stetig, wenn für jede gerichtete Teilmenge $\Delta \subseteq D$ gilt:*

$$f(\bigsqcup_D \Delta) \sqsubseteq_E \bigsqcup_E f(\Delta)$$

Beweis: Klar, da die Gültigkeit von \sqsubseteq_E bereits aus der Monotonie von f folgt, denn es gilt: $\bigsqcup_D \Delta$ ist eine obere Schranke von Δ , also ist $f(\bigsqcup_D \Delta)$ eine obere Schranke von $f(\Delta)$, und damit $f(\bigsqcup_D \Delta) \sqsupseteq_E \bigsqcup_E f(\Delta)$. □

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, erhält dafür aber ein konstruktives Verfahren zur Bildung des kleinsten Fixpunktes.

Satz 1.1 (Fixpunktsatz von Kleene) *Sei D eine dcpo, die ein kleinstes Element \perp besitzt, und $f : D \rightarrow D$ eine stetige Abbildung. Dann gilt:*

- (a) *Die Menge $\{f^n(\perp) \mid n \in \mathbb{N}\}$ ist gerichtet.*
- (b) *$\mu f = \bigsqcup \{f^n(\perp) \mid n \in \mathbb{N}\}$ ist kleinster Fixpunkt von f .*

Statt $\bigsqcup \{f^n(\perp) \mid n \in \mathbb{N}\}$ verwendet man häufig die Kurzschreibweise $\bigsqcup_{n \in \mathbb{N}} f^n(\perp)$.

Beweis:

- (a) Es gilt $\perp \sqsubseteq f(\perp)$, da \perp nach Voraussetzung kleinstes Element von D ist. Wegen der Monotonie von f folgt daraus $f(\perp) \sqsubseteq f^2(\perp)$, was sich durch vollständige Induktion auf $f^n(\perp) \sqsubseteq f^{n+1}(\perp)$ für alle $n \in \mathbb{N}$ erweitern lässt. Wegen der Transitivität von \sqsubseteq folgt damit, dass $\{f^n(\perp) \mid n \in \mathbb{N}\}$ gerichtet ist.
- (b) Wir zeigen zunächst, dass μf ein Fixpunkt von f ist, also:

$$\begin{aligned} f(\mu f) &= f(\bigsqcup_{n \in \mathbb{N}} f^n(\perp)) \\ &= \bigsqcup_{n \in \mathbb{N}} f^{n+1}(\perp) \\ &= \bigsqcup_{n \in \mathbb{N}} f^n(\perp) \\ &= \mu f \end{aligned}$$

Es bleibt noch zu zeigen, dass μf der kleinste Fixpunkt ist. Sei dazu $d \in D$ ein beliebiger Fixpunkt von f , also $f(d) = d$. Da $\mu f = \bigsqcup_{n \in \mathbb{N}} f^n(\perp)$ die kleinste obere Schranke aller $f^n(\perp)$ ist, genügt es zu zeigen, dass d eine obere Schranke von $\{f^n(\perp) \mid n \in \mathbb{N}\}$ ist. Dazu zeigen wir durch vollständige Induktion über n , dass $f^n(\perp) \sqsubseteq d$ für alle $n \in \mathbb{N}$:

- $n = 0$
 $f^0(\perp) = \perp \sqsubseteq d$

- $n \rightsquigarrow n + 1$

Nach Induktionsvoraussetzung gilt $f^n(\perp) \sqsubseteq d$. Daraus folgt

$$f^{n+1}(\perp) = f(f^n(\perp)) \sqsubseteq f(d) = d,$$

da f monoton ist und d nach Annahme ein Fixpunkt von f ist. □

TODO: Vielleicht noch Prosa?

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 *structural 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’Caml¹, 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 Light², 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 1.7 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 $n \in \text{Int}$ 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 1.8 (Abstrakte Syntax von \mathcal{L}_f) Die Menge Op aller Operatoren op ist definiert durch die kontextfreie Grammatik

$$\begin{array}{ll}
op ::= + & | \quad - & | \quad * & \text{arithmetische Operatoren} \\
& | \quad < & | \quad > & | \quad \leq & | \quad \geq & | \quad = & \text{Vergleichsoperatoren,}
\end{array}$$

die Menge $Const$ aller Konstanten c durch

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

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

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

Dem aufmerksamen Leser wird sicherlich nicht entgangen sein, dass keine Operatoren für Ganzzahldivision und Divisionsrest existieren. Diese wurden bewusst weggelassen, da eine Hinzunahme dieser Operatoren zur Kernsprache eine Laufzeitfehlerbehandlung notwendig macht, welche unsere Betrachtungen der Sprache unnötig kompliziert machen würden. Wir werden Laufzeitfehlerbehandlung später als Erweiterung der Kernsprache betrachten (**TODO:** Nur, wenn das Exception-Kapitel rein kommt).

Nichtsdestotrotz 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 1.9 (Frei vorkommende Variablen) Die Menge $free(e)$ aller im Ausdruck $e \in Exp$ frei vorkommenden Variablen ist wie folgt induktiv definiert.

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

TODO: Blub blah

Definition 1.10 Ein Ausdruck $e \in Exp$ heisst *abgeschlossen*, wenn $free(e) = \emptyset$.

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 1.11 (Substitution) Für $e, e' \in Exp$ und $x \in Var$ 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.

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

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 $x \in \text{free}(x)$. 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 $x' \notin \{y\} \cup \text{free}(x)$. 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.

$$(\lambda x'. e')[e/x] = \lambda x''. e'[x''/x'] [e/x] \\ \text{mit } x'' \notin \{x\} \cup \text{free}(e) \cup \text{free}(\lambda x'. e')$$

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 TPML³ 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 $\{x\} \cup \text{free}(e) \cup \text{free}(\lambda x'.e')$ 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

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

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

$$\begin{aligned} \text{div} &= \text{rec } \text{div}.\lambda x.\lambda y. \\ &\quad \text{if } x < y \text{ then } 0 \\ &\quad \text{else } (\text{div } (x - y) \ y) + 1 \end{aligned}$$

$$\text{mod} = \lambda x.\lambda y.x - y * (\text{div } x \ y)$$

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 Definitionen. Zunächst müssen wir festlegen, wie die Operatoren der Sprache zu interpretieren sind.

Definition 1.12 Für jeden Operator $op \in Op$ sei eine Funktion op^I vorgegeben mit

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

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

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

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

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

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

Definition 1.13 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 1.14 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 1.15 (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)	$\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightarrow_e e_1$
(COND-FALSE)	$\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rightarrow_e e_2$

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 ist⁴.
- 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 auszuwerten⁵.

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

⁵Diese 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 Teilausdruck 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’Caml¹, 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.

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

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

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

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.

$$\begin{array}{ll} e ::= & id \quad \text{Variable, Attribut- oder Objektname} \\ & | e_1 \# m \quad \text{Methodenaufruf} \\ & | \text{object } (self) \ r \ \text{end} \quad \text{Objekt} \\ & | \{ \langle a_1 = e_1; \dots; a_n = e_n \rangle \} \quad \text{Duplikation} \end{array}$$

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 $r \in \text{Row}$ eine Reihe.

(a) Die Menge der Attributnamen $\text{dom}_a(r)$ der Reihe r ist wie folgt induktiv definiert.

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

(b) Die Menge der Methodennamen $\text{dom}_m(r)$ der Reihe r ist wie folgt induktiv definiert.

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

(c) Für $a \in \text{dom}_a(r)$ ist der Ausdruck $r(a)$ wie folgt induktiv definiert.

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

(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

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

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.

$$\begin{aligned} \omega &::= \epsilon \\ &\quad | \ \mathbf{val} \ a = v; \omega \\ &\quad | \ \mathbf{method} \ m = e; \omega \end{aligned}$$

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

$$\begin{array}{ll} v ::= id & \text{Name} \\ | \text{object}(self) \ \omega \ \text{end} & \text{Objektwert} \end{array}$$

zur kontextfreien Grammatik von \mathcal{L}_f (vgl. Definition 1.13) 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 $free(e)$ aller im Ausdruck $e \in Exp$ frei vorkommenden Namen erhält man durch folgende Verallgemeinerung von Definition 1.9.

$$\begin{aligned} free(c) &= \emptyset \\ free(id) &= \{id\} \\ free(e \# m) &= free(e) \\ free(\omega \# m) &= free(\omega) \\ free(\text{object}(self) \ r \ \text{end}) &= free(r) \setminus \{self\} \\ free(\{\langle a_i = e_i^{i=1 \dots n} \rangle\}) &= \{self\} \cup \bigcup_{i=1}^n (free(e_i) \cup \{a_i\}) \end{aligned}$$

- (b) Die Menge $free(r)$ aller in der Reihe $r \in Row$ frei vorkommenden Namen wird wie folgt induktiv definiert.

$$\begin{aligned} free(\epsilon) &= \emptyset \\ free(\text{val } a = e; r) &= free(e) \cup (free(r) \setminus \{a\}) \\ free(\text{method } m = e; r) &= free(e) \cup free(r) \setminus \{self\} \end{aligned}$$

TODO: Prosa, zum Beispiel, dass Methodennamen nicht in $free(e)$ 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 $Exp^* \subseteq Exp$ aller Ausdrücken, die keine freien Vorkommen von Attribut- und Objektamen enthalten, ist wie folgt definiert.

$$Exp^* = \{e \in Exp \mid free(e) \subseteq Var\}$$

TODO: Überleitung

Definition 2.7 (Reiheneinsetzung) Für $r \in Row$, $e \in Exp^*$ 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 $e' \in Exp$, $e \in Exp^*$ 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 1.11.

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

- (b) Für $r \in Row$, $e \in Exp^*$ 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 Objektamen enthält. Insbesondere gilt aber zu beachten, dass die Substitution für Ausdrücke $e \in \text{Exp} \setminus \text{Exp}^*$ 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 $e' \in \text{Exp}$, $e \in \text{Exp}^*$ und $id \in \text{Id}$ ist $e'[e/id]$ eindeutig bestimmt.
- (b) Für $r \in \text{Row}$, $e \in \text{Exp}^*$ und $id \in \text{Id}$ ist $r[e/id]$ eindeutig bestimmt.

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

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) $\forall e' \in \text{Exp}, e \in \text{Exp}^*, id \in \text{Id} : id \notin \text{free}(e') \Rightarrow e' = e'[e/id]$
- (b) $\forall r \in \text{Row}, e \in \text{Exp}^*, id \in \text{Id} : id \notin \text{free}(r) \Rightarrow r = r[e/id]$

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 $id \notin \text{free}(id') = \{id'\}$, also insbesondere $id'[e/id] = id'$ wegen Definition 2.8.
- 3.) Für Applikationen $e_1 e_2$ gilt $id \notin \text{free}(e_1)$ und $id \notin \text{free}(e_2)$. 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 $(\mathbf{object} (self) r \mathbf{end})[e/id] = \mathbf{object} (self) r \mathbf{end}$ bereits aus Definition 2.8.

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

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

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

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

Für $a \neq id$ ist $(\mathbf{val} a = e'; r)[e/id] = \mathbf{val} a = e'[e/id]; r[e/id]$ 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 $(\mathbf{val} a = e'; r)[e/id] = \mathbf{val} a = e'; r$.

Die übrigen Fälle verlaufen analog. □

Korollar 2.1

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

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

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, $e \rightarrow_e e'$ mit $e, e' \in Exp$ oder $r \rightarrow_r r'$ mit $r, r' \in Row$, heißt *gültig* für \mathcal{L}_o , wenn er sich mit den small step Regeln von \mathcal{L}_f (vgl. Definition 1.15), sowie den folgenden small step Regeln für Objekte

$$\begin{array}{ll}
 \text{(OBJECT-EVAL)} & \frac{r \rightarrow_r r'}{\mathbf{object}(self) \ r \ \mathbf{end} \rightarrow_e \mathbf{object}(self) \ r' \ \mathbf{end}} \\
 \text{(SEND-EVAL)} & \frac{e \rightarrow_e e'}{e \# m \rightarrow_e e' \# m} \\
 \text{(SEND-UNFOLD)} & \mathbf{object}(self) \ \omega \ \mathbf{end} \# m \rightarrow_e \omega[\mathbf{object}(self) \ \omega \ \mathbf{end} /_{self}] \# m \\
 \text{(SEND-ATTR)} & (\mathbf{val} \ a = v; \omega) \# m \rightarrow_e \omega[v/a] \# m \\
 \text{(SEND-SKIP)} & (\mathbf{method} \ m' = e; \omega) \# m \rightarrow_e \omega \# m \quad \text{falls } m \neq m' \vee m \in \text{dom}_m(\omega) \\
 \text{(SEND-EXEC)} & (\mathbf{method} \ m' = e; \omega) \# m \rightarrow_e e \quad \text{falls } m = m' \wedge m \notin \text{dom}_m(\omega)
 \end{array}$$

und den small step Regeln für Reihen

$$\begin{array}{ll}
 \text{(ATTR-LEFT)} & \frac{e \rightarrow_e e'}{\mathbf{val} \ a = e; \ r \rightarrow_r \mathbf{val} \ a = e'; \ r} \\
 \text{(ATTR-RIGHT)} & \frac{r \rightarrow_r r'}{\mathbf{val} \ a = v; \ r \rightarrow_r \mathbf{val} \ a = v; \ r'} \\
 \text{(METHOD-RIGHT)} & \frac{r \rightarrow_r r'}{\mathbf{method} \ m = e; \ r \rightarrow_r \mathbf{method} \ m = e; \ r'}
 \end{array}$$

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 Exp^* ist, ansonsten bleibt der Interpreter stecken.

Im Folgenden betrachten wir \rightarrow_e als eine Relation

$$\rightarrow_e \subseteq Exp \times Exp$$

und \rightarrow_r als eine Relation

$$\rightarrow_r \subseteq Row \times Row$$

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 Induktionsvoraussetzung nicht möglich ist.

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

Für $\omega = \mathbf{val}\ a = v';\ \omega'$ 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 $\omega = \mathbf{method}\ m = e;\ \omega'$, in dem nur ein small step mit (METHOD-RIGHT) in Frage kommt. Nach Induktionsvoraussetzung existiert aber kein small step für ω' . \square

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 $e_1 \rightarrow e_2 \rightarrow \dots$
- (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 wichtigsten 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 $e \in \text{Exp}$ existiert höchstens ein $e' \in \text{Exp}$ mit $e \rightarrow e'$ und für jede Reihe $r \in \text{Row}$ existiert höchstens ein $r' \in \text{Row}$ mit $r \rightarrow r'$.*

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 $e_1 \rightarrow e'_1$ 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 $e_1 \not\rightarrow$, aber ein small step $e_2 \rightarrow e'_2$ 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 $e_1 \not\rightarrow$ und $e_2 \not\rightarrow$. 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 **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 **let** $x = e_1$ **in** e_2 .

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

Existiert ein small step $e_1 \rightarrow e'_1$ 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 **let** $x = e'_1$ **in** e_2 eindeutig bestimmt durch **let** $x = e_1$ **in** e_2 .

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

- 5.) e ist von der Form **if** e_0 **then** e_1 **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 $e_0 \rightarrow e'_0$ 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 **if** e'_0 **then** e_1 **else** e_2 eindeutig bestimmt durch **if** e_0 **then** e_1 **else** e_2 . ■

Für $e_0 \not\rightarrow$ 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 $e_1 \rightarrow e'_1$, 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 $e_1 \not\rightarrow$, 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 $e \in Val$ 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 $e \in Val$ und $r_1 \in RVal$ 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 $r_1 \in RVal$ 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 $e_1 \rightarrow \dots \rightarrow e_n$ ist das Resultat e_n eindeutig durch e_1 bestimmt.

Beweis: Folgt trivialerweise aus der Eindeutigkeit des Übergangsschritts. \square

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 Erleichterung².

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 $e \in \text{Exp}, v \in \text{Val}$ oder $r \Downarrow_r \omega$ mit $r \in \text{Row}, \omega \in \text{RVal}$.

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 $e \in \text{Exp}, v \in \text{Val}$ oder $r \Downarrow_r \omega$ mit $r \in \text{Row}, \omega \in \text{RVal}$, heisst *gültig*, wenn er sich mit den big step Regeln für die funktionale Sprache \mathcal{L}_f

²Teilweise 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)	$\frac{e_0 \Downarrow_e \mathbf{true} \quad e_1 \Downarrow_e v}{\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \Downarrow_e v}$
(COND-FALSE)	$\frac{e_0 \Downarrow_e \mathbf{false} \quad e_2 \Downarrow_e v}{\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \Downarrow_e v}$

sowie den big step Regeln für Objekte

(OBJECT)	$\frac{r \Downarrow_r \omega}{\mathbf{object}\ (self)\ r\ \mathbf{end} \Downarrow_e \mathbf{object}\ (self)\ \omega\ \mathbf{end}}$
(SEND)	$\frac{e \Downarrow_e \mathbf{object}\ (self)\ \omega\ \mathbf{end} \quad \omega[\mathbf{object}\ (self)\ \omega\ \mathbf{end}/_{self}] \# m \Downarrow_e v}{e \# m \Downarrow_e v}$
(SEND-ATTR)	$\frac{\omega[v/a] \# m \Downarrow_e v'}{(\mathbf{val}\ a = v; \omega) \# m \Downarrow_e v'}$
(SEND-SKIP)	$\frac{m' \neq m \vee m \in \text{dom}_m(\omega) \quad \omega \# m \Downarrow_e v}{(\mathbf{method}\ m' = e; \omega) \# m \Downarrow_e v}$
(SEND-EXEC)	$\frac{m' = m \wedge m \notin \text{dom}_m(\omega) \quad e \Downarrow_e v}{(\mathbf{method}\ m' = e; \omega) \# m \Downarrow_e v}$

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 $\Downarrow_e \subseteq \text{Exp} \times \text{Val}$ und $\Downarrow_r \subseteq \text{Row} \times \text{RVal}$ 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 Semantiken 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) $\forall e \in \text{Exp} : \forall v \in \text{Val} : e \Downarrow v \Leftrightarrow e \xrightarrow{*} v$
(b) $\forall r \in \text{Row} : \forall \omega \in \text{RVal} : r \Downarrow \omega \Leftrightarrow r \xrightarrow{*} \omega$

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.

„ \Rightarrow “ 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 $op\ n_1\ n_2 \xrightarrow{*} op^I(n_1, n_2)$ 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 $(\lambda x.e)\ v \xrightarrow{*} e[v/x] \xrightarrow{*} v'$.
- 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 $e_1\ e_2 \xrightarrow{*} v_1\ v_2 \xrightarrow{*} v$.

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 $e \Downarrow \mathbf{object}(self) \ \omega \ \mathbf{end}$ und $\omega[\mathbf{object}(self) \ \omega \ \mathbf{end} /_{self}] \# m \Downarrow v$, also gilt nach Induktionsvoraussetzung $e \xrightarrow{*} \mathbf{object}(self) \ \omega \ \mathbf{end}$ und $\omega[\mathbf{object}(self) \ \omega \ \mathbf{end} /_{self}] \# m \xrightarrow{*} v$. Wegen Regel (SEND-UNFOLD) gilt $\mathbf{object}(self) \ \omega \ \mathbf{end} \rightarrow \omega[\mathbf{object}(self) \ \omega \ \mathbf{end} /_{self}] \# m$, weiter existiert eine Berechnungsfolge $e \# m \xrightarrow{*} \mathbf{object}(self) \ \omega \ \mathbf{end} \# m$ mit small step Regel (SEND-EVAL), also existiert insgesamt eine Berechnung

$$\begin{aligned} & e \# m \\ & \xrightarrow{*} \mathbf{object}(self) \ \omega \ \mathbf{end} \# m \\ & \rightarrow \omega[\mathbf{object}(self) \ \omega \ \mathbf{end} /_{self}] \# m \\ & \xrightarrow{*} v. \end{aligned}$$

Die restlichen Fälle sind ebenso einfach zu beweisen.

„ \Leftarrow “ 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 $e_1, e_2 \in Val$, 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

$$(\lambda x. e'_1) e_2 \rightarrow e'_1[e_2/x] \xrightarrow{n} v$$

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 $v_1, v_2 \in Val$, 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 $e_1 \in Val$ bzw. $e_2 \in Val$, 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 $e = \mathbf{if } e_0 \mathbf{ then } e_1 \mathbf{ else } e_2$ ist die Berechnungsfolge $e \xrightarrow{*} v$ von der Form

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

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

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

entweder mit small step Regel (COND-TRUE), für $v_0 = \text{true}$, oder mit small step Regel (COND-FALSE), für $v_0 = \text{false}$, beginnen muss. Für $e_0 \xrightarrow{*} v_0$ existiert nach Induktionsvoraussetzung ein big step $e_0 \Downarrow v_0$.

Wenn $e_0 = \text{true}$, dann hat die restliche Berechnungsfolge die Form

$$\mathbf{if } \text{true} \mathbf{ then } e_1 \mathbf{ else } e_2 \rightarrow e_1 \xrightarrow{*} v$$

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

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

mit big step Regel (COND-TRUE).

Entsprechend folgt für $v_0 = \text{false}$ 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 $e \in Exp, v \in Val$, 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. \square

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.

structural 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

$$\begin{array}{ll} \tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit} & \text{Basistypen} \\ \mid \tau_1 \rightarrow \tau_2 & \text{Funktionstypen} \\ \mid \langle \phi \rangle & \text{Objektypen} \end{array}$$

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

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

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.

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

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 $m_1, m_2 \in Method$, $\tau_1, \tau_2 \in Type$ und $\phi \in RType$.

TODO: Überleitung

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

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

und

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

ist definiert als

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

mit

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

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

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

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

und

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

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

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.

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

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) $b :: \mathbf{bool}$
- (INT) $n :: \mathbf{int}$
- (UNIT) $() :: \mathbf{unit}$
- (AOP) $op :: \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ falls $op \in \{+, -, *\}$
- (ROP) $op :: \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ 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 $\Gamma : Id \rightarrow Type$ mit endlichem Definitionsbereich.
- (b) Sei Γ eine Typumgebung, dann bezeichnet Γ^* die Typumgebung mit den folgenden Eigenschaften.
 - $dom(\Gamma^*) = dom(\Gamma) \cap Var$
 - $\forall id \in dom(\Gamma^*) : \Gamma^*(id) = \Gamma(id)$
- (c) Sei Γ eine Typumgebung, dann bezeichnet Γ^+ die Typumgebung mit den folgenden Eigenschaften.
 - $dom(\Gamma^+) = dom(\Gamma) \setminus dom(\Gamma^*)$
 - $\forall id \in dom(\Gamma^+) : \Gamma^+(id) = \Gamma(id)$
- (d) Die Menge $TEnv = \{\Gamma \mid \Gamma \text{ ist eine Typumgebung}\}$ enthält alle Typumgebungen.

TODO: Prosa.

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

- (a) $dom(\Gamma) = \{id_1, \dots, id_n\}$ und
- (b) $\forall i \in \{1, \dots, n\} : \Gamma(id_i) = \tau_i$.

Die Listenschreibweise verdeutlicht, dass eine Typumgebung letztlich nichts anderes ist als eine „Tabelle“, in der die Typen der bereits bekannten Namen eingetragen sind³. 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)

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

- (a) Ein *Typurteil für Ausdrücke* ist eine Formel der Gestalt $\Gamma \triangleright_e e :: \tau$ mit $\Gamma : Id \multimap Type$, $e \in Exp$ und $\tau \in Type$.
- (b) Ein *Typurteil für Reihen* ist eine Formel der Gestalt $\Gamma \triangleright_r r :: \phi$ mit $\Gamma : Id \multimap Type$, $r \in Row$ und $\phi \in RType$.

TODO: Prosa

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

$$\begin{aligned}
 (\text{ID}) \quad & \Gamma \triangleright_e id :: \tau \quad \text{falls } id \in \text{dom}(\Gamma) \wedge \Gamma(id) = \tau \\
 (\text{CONST}) \quad & \frac{c :: \tau}{\Gamma \triangleright_e c :: \tau} \\
 (\text{APP}) \quad & \frac{\Gamma \triangleright_e e_1 :: \tau_1 \rightarrow \tau \quad \Gamma \triangleright_e e_2 :: \tau_1}{\Gamma \triangleright_e e_1 e_2 :: \tau} \\
 (\text{ABSTR}) \quad & \frac{\Gamma[\tau/x] \triangleright_e e :: \tau'}{\Gamma \triangleright_e \lambda x : \tau. e :: \tau \rightarrow \tau'} \\
 (\text{REC}) \quad & \frac{\Gamma[\tau/x] \triangleright_e e :: \tau}{\Gamma \triangleright_e \mathbf{rec} x : \tau. e :: \tau} \\
 (\text{LET}) \quad & \frac{\Gamma \triangleright_e e_1 :: \tau_1 \quad \Gamma[\tau_1/x] \triangleright_e e_2 :: \tau_2}{\Gamma \triangleright_e \mathbf{let} x = e_1 \mathbf{in} e_2 :: \tau_2} \\
 (\text{COND}) \quad & \frac{\Gamma \triangleright_e e_0 :: \mathbf{bool} \quad \Gamma \triangleright_e e_1 :: \tau \quad \Gamma \triangleright_e e_2 :: \tau}{\Gamma \triangleright_e \mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 :: \tau}
 \end{aligned}$$

sowie den Typregeln für Objekte

$$\begin{aligned}
 (\text{SEND}) \quad & \frac{\Gamma \triangleright_e e :: \langle m : \tau; \phi \rangle}{\Gamma \triangleright_e e \# m :: \tau} \\
 (\text{SEND}') \quad & \frac{\Gamma \triangleright_r \omega :: (m : \tau; \phi)}{\Gamma \triangleright_e \omega \# m :: \tau} \\
 (\text{OBJECT}) \quad & \frac{\Gamma^*[\tau/self] \triangleright_r r :: \phi \quad \tau = \langle \phi \rangle}{\Gamma \triangleright_e \mathbf{object} (self : \tau) r \mathbf{end} :: \tau} \\
 (\text{DUPL}) \quad & \frac{\Gamma \triangleright_e self :: \tau \quad \forall i = 1 \dots n : \Gamma \triangleright_e a_i :: \tau_i \wedge \Gamma \triangleright_e e_i :: \tau_i}{\Gamma \triangleright_e \{ \langle a_1 = e_1; \dots; a_n = e_n \rangle \} :: \tau}
 \end{aligned}$$

und den Typregeln für Reihen

$$\begin{array}{lcl}
 (\text{EMPTY}) & \Gamma \triangleright_r e :: \emptyset \\
 (\text{ATTR}) & \frac{\Gamma^* \triangleright_e e :: \tau \quad \Gamma[\tau/a] \triangleright_r r_1 :: \phi}{\Gamma \triangleright_r \mathbf{val} \ a = e; r_1 :: \phi} \\
 (\text{METHOD}) & \frac{\Gamma \triangleright_e e :: \tau \quad \Gamma \triangleright_r r_1 :: \phi}{\Gamma \triangleright_r \mathbf{method} \ m = e; r_1 :: (m : \tau; \emptyset) \oplus \phi}
 \end{array}$$

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 $\Gamma \triangleright_e e :: \tau$ kurz $\Gamma \triangleright e :: \tau$ und statt $\Gamma \triangleright_r r :: \phi$ kurz $\Gamma \triangleright r :: \phi$, 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 $e \in \text{Exp}$ heisst *wohlgetyp* in Γ , wenn es ein $\tau \in \text{Type}$ gibt, so dass gilt: $\Gamma \triangleright e :: \tau$.
- (b) Eine Reihe $r \in \text{Row}$ heisst *wohlgetyp* in Γ , wenn es ein $\phi \in \text{RType}$ gibt, so dass gilt: $\Gamma \triangleright r :: \phi$.

TODO: Prosa

Satz 2.3 (Typeindeutigkeit)

- (a) Für jede Typumgebung Γ und jeden Ausdruck $e \in \text{Exp}$ existiert höchstens ein Typ $\tau \in \text{Type}$ mit $\Gamma \triangleright e :: \tau$.
- (b) Für jede Typumgebung Γ und jede Reihe $r \in \text{Row}$ existiert höchstens ein Reihentyp $\phi \in \text{RType}$ mit $\Gamma \triangleright r :: \phi$.

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. \square

Korollar 2.4 Sei Γ eine Typumgebung.

- (a) Ist $e \in \text{Exp}$ wohlgetyp in Γ , dann existiert genau ein $\tau \in \text{Type}$ mit $\Gamma \triangleright e :: \tau$.

(b) Ist $r \in \text{Row}$ wohlgetypt in Γ , dann existiert genau ein $\phi \in R\text{Type}$ 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 arbeitet⁴, 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.

$$\begin{array}{l} v ::= \lambda x : \tau. e \\ \quad | \text{object}(self : \tau) \omega \text{ end} \end{array}$$

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. ■

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

(BETA-V)	$(\lambda x : \tau. e) v \rightarrow_e e[v/x]$
(UNFOLD)	$\mathbf{rec} x : \tau. e \rightarrow_e e[\mathbf{rec} x : \tau. e / x]$
(OBJECT-EVAL)	$\frac{r \rightarrow_r r'}{\mathbf{object} (self : \tau) r \mathbf{end} \rightarrow_e \mathbf{object} (self : \tau) r' \mathbf{end}}$
(SEND-UNFOLD)	$\mathbf{object} (self : \tau) \omega \mathbf{end} \# m \rightarrow_e \omega[\mathbf{object} (self : \tau) \omega \mathbf{end} /_{self}] \# m$

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 diesen Sachverhalt.

Definition 2.22 Für jeden Ausdruck $e \in \text{Exp}(\mathcal{L}_o^t)$ sei $\text{erase}(e) \in \text{Exp}(\mathcal{L}_o)$ der Ausdruck, der aus e durch Entfernen aller Typen entsteht⁵, also

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

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

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

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

⁵Wir 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

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

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

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. \square

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

Beweis: Trivial. \square

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:

- $A \subseteq \text{dom}(\Gamma_1)$
- $A \subseteq \text{dom}(\Gamma_2)$
- $\forall id \in A : \Gamma_1(id) = \Gamma_2(id)$

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:

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

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

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

Beweis:

- (a) Sei $B \subseteq A$. Dann folgt unmittelbar $B \subseteq A \subseteq \text{dom}(\Gamma_1)$, $B \subseteq A \subseteq \text{dom}(\Gamma_2)$ und $\forall id \in B \subseteq A : \Gamma_1(id) = \Gamma_2(id)$.
- (b) Folgt unmittelbar aus der Definition, da $\text{dom}(\Gamma^*) = \text{dom}(\Gamma) \cap \text{Var}$.
- (c) Folgt ebenso problemlos aus $A \cup \{id\} \subseteq \text{dom}(\Gamma_1[\tau/id])$, $A \cup \{id\} \subseteq \text{dom}(\Gamma_2[\tau/id])$ und $\Gamma_1[\tau/id](id) = \tau = \Gamma_2[\tau/id](id)$. \square

TODO: Überleitung

Lemma 2.5 (Typumgebungen und frei vorkommende Namen)

- (a) $\forall \Gamma \in TEnv, e \in Exp, \tau \in Type : \Gamma \triangleright e :: \tau \Rightarrow \text{free}(e) \subseteq \text{dom}(\Gamma)$
- (b) $\forall \Gamma \in TEnv, r \in Row, \phi \in RType : \Gamma \triangleright r :: \phi \Rightarrow \text{free}(r) \subseteq \text{dom}(\Gamma)$

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 $\text{free}(c) = \emptyset \subseteq \text{dom}(\Gamma)$.
- 2.) $\Gamma \triangleright id :: \tau$ mit Typregel (ID). Dann gilt $id \in \text{dom}(\Gamma)$ und $\Gamma(id) = \tau$, also folgt $\text{free}(id) = \{id\} \subseteq \text{dom}(\Gamma)$.
- 3.) $\Gamma \triangleright e_1 e_2 :: \tau$ mit Typregel (APP) kann nur aus Prämissen der Form $\Gamma \triangleright e_1 :: \tau' \rightarrow \tau$ und $\Gamma \triangleright e_2 :: \tau'$ folgen. Nach Induktionsvoraussetzung gilt $\text{free}(e_1) \subseteq \text{dom}(\Gamma)$ und $\text{free}(e_2) \subseteq \text{dom}(\Gamma)$. Wegen Definition 2.5 folgt schliesslich $\text{free}(e_1 e_2) \subseteq \text{dom}(\Gamma)$.
- 4.) $\Gamma \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau$ mit Typregel (OBJECT) kann ausschliesslich aus $\Gamma^*[\tau/self] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen. Nach Induktionsvoraussetzung gilt $\text{free}(r) \subseteq \text{dom}(\Gamma^*[\tau/self])$ und wegen $\text{dom}(\Gamma^*[\tau/self]) \subseteq \text{dom}(\Gamma) \cup \{self\}$ folgt $\text{free}(r) \setminus \{self\} \subseteq \text{dom}(\Gamma)$. Vermöge Definition 2.5 gilt folglich $\text{free}(\mathbf{object}(self : \tau) r \mathbf{end}) \subseteq \text{dom}(\Gamma)$.
- 5.) $\Gamma \triangleright \{\langle a_i = e_i^{i=1..n} \rangle\} :: \tau$ 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 $i = 1, \dots, n$ folgen. Nach Induktionsvoraussetzung gilt dann einerseits $\text{free}(self) \subseteq \text{dom}(\Gamma)$, also insbesondere $self \in \text{dom}(\Gamma)$, und andererseits $\text{free}(a_i) \subseteq \text{dom}(\Gamma)$ und $\text{free}(e_i) \subseteq \text{dom}(\Gamma)$ für $i = 1, \dots, n$. Nach Definition 2.5 folgt somit $\text{free}(\{\langle a_i = e_i^{i=1..n} \rangle\}) \subseteq \text{dom}(\Gamma)$.

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

Die restlichen Fälle verlaufen analog. \square

Insbesondere folgt, dass Ausdrücke, die in einer Typumgebung Γ^* 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 $\forall \Gamma \in TEnv, e \in Exp, \tau \in Type : \Gamma^* \triangleright e :: \tau \Rightarrow e \in Exp^*$

Beweis: Folgt unmittelbar aus Lemma 2.5. \square

TODO: Überleitung

Lemma 2.6 (Koinzidenzlemma für \mathcal{L}_o^t)

- (a) $\forall \Gamma_1, \Gamma_2 \in TEnv, e \in Exp, \tau \in Type : \Gamma_1 \triangleright e :: \tau \wedge \Gamma_1 =_{free(e)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright e :: \tau$
(b) $\forall \Gamma_1, \Gamma_2 \in TEnv, r \in Row, \phi \in RType : \Gamma_1 \triangleright r :: \phi \wedge \Gamma_1 =_{free(r)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright r :: \phi$

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.) $\Gamma_1 \triangleright id :: \tau$ mit Typregel (ID) kann ausschliesslich aus $id \in dom(\Gamma_1)$ und $\Gamma_1(id) = \tau$ folgen. Da $free(id) = \{id\}$ gilt nach Definition 2.23 aber ebenfalls $id \in dom(\Gamma_2)$ und $\Gamma_2(id) = \Gamma_1(id) = \tau$. Also folgt mit Typregel (ID) $\Gamma_2 \triangleright id :: \tau$.
- 3.) $\Gamma_1 \triangleright e_1 e_2 :: \tau$ mit Typregel (APP) kann nur aus Prämissen der Form $\Gamma_1 \triangleright e_1 :: \tau' \rightarrow \tau$ und $\Gamma_1 \triangleright e_2 :: \tau'$ folgen. Wegen Lemma 2.4 gilt nach Induktionsvoraussetzung $\Gamma_2 \triangleright e_1 :: \tau' \rightarrow \tau$ und $\Gamma_2 \triangleright e_2 :: \tau'$, also $\Gamma_2 \triangleright e_1 e_2 :: \tau$ mit Typregel (APP).

- 4.) $\Gamma_1 \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \tau$ mit Typregel (OBJECT) kann nur aus $\Gamma_1^*[\tau/self] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen. Wegen Lemma 2.4 folgt mit Induktionsvoraussetzung $\Gamma_2^*[\tau/self] \triangleright r :: \phi$, und mit Typregel (OBJECT) folgt daraus $\Gamma_2 \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \tau$.
- 5.) $\Gamma_1 \triangleright \{ \langle a_i = e_i^{i=1\dots n} \rangle \} :: \tau$ mit Typregel (DUPL) kann nur aus $\Gamma_1 \triangleright self :: \tau$ und $\Gamma_1 \triangleright a_i :: \tau_i \wedge \Gamma_1 \triangleright e_i :: \tau_i$ für $i = 1, \dots, n$ folgen. Mit Lemma 2.4 gilt nach Induktionsvoraussetzung $\Gamma_2 \triangleright self :: \tau$ und $\Gamma_2 \triangleright a_i :: \tau_i \wedge \Gamma_2 \triangleright e_i :: \tau_i$ für alle $i = 1, \dots, n$. Also folgt mit Typregel (DUPL) $\Gamma_2 \triangleright \{ \langle a_i = e_i^{i=1\dots n} \rangle \} :: \tau$.
- 6.) Für $\Gamma_1 \triangleright \mathbf{method} m = e; r :: (m : \tau; \emptyset) \oplus \phi$ 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. \square

Korollar 2.7

- (a) $\forall \Gamma \in TEnv, e \in Exp, \tau \in Type : \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright e :: \tau$.
- (b) $\forall \Gamma \in TEnv, r \in Row, \phi \in RType : \Gamma^* \triangleright r :: \phi \Rightarrow \Gamma \triangleright r :: \phi$.

Beweis: Folgt mit Lemma 2.5 unmittelbar aus Lemma 2.6. \square

TODO: Überleitung

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

- (a) $\forall e' \in Exp : \forall \tau' \in Type : \Gamma[\tau/id] \triangleright e' :: \tau' \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright e'[e/id] :: \tau'$
- (b) $\forall r \in Row : \forall \phi \in RType : \Gamma[\tau/id] \triangleright r :: \phi \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright r[e/id] :: \phi$

Beweis: Zunächst ist festzuhalten, dass vermöge Lemma 2.5 die Substitution $e'[e/id]$ bzw. $r[e/id]$ stets definiert ist. Damit können wir dann den Beweis durch simultane 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 zuletzt angewandten Typregel führen. Wir betrachten dazu die folgenden Fälle.

- 1.) $\Gamma[\tau/id] \triangleright c :: \tau'$ mit Typregel (CONST).

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

2.) $\Gamma[\tau/id] \triangleright id' :: \tau'$ mit Typregel (ID).

Nach Voraussetzung gilt $id' \in \text{dom}(\Gamma)$ 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 $\text{free}(e) \subseteq \text{dom}(\Gamma^*) \subseteq \text{dom}(\Gamma)$ 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 $\Gamma[\tau/id] \triangleright e_1 :: \tau'' \rightarrow \tau'$ und $\Gamma[\tau/id] \triangleright e_2 :: \tau''$. Mit Induktionsvoraussetzung folgt $\Gamma \triangleright e_1[e/id] :: \tau'' \rightarrow \tau'$ 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.) $\Gamma[\tau/id] \triangleright \lambda x : \tau'. e' :: \tau' \rightarrow \tau''$ mit Typregel (ABSTR).

Dann gilt $\Gamma[\tau/id][\tau'/x] \triangleright e' :: \tau''$ und durch gebundene Umbenennung lässt sich die Voraussetzung $x \notin \{id\} \cup \text{free}(e)$ 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 $\Gamma \triangleright \lambda x : \tau'. e'[e/id] :: \tau' \rightarrow \tau''$ und wegen $id \notin \{id\} \cup \text{free}(e)$ mit Definition 2.8 schliesslich $\Gamma \triangleright (\lambda x : \tau'. e')[e/id] :: \tau' \rightarrow \tau''$.

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

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

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

$$\Gamma \triangleright r :: \phi \wedge \Gamma^* \triangleright r(a) :: \tau \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright r\langle^e/a\rangle :: \phi$$

Beweis: Wegen $\Gamma^* \triangleright e :: \tau$ ist nach Lemma 2.5 $e \in Exp^*$, und wegen $\Gamma^* \triangleright r(a) :: \tau$ 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 $\Gamma^* \triangleright \epsilon(a) :: \tau$ nicht gelten.

2.) $\Gamma \triangleright \mathbf{val} \ a' = e'; r' :: \phi$ kann nur mit Typregel (ATTR) aus $\Gamma^* \triangleright e' :: \tau'$ 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 $(\Gamma[\tau'/a'])^* = \Gamma^*$ folgt nach Induktionsvoraussetzung $\Gamma[\tau'/a'] \triangleright r'\langle^e/a\rangle :: \phi$, 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 $\Gamma \triangleright (\mathbf{method} \ m = e'; r')\langle^e/a\rangle :: (m : \tau'; \emptyset) \oplus \phi$. \square

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) $\Gamma^* \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \tau$
 (3) $\forall a \in \text{dom}(\Gamma) \cap \text{Attribute}, \tau_a \in \text{Type} : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright e[\mathbf{object} (self:\tau) r \mathbf{end} /_{self}] :: \tau'$

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

- (1) $\Gamma[\tau /_{self}] \triangleright r' :: \phi$
 (2) $\Gamma^* \triangleright \mathbf{object} (self : \tau) r \oplus r' \mathbf{end} :: \tau$
 (3) $\forall a \in \text{dom}(\Gamma) \cap \text{Attribute}, \tau_a \in \text{Type} : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright r'[\mathbf{object} (self:\tau) r \oplus r' \mathbf{end} /_{self}] :: \phi$

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 $\Gamma[\tau /_{self}] \triangleright e :: \tau'$ und $\Gamma[\tau /_{self}] \triangleright r' :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel. Dazu betrachten wir exemplarisch die folgenden Fälle.

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

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

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

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

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

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

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

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

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

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

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

Dann gilt $\Gamma \triangleright e_i[\tilde{o}/self] :: \tau_i$ nach Induktionsvoraussetzung für alle $i = 1, \dots, n$.

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

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

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

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

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

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

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

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

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

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 $e \rightarrow e'$, dann läßt sich auch für e' in der Typumgebung Γ der Typ τ herleiten.

Satz 2.5 (Typerhaltung, „Preservation“)

- (a) $\forall \Gamma \in TEnv : \forall e, e' \in Exp : \forall \tau \in Type : \Gamma^* \triangleright e :: \tau \wedge e \rightarrow e' \Rightarrow \Gamma^* \triangleright e' :: \tau$
- (b) $\forall \Gamma \in TEnv : \forall r, r' \in Row : \forall \phi \in RType : \Gamma \triangleright r :: \phi \wedge r \rightarrow r' \Rightarrow \Gamma \triangleright r' :: \phi$

Wir wählen Γ^* statt Γ , da die Substitution nur definiert ist, wenn $e \in Exp^*$, also $\Gamma^* \triangleright e :: \tau$. 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 $e \rightarrow e'$ und $r \rightarrow r'$, und Fallunterscheidung nach der zuletzt angewandten small step Regel. Wir betrachten dazu exemplarisch die folgenden Fälle.

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

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

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

- 2.) $(\lambda x : \tau.e)\ v \rightarrow e[v/x]$ mit (BETA-V).

$\Gamma^* \triangleright (\lambda x : \tau.e)\ v :: \tau'$ kann nur mit Typregel (APP) aus Prämissen der Form $\Gamma^* \triangleright v :: \tau$ und $\Gamma^* \triangleright \lambda x : \tau.e :: \tau \rightarrow \tau'$ folgen. Letzteres kann wiederum nur mit Typregel (ABSTR) aus $\Gamma^*[x/\tau] \triangleright e :: \tau'$ folgen. Mit Lemma 2.7 folgt daraus $\Gamma^* \triangleright e[v/x] :: \tau'$.

- 3.) $e_1\ e_2 \rightarrow e'_1\ e_2$ mit (APP-LEFT) aus $e_1 \rightarrow e'_1$.

$\Gamma^* \triangleright e_1\ e_2 :: \tau$ kann nur mit Typregel (APP) aus $\Gamma^* \triangleright e_2 :: \tau'$ und $\Gamma^* \triangleright e_1 :: \tau' \rightarrow \tau$ folgen. Nach Induktionsvoraussetzung gilt $\Gamma^* \triangleright e'_1 :: \tau' \rightarrow \tau$ und mit Typregel (APP) folgt dann $\Gamma^* \triangleright e'_1\ e_2 :: \tau$.

- 4.) $\mathbf{object}(self : \tau)\ r\ \mathbf{end} \rightarrow \mathbf{object}(self : \tau)\ r'\ \mathbf{end}$ mit (OBJECT-EVAL) aus $r \rightarrow r'$.

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

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

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

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

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

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

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

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

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

Die übrigen Fälle verlaufen analog. □

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

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 $v \in Val$ mit $[] \triangleright v :: \mathbf{int}$ gilt $v \in Int$.
- (b) Für alle $v \in Val$ mit $[] \triangleright v :: \tau \rightarrow \tau'$ 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. $v = \lambda x : \tau. e$ mit $x \in Var$ 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: Trivial. □

Satz 2.6 (Existenz des Übergangsschritts, „Progress“)

- (a) $\forall e \in Exp, \tau \in Type : [] \triangleright e :: \tau \Rightarrow (e \in Val \vee \exists e' \in Exp : e \rightarrow e')$
- (b) $\forall \Gamma \in TEnv, r \in Row, \phi \in RType : \Gamma^+ \triangleright r :: \phi \Rightarrow (r \in RVal \vee \exists r' \in Row : r \rightarrow r')$

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 $c \in Val$.
- 2.) Für Namen id existiert kein τ mit $[] \triangleright id :: \tau$.
- 3.) Für Abstraktionen $\lambda id : \tau'. e'$ gilt $(\lambda id : \tau'. e') \in Val$.
- 4.) Für Applikationen $e_1\ e_2$ kann $[] \triangleright e_1\ e_2 :: \tau$ nur mit Typregel (APP) aus Prämissen der Form $[] \triangleright e_1 :: \tau' \rightarrow \tau$ und $[] \triangleright e_2 :: \tau'$ folgen.

Wenn $e_1 \notin Val$, dann existiert nach Induktionsvoraussetzung ein e'_1 mit $e_1 \rightarrow e'_1$. Folglich existiert mit Regel (APP-LEFT) ein small step $e_1\ e_2 \rightarrow e'_1\ e_2$.

Andererseits existiert für $e_1 \in Val, e_2 \notin Val$ nach Induktionsvoraussetzung ein e'_2 mit $e_2 \rightarrow e'_2$, und somit ein small step $e_1\ e_2 \rightarrow e_1\ e'_2$ mit Regel (APP-RIGHT).

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

- 1.) Für $e_1 \in Op$ ist $e_1 e_2 \in Val$.
 - 2.) Für $e_1 = op\ v_1$ ist zunächst zu bemerken, dass op nur vom Typ $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ oder $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ sein kann. Mit dem Canonical Forms Lemma (2.10) folgt aus $[] \triangleright v_1 :: \mathbf{int}$ und $[] \triangleright e_2 :: \mathbf{int}$, dass $v_1, e_2 \in Int$ gilt, und somit existiert ein small step $op\ v_1\ e_2 \rightarrow op^I(v_1, e_2)$ mit Regel (OP).
 - 3.) Für $e_1 = \lambda x : \tau'_2. e'_1$ existiert ein small step $(\lambda x : \tau'_2. e'_1)\ e_2 \rightarrow e'_1[e_2/x]$ mit Regel (BETA-V), da nach Voraussetzung $e_2 \in Val$.
- 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 $(\mathbf{val}\ a = v; \omega') \# m \rightarrow \omega'[v/a] \# m$.
- Für $\omega = (\mathbf{method}\ m' = e; \omega')$ mit $m \neq m' \vee m \in dom_m(\omega')$ existiert ein small step $(\mathbf{method}\ m' = e; \omega') \# m \rightarrow \omega' \# m$ mit Regel (SEND-SKIP).
- Für $\omega = (\mathbf{method}\ m = e; \omega')$ mit $m \notin dom_m(\omega')$ existiert mit Regel (SEND-EXEC) ein small step $(\mathbf{method}\ m = e; \omega') \# m \rightarrow e$.
- 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 $e \notin Val$ existiert nach Induktionsvoraussetzung ein e' mit $e \rightarrow e'$, und somit ein small step $e \# m \rightarrow e' \# m$ mit Regel (SEND-EVAL).
- Für $e \in Val$ gilt nach Lemma 2.10 $e = \mathbf{object}\ (self : \tau)\ \omega\ \mathbf{end}$. Somit existiert ein small step $\mathbf{object}\ (self : \tau)\ \omega\ \mathbf{end} \# m \rightarrow \omega[\mathbf{object}\ (self : \tau)\ \omega\ \mathbf{end} / self] \# m$ mit Regel (SEND-UNFOLD).
- 7.) Für Objekte $\mathbf{object}\ (self : \tau)\ r\ \mathbf{end}$ kann $[] \triangleright \mathbf{object}\ (self : \tau)\ r\ \mathbf{end} :: \tau$ ausschliesslich mit Typregel (OBJECT) aus Prämissen der Form $[self : \tau] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen.
- Falls $r \notin RVal$, dann existiert nach Induktionsvoraussetzung ein r' mit $r \rightarrow r'$. Also existiert ein small step $\mathbf{object}\ (self : \tau)\ r\ \mathbf{end} \rightarrow \mathbf{object}\ (self : \tau)\ r'\ \mathbf{end}$ mit Regel (OBJECT-EVAL).
- Ist andererseits $r \in RVal$, dann gilt $(\mathbf{object}\ (self : \tau)\ r\ \mathbf{end}) \in Val$.
- 8.) Für Attributdeklarationen $\mathbf{val}\ a = e; r$ kann $\Gamma^+ \triangleright \mathbf{val}\ a = e; r :: \phi$ nur mit Typregel (ATTR) aus $(\Gamma^+)^* \triangleright e :: \tau$ und $\Gamma^+[\tau / a] \triangleright r :: \phi$ folgen.

Für $e \notin \text{Val}$ existiert wegen $(\Gamma^+)^* = []$ nach Induktionsvoraussetzung ein e' mit $e \rightarrow e'$. Also existiert ein small step $\mathbf{val} \ a = e; r \rightarrow \mathbf{val} \ a = e'; r$ mit Regel (ATTR-LEFT).

Für $e \in \text{Val}, r \notin R\text{Val}$ existiert nach Induktionsvoraussetzung ein r' mit $r \rightarrow r'$, und folglich ein small step $\mathbf{val} \ a = e; r \rightarrow \mathbf{val} \ a = e; r'$ mit Regel (ATTR-RIGHT).

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

- 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 $r \notin R\text{Val}$, dann existiert nach Induktionsvoraussetzung ein r' mit $r \rightarrow r'$, also auch ein small step $\mathbf{method} \ m = e; r \rightarrow \mathbf{method} \ m = e; r'$ mit Regel (METHOD-RIGHT).

Für $r \in R\text{Val}$ ist $(\mathbf{method} \ m = e; r) \in R\text{Val}$.

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 $e = e_0 \rightarrow \dots \rightarrow e_n$ mit $e_n \not\rightarrow$ und $e_n \notin \text{Val}$ für ein $n \geq 0$. Wegen $[] \triangleright e :: \tau$ folgt mit Satz 2.5 induktiv $[] \triangleright e_i :: \tau$ für alle $i = 1, \dots, n$. Also gilt insbesondere $[] \triangleright e_n :: \tau$, aber $e_n \notin \text{Val}$ und $e_n \not\rightarrow$. Dies steht im Widerspruch zu Satz 2.6. □

TODO: Prosa

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

- (a) *Es existiert ein $v \in \text{Val}$ mit $e \xrightarrow{*} v$.*
- (b) *Die Berechnung von e divergiert.*

Beweis: Folgt unmittelbar aus Satz 2.7. □

3 Subtyping

Im vorangegangenen 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 handelt¹.

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 $Type \times Type$, die sich mit den folgenden Regeln herleiten lässt.

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

(S-REFL)	$\tau \leq \tau$
(S-TRANS)	$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$
(S-ARROW)	$\frac{\tau'_1 \leq \tau_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2}$
(S-OBJ-WIDTH)	$\langle m_1 : \tau_1; \dots; m_{n+k} : \tau_{n+k} \rangle \leq \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$
(S-OBJ-DEPTH)	$\frac{\tau_i \leq \tau'_i \text{ für } i = 1, \dots, n}{\langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle \leq \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle}$

Betrachtet man diese Definition der Subtyping-Regeln, so ist unmittelbar ersichtlich, dass die Subtyprelation selbst reflexiv und transitiv ist. Wir wollen nun zeigen, dass die Subtyprelation eine partielle Ordnung auf der Menge der Typen ist. Dazu zeigen wir zunächst, dass die Relation auch antisymmetrisch ist.

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

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 $\tau_3 \in \text{Type}$ 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 $\tau_1 = \tau'_1 \rightarrow \tau''_1$ und $\tau_2 = \tau'_2 \rightarrow \tau''_2$ 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 $\tau_3 \in \text{Type}$ 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 $\tau'_1 \rightarrow \tau''_1 = \tau'_2 \rightarrow \tau''_2$.
- 4.) $\tau_1 \leq \tau_2$ mit Regel (S-OBJ-WIDTH). Dann ist $\tau_1 = \langle m_1 : \tau'_1; \dots; m_{n+k} : \tau'_{n+k} \rangle$ und $\tau_2 = \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$ 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 $\tau_1 = \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$ und $\tau_2 = \langle m_1 : \tau''_1; \dots; m_n : \tau''_n \rangle$ aus Prämissen der Form $\tau'_i \leq \tau''_i$ für $i = 1, \dots, n$ folgen. Also kann $\tau_2 \leq \tau_1$ ausschliesslich mit den Regeln (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.) $\langle m_1 : \tau''_1; \dots; m_n : \tau''_n \rangle \leq \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$ mit Regel (S-OBJ-DEPTH) kann nur aus $\tau''_i \leq \tau'_i$ für $i = 1, \dots, n$ folgen. Nach Induktionsvoraussetzung gilt $\tau'_i = \tau''_i$ für $i = 1, \dots, n$ und es folgt $\tau_1 = \tau_2$.
 - 3.) $\langle m_1 : \tau''_1; \dots; m_n : \tau''_n \rangle \leq \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$ mit Regel (S-OBJ-WIDTH) kann nur mit $\tau''_i = \tau'_i$ für $i = 1, \dots, n$ folgen. Also ist $\tau_1 = \tau_2$. \square

Damit ist nun klar, dass die Relation \leq auch antisymmetrisch ist, und wir können die eigentlich interessante Aussage als Proposition formulieren.

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. \square

Bevor wir nun zur zentralen Aussage dieses Abschnittes kommen, dem sogenannten Subtyping-Lemma, betrachten wir zunächst noch einige weitere Eigenschaften der Subtyprelation.

Lemma 3.2 (Maximale und minimale Typen)

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

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

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 $\tau' \in Type$ 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 $\tau' \in Type$ 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 $\tau = \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$ 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. Da es sich hierbei um mehr oder weniger triviale Beweise handelt, formulieren wir die Aussagen lediglich als Korollar.

Korollar 3.1

- (a) $(Type, \leq)$ 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 $(Type, \leq)$ wäre total, dann müsste für alle Typen $\tau_1, \tau_2 \in Type$ 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 (Maximale und minimale Typen) 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 $\tau_{\top} \in Type$ 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 (Maximale und minimale Typen) 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. \square

Nach diesen Vorüberlegungen können wir nun das zentrale Lemma über die Subtyprelation \leq formulieren und beweisen. Im wesentlichen dient das Lemma einem leichteren Umgang mit der Subtyprelation, so dass in den späteren Beweisen nicht ständig die (S-TRANS)-Regel berücksichtigt werden muss.

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

- (a) $\tau = \tau'$
- (b) $\tau = \tau_1 \rightarrow \tau_2$ und $\tau' = \tau'_1 \rightarrow \tau'_2$ mit $\tau'_1 \leq \tau_1$ und $\tau_2 \leq \tau'_2$
- (c) $\tau = \langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle$ und $\tau' = \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle$ mit $\{m'_1, \dots, m'_l\} \subseteq \{m_1, \dots, m_k\}$ 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.

„ \Leftarrow “ 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) $\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$, also $\tau \leq \tau'$.
- (c) Für die Formel $\langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle \leq \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle$ 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

$$\tau''_i = \begin{cases} \tau'_j & \text{falls } m_i = m'_j \\ \tau_i & \text{sonst.} \end{cases}$$

$$\begin{array}{c} \text{S-OBJ-DEPTH} \quad \frac{\tau_1 \leq \tau''_1 \quad \dots \quad \tau_k \leq \tau''_k}{\langle m_i : \tau_i^{i=1, \dots, k} \rangle \leq \langle m_i : \tau''_i^{i=1, \dots, k} \rangle} \quad \text{S-OBJ-WIDTH} \quad \frac{\langle m_i : \tau''_i^{i=1, \dots, k} \rangle \leq \langle m'_i : \tau'_i^{i=1, \dots, l} \rangle}{\langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle} \\ \text{S-TRANS} \quad \frac{\langle m_i : \tau_i^{i=1, \dots, k} \rangle \leq \langle m_i : \tau''_i^{i=1, \dots, k} \rangle \quad \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle}{\langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle \leq \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle} \end{array}$$

$\tau_i \leq \tau''_i$ 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.

„ \Rightarrow “ 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 $\tau'' = \tau''_1 \rightarrow \tau''_2$ muss $\tau = \tau_1 \rightarrow \tau_2$ und $\tau' = \tau'_1 \rightarrow \tau'_2$ 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 $\tau = \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$ verläuft analog.
- 3.) $\tau \leq \tau'$ mit Regel (S-ARROW), dann ist $\tau = \tau_1 \rightarrow \tau_2$ und $\tau' = \tau'_1 \rightarrow \tau'_2$ 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 $\tau = \langle m_1 : \tau_1; \dots; m_{n+k} : \tau_{n+k} \rangle$ und $\tau' = \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$ folgen, und es gilt $\{m_1, \dots, m_n\} \subseteq \{m_1, \dots, m_{n+k}\}$. Mit Regel (S-REFL) folgt darüberhinaus $\tau_i \leq \tau_i$ für $i = 1, \dots, n$. ■

- 5.) $\tau \leq \tau'$ mit (S-OBJ-DEPTH) kann nur mit $\tau = \langle m_1 : \tau_1; \dots; m_n : \tau_n \rangle$ und $\tau' = \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$ aus $\tau_i \leq \tau'_i$ für $i = 1, \dots, n$ folgen. Offensichtlich gilt $\{m_1, \dots, m_n\} \subseteq \{m_1, \dots, m_n\}$. \square

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 3.2 *Seien $\tau, \tau_1, \tau_2 \in \text{Type}$. Dann gilt:*

- (a) $\tau \leq \tau_1 \rightarrow \tau_2 \Rightarrow \exists \tau'_1, \tau'_2 \in \text{Type} : \tau = \tau'_1 \rightarrow \tau'_2 \wedge \tau_1 \leq \tau'_1 \wedge \tau'_2 \leq \tau_2$
- (b) $\tau_1 \rightarrow \tau_2 \leq \tau \Rightarrow \exists \tau'_1, \tau'_2 \in \text{Type} : \tau = \tau'_1 \rightarrow \tau'_2 \wedge \tau'_1 \leq \tau_1 \wedge \tau_2 \leq \tau'_2$

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. \square

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

Korollar 3.3 *Seien $\tau, \tau'_1, \dots, \tau'_l \in \text{Type}$ und $m'_1, \dots, m'_l \in \text{Method}$. Dann gilt:*

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

Beweis: Folgt ebenso unmittelbar aus Lemma 3.3. \square

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 *structural 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* verwendet². 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 vorangegangenen 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 (aus Definition 2.20) sowie der Subsumption-Regel

$$(\text{SUBSUME}) \quad \frac{\Gamma \triangleright_m e :: \tau \quad \tau \leq \tau'}{\Gamma \triangleright_m e :: \tau'}$$

herleiten lässt.

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

object (*self* : $\langle m : \text{int} \rangle$) **method** $m = 1$; **end**

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

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

$$\begin{array}{c}
\text{CONST} \frac{\text{INT} \quad 1 :: \mathbf{int}}{[self : \langle m : \mathbf{int}; \emptyset \rangle] \triangleright 1 :: \mathbf{int}} \quad \text{EMPTY} \frac{}{[self : \langle m : \mathbf{int}; \emptyset \rangle] \triangleright \epsilon :: \emptyset} \\
\text{METHOD} \frac{}{[self : \langle m : \mathbf{int}; \emptyset \rangle] \triangleright \mathbf{method} \ m = 1; \epsilon :: \langle m : \mathbf{int}; \emptyset \rangle} \\
\text{OBJECT} \frac{}{[] \triangleright \mathbf{object} \ (self : \langle m : \mathbf{int}; \emptyset \rangle) \ \mathbf{method} \ m = 1; \epsilon \ \mathbf{end} :: \langle m : \mathbf{int}; \emptyset \rangle}
\end{array}$$

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.

$$\begin{array}{c}
\text{OBJECT} \frac{\vdots}{[] \triangleright \mathbf{object} \ \dots \ \mathbf{end} :: \langle m : \mathbf{int}; \emptyset \rangle} \quad \text{S-OBJ-WIDTH} \frac{}{\langle m : \mathbf{int}; \emptyset \rangle \leq \langle \emptyset \rangle} \\
\text{SUBSUME} \frac{}{[] \triangleright \mathbf{object} \ (self : \langle m : \mathbf{int}; \emptyset \rangle) \ \mathbf{method} \ m = 1; \epsilon \ \mathbf{end} :: \langle \emptyset \rangle}
\end{array}$$

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

object (self : $\langle m : \mathbf{int} \rangle$) **method** $m = 1$; **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 \mathbf{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.

$$\text{ABSTR} \frac{\text{SEND} \frac{\text{ID} \frac{[o : \langle m : \mathbf{int} \rangle] \triangleright o :: \langle m : \mathbf{int} \rangle}{[o : \langle m : \mathbf{int} \rangle] \triangleright o \# m :: \mathbf{int}}}{[] \triangleright \lambda o : \langle m : \mathbf{int} \rangle . o \# m :: \langle m : \mathbf{int} \rangle \rightarrow \mathbf{int}}}{[] \triangleright \lambda o : \langle m : \mathbf{int} \rangle . o \# m :: \langle m : \mathbf{int} \rangle \rightarrow \mathbf{int}}$$

Betrachten wir weiterhin das Objekt

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

welches zwei Methoden von Typ \mathbf{int} bereitstellt. Mit den Typregeln von \mathcal{L}_o^t lässt sich hierfür offensichtlich der Typ $\langle m : \mathbf{int}; n : \mathbf{int} \rangle$ herleiten. Gemäß der Typeindeutigkeit 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 $\Gamma = [f_m : \langle m : \mathbf{int} \rangle \rightarrow \mathbf{int}, obj : \langle m : \mathbf{int}; n : \mathbf{int} \rangle]$ 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 .

$$\text{APP} \frac{\text{ID} \frac{\Gamma \triangleright f_m :: \langle m : \mathbf{int} \rangle \rightarrow \mathbf{int}}{\Gamma \triangleright f_m :: \langle m : \mathbf{int} \rangle \rightarrow \mathbf{int}} \quad \text{ID} \frac{\Gamma \triangleright obj :: \langle m : \mathbf{int}; n : \mathbf{int} \rangle}{\Gamma \triangleright obj :: \langle m : \mathbf{int}; n : \mathbf{int} \rangle}}{\Gamma \triangleright f_m \ obj ::}$$

Die Typregel (APP) fordert $\langle m : \mathbf{int} \rangle = \langle m : \mathbf{int}; n : \mathbf{int} \rangle$, 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:

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

- (a) den Typ von f_m anpassen zu $\langle m : \mathbf{int}; n : \mathbf{int} \rangle \rightarrow \mathbf{int}$,
- (b) oder den Typ von obj anpassen zu $\langle m : \mathbf{int} \rangle$.

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

$$\begin{array}{c}
\text{ID} \quad \Gamma \triangleright f_m :: \langle m : \mathbf{int} \rangle \rightarrow \mathbf{int} \quad \begin{array}{c} \text{S-OBJ-WIDTH} \\ \frac{\langle m : \mathbf{int}; n : \mathbf{int} \rangle \leq \langle m : \mathbf{int} \rangle}{\langle m : \mathbf{int} \rangle \rightarrow \mathbf{int} \leq \langle m : \mathbf{int}; n : \mathbf{int} \rangle \rightarrow \mathbf{int}} \quad \begin{array}{c} \text{S-REFL} \\ \mathbf{int} \leq \mathbf{int} \end{array} \\ \text{S-ARROW} \end{array} \\
\hline
\Gamma \triangleright f_m :: \langle m : \mathbf{int}; n : \mathbf{int} \rangle \rightarrow \mathbf{int} \quad \text{SUBSUME}
\end{array}$$

Nach dieser Typanpassung sind die Voraussetzungen für die Typregel (APP) erfüllt und für den Gesamtausdruck lässt sich somit der Typ \mathbf{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) $\forall \Gamma \in TEnv, e \in Exp, \tau \in Type : \Gamma \triangleright e :: \tau \Rightarrow \text{free}(e) \subseteq \text{dom}(\Gamma)$
- (b) $\forall \Gamma \in TEnv, r \in Row, \phi \in RType : \Gamma \triangleright r :: \phi \Rightarrow \text{free}(r) \subseteq \text{dom}(\Gamma)$

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 $\text{free}(e) \subseteq \text{dom}(\Gamma)$, was zu zeigen war.

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

TODO: Überleitung

Lemma 3.5 (Koinzidenzlemma für \mathcal{L}_o^{sub})

- (a) $\forall \Gamma_1, \Gamma_2 \in TEnv, e \in Exp, \tau \in Type : \Gamma_1 \triangleright e :: \tau \wedge \Gamma_1 =_{free(e)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright e :: \tau$
- (b) $\forall \Gamma_1, \Gamma_2 \in TEnv, r \in Row, \phi \in RType : \Gamma_1 \triangleright r :: \phi \wedge \Gamma_1 =_{free(r)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright r :: \phi$

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 $\Gamma_2 \triangleright e :: \tau$. \square

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) $\forall e' \in Exp : \forall \tau' \in Type : \Gamma[\tau/id] \triangleright e' :: \tau' \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright e'[e/id] :: \tau'$
- (b) $\forall r \in Row : \forall \phi \in RType : \Gamma[\tau/id] \triangleright r :: \phi \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright r[e/id] :: \phi$

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. \square

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) $\Gamma[\tau/self] \triangleright e :: \tau'$
 - (2) $\Gamma^* \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau$
 - (3) $\forall a \in dom(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright e[\mathbf{object}(self : \tau) r \mathbf{end} / self] :: \tau'$
- (b) Für alle $r' \in Row$ und $\phi \in RType$:

- $$\begin{aligned}
 (1) \quad & \Gamma[\tau/self] \triangleright r' :: \phi \\
 (2) \quad & \Gamma^* \triangleright \mathbf{object}(self : \tau) \ r \oplus r' \ \mathbf{end} :: \tau \\
 (3) \quad & \forall a \in \text{dom}(\Gamma) \cap \text{Attribute}, \tau_a \in \text{Type} : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a \\
 \Rightarrow \quad & \Gamma \triangleright r'[\mathbf{object}(self:\tau) \ r \oplus r' \ \mathbf{end} / self] :: \phi
 \end{aligned}$$

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

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)

- $$\begin{aligned}
 (a) \quad & \text{Wenn } \Gamma \triangleright op :: \tau, \text{ dann gilt } \tau = \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int} \text{ für } op \in \{+, -, *\} \text{ oder } \\
 & \tau = \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool} \text{ für } op \in \{\leq, \geq, <, >, =\}. \\
 (b) \quad & \text{Wenn } \Gamma \triangleright \lambda x : \tau_1. e :: \tau'_1 \rightarrow \tau_2, \text{ dann gilt } \Gamma[\tau_1/x] \triangleright e :: \tau_2 \text{ und } \tau'_1 \leq \tau_1. \\
 (c) \quad & \text{Wenn } \Gamma \triangleright e_1 e_2 :: \tau, \text{ dann gilt } \Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau, \Gamma \triangleright e_2 :: \tau_2 \text{ und } \tau_2 \leq \tau'_2. \\
 (d) \quad & \text{Wenn } \Gamma \triangleright \mathbf{object}(self : \tau) \ r \ \mathbf{end} :: \tau', \text{ dann gilt } \Gamma^*[\tau/self] \triangleright r :: \phi, \tau = \langle \phi \rangle \text{ und } \\
 & \tau \leq \tau'.
 \end{aligned}$$

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.

- $$\begin{aligned}
 (a) \quad & \Gamma \triangleright op :: \tau \text{ kann nur mit einer der Typregeln (CONST) oder (SUBSUME) hergeleitet} \\
 & \text{worden sein.} \\
 1.) \quad & \text{Im Fall von (CONST) gilt } op :: \tau, \text{ welches wiederum nur mit (AOP) oder} \\
 & \text{(ROP) hergeleitet worden sein kann. Für (AOP) gilt } op \in \{+, -, *\} \text{ und} \\
 & \tau = \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}, \text{ andernfalls muss gelten } op \in \{\leq, \geq, <, >, =\} \text{ und} \\
 & \tau = \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}.
 \end{aligned}$$

- 2.) Für $\Gamma \triangleright op :: \tau$ mit Typregel (SUBSUME) gilt $\Gamma \triangleright op :: \tau'$ und $\tau' \leq \tau$. Nach Induktionsvoraussetzung ist $\tau' = \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ und $op \in \{+, -, *\}$ oder $\tau' = \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ und $op \in \{\leq, \geq, <, >, =\}$.
Wir betrachten nur den Fall $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int} \leq \tau$, der zweite Fall folgt entsprechend. Gemäß Subtyping-Lemma (3.3) existieren $\tau_1, \tau_2 \in Type$ mit $\tau = \tau_1 \rightarrow \tau_2$, $\tau_1 \leq \mathbf{int}$ und $\mathbf{int} \rightarrow \mathbf{int} \leq \tau_2$. Nach erneuter Anwendung des Lemmas also $\tau_2 = \tau'_2 \rightarrow \tau''_2$, $\tau'_2 \leq \mathbf{int}$ und $\mathbf{int} \leq \tau''_2$. Mit Lemma 3.2 folgt schliesslich $\tau_1 = \tau'_2 = \tau''_2 = \mathbf{int}$, also $\tau = \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$, was zu zeigen war.
- (b) Zur Herleitung des Typurteils $\Gamma \triangleright \lambda x : \tau_1. e :: \tau'_1 \rightarrow \tau_2$ 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 $\Gamma \triangleright \lambda x : \tau_1. e :: \tau'_1 \rightarrow \tau_2$ mit (SUBSUME) existiert nach Voraussetzung ein $\tau'' \in Type$ mit $\Gamma \triangleright \lambda x : \tau_1. e :: \tau''$ und $\tau'' \leq \tau'_1 \rightarrow \tau_2$. Gemäß dem Subtyping-Lemma (3.3) existieren somit $\tau'_1, \tau''_2 \in Type$ mit $\tau'' = \tau'_1 \rightarrow \tau''_2$, $\tau'_1 \leq \tau'_1$ und $\tau''_2 \leq \tau_2$. Für $\Gamma \triangleright \lambda x : \tau_1. e :: \tau'_1 \rightarrow \tau''_2$ 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 $\tau''_2 \leq \tau_2$ 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 $\Gamma \triangleright e_1 :: \tau_2 \rightarrow \tau$ 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 $\tau_2, \tau'_2 \in Type$ mit $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau'$, $\Gamma \triangleright e_2 :: \tau_2$ und $\tau_2 \leq \tau'_2$. Schliesslich folgt wegen $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau'$ und $\tau' \leq \tau$ mit Typregel (SUBSUME) und den Subtyping-Regeln (S-ARROW) und (S-REFL) $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau$.
- (d) Das Typurteil $\Gamma \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau'$ kann nur mit einer der Typregeln (OBJECT) oder (SUBSUME) hergeleitet worden sein.
- 1.) Falls $\Gamma \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau'$ mit Typregel (OBJECT) hergeleitet wurde, existiert nach Voraussetzung ein $\phi \in RType$ mit $\Gamma^*[\tau/self] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle = \tau'$. Wegen (S-REFL) also $\tau \leq \tau'$.

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

TODO

Satz 3.1 (Typerhaltung, „Preservation“)

- (a) $\forall \Gamma \in TEnv : \forall e, e' \in Exp : \forall \tau \in Type : \Gamma^* \triangleright e :: \tau \wedge e \rightarrow e' \Rightarrow \Gamma^* \triangleright e' :: \tau$
- (b) $\forall \Gamma \in TEnv : \forall r, r' \in Row : \forall \phi \in RType : \Gamma \triangleright r :: \phi \wedge r \rightarrow r' \Rightarrow \Gamma \triangleright r' :: \phi$

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 $\Gamma^* \triangleright e :: \tau$ 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 $e \not\rightarrow$ bzw. $r \not\rightarrow$.
- 2.) $\Gamma^* \triangleright e :: \tau$ mit (SUBSUME) kann nur aus Prämissen der Form $\Gamma^* \triangleright e :: \tau'$ und $\tau' \leq \tau$ folgen. Nach Induktionsvoraussetzung gilt dann $\Gamma^* \triangleright e' :: \tau'$, und mit Typregel (SUBSUME) folgt schliesslich $\Gamma^* \triangleright e' :: \tau$.
- 3.) $\Gamma^* \triangleright e_1 e_2 :: \tau$ mit Typregel (APP) kann nur aus Prämissen der Form $\Gamma^* \triangleright e_1 :: \tau_2 \rightarrow \tau$ und $\Gamma^* \triangleright e_2 :: \tau_2$ folgen. Der small step $e_1 e_2 \rightarrow e'$ 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 $e_1 \rightarrow e'_1$. Nach Induktionsvoraussetzung gilt $\Gamma^* \triangleright e'_1 :: \tau_2 \rightarrow \tau$ und mit Typregel (APP) folgt daraus $\Gamma^* \triangleright e'_1 e_2 :: \tau$.

- 2.) $e_1 e_2 \rightarrow e'$ mit (APP-RIGHT) kann nur aus $e_2 \rightarrow e'_2$ und $e' = e_1 e'_2$ folgen. Mit Induktionsvoraussetzung folgt $\Gamma^* \triangleright e'_2 :: \tau_2$ und mit Typregel (APP) somit $\Gamma^* \triangleright e_1 e'_2 :: \tau$.
 - 3.) Für $e_1 e_2 \rightarrow e'$ 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 $\Gamma^* \triangleright op\ n_1\ n_2 :: \mathbf{int}$ und insbesondere $op :: \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ wegen (AOP). Also ist $op^I(n_1, n_2) \in Int$, und es gilt $\Gamma^* \triangleright op^I(n_1, n_2) :: \mathbf{int}$.
 Sei andererseits $op \in \{<, >, \leq, \geq, =\}$. Es gilt $\Gamma^* \triangleright op\ n_1\ n_2 :: \mathbf{bool}$ nach Voraussetzung und insbesondere $op :: \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ wegen (ROP). Dann muss also gelten $op^I(n_1, n_2) \in Bool$ und somit $\Gamma^* \triangleright op^I(n_1, n_2) :: \mathbf{bool}$.
 - 4.) Falls $e_1 e_2 \rightarrow e'$ mit (BETA-V) hergeleitet worden ist, so gilt $e_1 = \lambda x : \tau'_2. e'_1$, $e_2 \in Val$ und $e' = e'_1[e_2/x]$. Nach Lemma 3.8 gilt $\Gamma^*[\tau'_2/x] \triangleright e'_1 :: \tau$ und $\tau_2 \leq \tau'_2$. Aus $\Gamma^* \triangleright e_2 :: \tau_2$ und $\tau_2 \leq \tau'_2$ folgt mit Typregel (SUBSUME) $\Gamma^* \triangleright e_2 :: \tau'_2$, und nach Lemma 3.6 gilt dann $\Gamma^* \triangleright e'_1[e_2/x] :: \tau$ wegen $\Gamma^*[\tau'_2/x] \triangleright e'_1 :: \tau$ und $\Gamma^* \triangleright v_2 :: \tau'_2$.
- 4.) Für $\Gamma^* \triangleright e_1 \# m :: \tau$ mit Typregel (SEND) gilt nach Voraussetzung $\Gamma^* \triangleright e_1 :: \langle m : \tau; \emptyset \rangle$. Der small step $e_1 \# m \rightarrow e'$ kann entweder mit (SEND-EVAL) oder (SEND-UNFOLD) hergeleitet worden sein.
- 1.) Im Fall von (SEND-EVAL) existiert ein $e'_1 \in Exp$ mit $e' = e'_1 \# m$ und ein small step $e_1 \rightarrow e'_1$. Nach Induktionsvoraussetzung gilt $\Gamma^* \triangleright e'_1 :: \langle m : \tau; \emptyset \rangle$ und mit Typregel (SEND) folgt $\Gamma^* \triangleright e'_1 \# m :: \tau$.
 - 2.) $e_1 \# m \rightarrow e'$ mit (SEND-UNFOLD) andererseits bedingt $e_1 = \mathbf{object}(self : \tau') \ \omega \ \mathbf{end}$ sowie $e' = \omega[\mathbf{object}(self : \tau') \ \omega \ \mathbf{end} / self] \# m$, also gilt insbesondere $\Gamma^* \triangleright \mathbf{object}(self : \tau) \ \omega \ \mathbf{end} :: \langle m : \tau; \phi \rangle$.
 Nach Lemma 3.8 existiert ein $\phi' \in RType$ mit $(\Gamma^*)^*[\tau' / self] \triangleright \omega :: \phi'$, $\tau' = \langle \phi' \rangle$ und $\langle \phi' \rangle \leq \langle m : \tau; \phi \rangle$. Mit Typregel (OBJECT) folgt $\Gamma^* \triangleright \mathbf{object}(self : \tau') \ \omega \ \mathbf{end} :: \tau'$ wegen $(\Gamma^*)^*[\tau' / self] \triangleright \omega :: \phi'$ und $\tau' = \langle \phi' \rangle$.
 Aufgrund von $dom(\Gamma^*) \cap Attribute = \emptyset$ folgt somit wegen Lemma 3.7 $\Gamma^* \triangleright \omega[\mathbf{object}(self : \tau') \ \omega \ \mathbf{end} / self] :: \phi'$. Wegen $\langle \phi' \rangle \leq \langle m : \tau; \phi \rangle$ existieren nach Lemma 3.3 $\phi'' \in RType$ und $\tau'' \in Type$, so dass gilt $\phi' = (m : \tau''; \phi'')$ mit $\tau'' \leq \tau$. Also folgt mit Typregel (SEND') $\Gamma^* \triangleright \omega[\mathbf{object}(self : \tau') \ \omega \ \mathbf{end} / self] \# m :: \tau''$ und

wegen $\tau'' \leq \tau$ schliesslich $\Gamma^* \triangleright \omega[\mathbf{object}(self:\tau') \ \omega \ \mathbf{end}/_{self}] \# m :: \tau$ mit Typregel (SUBSUME).

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

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

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 $[] \triangleright v :: \tau_1 \rightarrow \tau_2$ 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 \mathbf{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 $v \in \mathit{Int}$.
 - 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 $v \in \mathit{Int}$.
- (b) Das Typurteil $[] \triangleright v :: \tau_1 \rightarrow \tau_2$ kann nur mit den Typregeln (APP), (ABSTR), (CONST) oder (SUBSUME) hergeleitet worden sein.
- 1.) $[] \triangleright v :: \tau_1 \rightarrow \tau_2$ mit Typregel (CONST) kann nur aus $v \in \mathit{Const}$ und $v :: \tau_1 \rightarrow \tau_2$ folgen. Letzteres wiederum kann aber nur mit (AOP) oder (ROP) hergeleitet worden sein, also gilt $v \in \mathit{Op}$.
 - 2.) Falls $[] \triangleright v :: \tau_1 \rightarrow \tau_2$ mit Typregel (ABSTR) hergeleitet worden ist, gilt unmittelbar nach Voraussetzung $v = \lambda x : \tau'_1. e$.
 - 3.) Für $[] \triangleright v :: \tau_1 \rightarrow \tau_2$ 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.) $[] \triangleright v :: \tau_1 \rightarrow \tau_2$ mit Typregel (SUBSUME) kann nur aus Prämissen der Form $[] \triangleright v :: \tau'$ und $\tau' \leq \tau_1 \rightarrow \tau_2$ folgen. Nach Lemma 3.3 existieren $\tau'_1, \tau'_2 \in \mathit{Type}$ mit $\tau' = \tau'_1 \rightarrow \tau'_2$, $\tau_1 \leq \tau'_1$ und $\tau'_2 \leq \tau_2$. Aus $[] \triangleright v :: \tau'_1 \rightarrow \tau'_2$ 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. \square

Satz 3.2 (Existenz des Übergangsschritts, „Progress“)

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

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

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 $e \in Val$, und für (EMPTY) gilt $r \in RVal$.
- 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 $[] \triangleright e_1 :: \tau_2 \rightarrow \tau$ und $[] \triangleright e_2 :: \tau_2$. Weiterhin ist nach der Art der Teilausdrücke e_1, e_2 zu unterscheiden.

Wenn $e_1 \notin Val$, dann existiert nach Induktionsvoraussetzung ein $e'_1 \in Exp$ mit $e_1 \rightarrow e'_1$. Also existiert für $e_1 e_2$ ein small step $e_1 e_2 \rightarrow e'_1 e_2$ mit Regel (APP-LEFT).

Entsprechendes gilt für den Fall $e_1 \in Val$ und $e_2 \notin Val$. Dann existiert ein small step mit Regel (APP-RIGHT).

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

- 1.) Wenn $e_1 \in Op$, dann gilt nach Definition 2.4 $e_1 e_2 \in Val$.
- 2.) Ist $e_1 = op v_1$ für $op \in Op$ und $v_1 \in Val$, so folgt aus $[] \triangleright op v_1 :: \tau_2 \rightarrow \tau$ mit Lemma 3.8 $[] \triangleright op :: \tau'_1 \rightarrow \tau_2 \rightarrow \tau$, $[] \triangleright v_1 :: \tau_1$ und $\tau_1 \leq \tau'_1$. Ebenfalls mit Lemma 3.8 folgt aus $[] \triangleright op :: \tau'_1 \rightarrow \tau_2 \rightarrow \tau$, 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 $v_1, e_2 \in Int$, also existiert ein small step $op v_1 e_2 \rightarrow op^I(v_1, e_2)$ mit Regel (OP).
- 3.) Für $e_1 = \lambda x : \tau'_2. e'_1$ existiert ein small step $(\lambda x : \tau'_2. e'_1) e_2 \rightarrow e'_1[e_2/x]$ mit Regel (BETA-V), da nach Voraussetzung $e_2 \in Val$.
- 4.) $[] \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau$ mit Typregel (OBJECT) kann ausschliesslich mit $[self : \tau] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen. Nach Induktionsvoraussetzung ist $r \in RVal$ oder es existiert ein $r' \in Row$ mit $r \rightarrow r'$. Falls also $r \in RVal$, so gilt $(\mathbf{object}(self : \tau) r \mathbf{end}) \in Val$. Anderenfalls, für $r \rightarrow r'$, existiert ein small step $\mathbf{object}(self : \tau) r \mathbf{end} \rightarrow \mathbf{object}(self : \tau) r' \mathbf{end}$ 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 $e_1 \in \text{Val}$ oder es existiert ein $e'_1 \in \text{Exp}$ mit $e_1 \rightarrow e'_1$.

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

Anderenfalls existiert ein small step $e_1 \# m \rightarrow e'_1 \# m$ 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 $Type \times Type$, die sich mit den folgenden Regeln herleiten lässt.

$$\begin{aligned}
 (\text{SM-REFL}) \quad & \tau_\beta \leq_m \tau_\beta \text{ für alle } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\
 (\text{SM-ARROW}) \quad & \frac{\tau'_1 \leq_m \tau_1 \quad \tau_2 \leq_m \tau'_2}{\tau_1 \rightarrow \tau_2 \leq_m \tau'_1 \rightarrow \tau'_2} \\
 (\text{SM-OBJECT}) \quad & \frac{\tau_i \leq_m \tau'_j \text{ für alle } i, j \text{ mit } m_i = m'_j}{\langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle \leq_m \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle} \\
 & \text{falls } \{m'_1, \dots, m'_l\} \subseteq \{m_1, \dots, m_k\}
 \end{aligned}$$

Intuitiv 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 $\forall \tau_1, \tau_2, \tau_3 \in Type : \tau_1 \leq_m \tau_2 \wedge \tau_2 \leq_m \tau_3 \Rightarrow \tau_1 \leq_m \tau_3$

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 $\tau_2 = \tau'_2 \rightarrow \tau''_2$ können $\tau_1 \leq_m \tau_2$ und $\tau_2 \leq_m \tau_3$ nur Regel (SM-ARROW) hergeleitet worden sein. Also ist $\tau_1 = \tau'_1 \rightarrow \tau''_1$ und $\tau_3 = \tau'_3 \rightarrow \tau''_3$, und es gilt $\tau'_2 \leq_m \tau'_1$, $\tau''_2 \leq_m \tau''_1$, $\tau'_3 \leq_m \tau'_2$ und $\tau''_3 \leq_m \tau''_2$. Nach Induktionsvoraussetzung folgt daraus $\tau'_3 \leq_m \tau'_1$ und $\tau''_3 \leq_m \tau''_1$, und mit Regel (SM-ARROW) schliesslich $\tau'_1 \rightarrow \tau''_1 \leq_m \tau'_3 \rightarrow \tau''_3$.
- 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. \square

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

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

Beweis: Die Äquivalenz zeigen wir in zwei Schritten.

„ \Rightarrow “ 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.

„ \Leftarrow “ 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 $\tau_1 = \tau'_1 \rightarrow \tau''_1$, 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 $\tau'_1 \rightarrow \tau''_1 \leq_m \tau'_1 \rightarrow \tau''_1$.
 Bleibt noch der Fall $\tau_1 = \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$ zu betrachten. Wegen (S-REFL) gilt wieder $\tau'_i \leq \tau'_i$ für $i = 1, \dots, n$ und nach Induktionsvoraussetzung gilt somit $\tau'_i \leq_m \tau'_i$ für $i = 1, \dots, n$. Mit Regel (SM-OBJECT) folgt daraus $\langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle \leq_m \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$, denn trivialerweise gilt $\{m_1, \dots, m_n\} \subseteq \{m_1, \dots, m_n\}$.
- 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 $\tau_1 \leq_m \tau_2$ 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 $\tau_1 = \langle m_1 : \tau'_1; \dots; m_{n+k} : \tau'_{n+k} \rangle$ und $\tau_2 = \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$. Es gilt also insbesondere $\{m_1, \dots, m_n\} \subseteq \{m_1, \dots, m_{n+k}\}$, und weiterhin

$\tau'_i \leq \tau'_i$ für $i = 1, \dots, n$ wegen (S-REFL). Daraus folgt mit Induktionsvoraussetzung $\tau'_i \leq_m \tau'_i$ für $i = 1, \dots, n$, und schliesslich mit Regel (SM-OBJECT) $\langle m_1 : \tau'_1; \dots; m_{n+k} : \tau'_{n+k} \rangle \leq_m \langle m_1 : \tau'_1; \dots; m_n : \tau'_n \rangle$.

- 5.) Für $\tau_1 \leq \tau_2$ mit Regel (S-OBJ-DEPTH) folgt die Behauptung ebenso problemlos mit (SM-OBJECT) aus der Induktionsvoraussetzung. \square

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 $\tau_1, \tau_2 \in \text{Type}$ überprüft, ob $\tau_1 \leq \tau_2$ gilt.

Algorithmus 3.1 (Entscheidungsalgorithmus für die Subtyprelation) Der Algorithmus erhält als Eingabe zwei Typen $\tau, \tau' \in \text{Type}$ 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 $\tau = \tau_1 \rightarrow \tau_2$ und $\tau' = \tau'_1 \rightarrow \tau'_2$, dann überprüfe rekursiv, ob $\tau'_1 \leq \tau_1$ und $\tau_2 \leq \tau'_2$ gelten.
- 3.) Wenn $\tau = \langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle$ und $\tau' = \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle$ ist, wobei $\{m'_1, \dots, m'_l\} \subseteq \{m_1, \dots, m_k\}$, 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 $\tau, \tau' \in \text{Type}$ ist wie folgt induktiv definiert:

$$\begin{aligned} \tau_\beta \vee \tau_\beta &= \tau_\beta \text{ für alle } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ \tau_1 \rightarrow \tau_2 \vee \tau'_1 \rightarrow \tau'_2 &= (\tau_1 \wedge \tau'_1) \rightarrow (\tau_2 \vee \tau'_2) \\ \langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle \\ \vee \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle &= \langle m''_1 : \tau''_1; \dots; m''_n : \tau''_n \rangle \\ &\text{mit } \{m''_1, \dots, m''_n\} = \{m_1, \dots, m_k\} \cap \{m'_1, \dots, m'_l\} \\ &\text{und } \tau''_h = \tau_i \vee \tau'_i \text{ für } m_i = m''_h = m'_j \end{aligned}$$

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

$$\begin{aligned} \tau_\beta \wedge \tau_\beta &= \tau_\beta \text{ für alle } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ \tau_1 \rightarrow \tau_2 \wedge \tau'_1 \rightarrow \tau'_2 &= (\tau_1 \vee \tau'_1) \rightarrow (\tau_2 \wedge \tau'_2) \\ \langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle \\ \wedge \langle m'_1 : \tau'_1; \dots; m'_l : \tau'_l \rangle &= \langle m''_1 : \tau''_1; \dots; m''_n : \tau''_n \rangle \\ &\text{mit } \{m''_1, \dots, m''_n\} = \{m_1, \dots, m_k\} \cup \{m'_1, \dots, m'_l\} \\ &\text{und } \tau''_h = \tau_i \wedge \tau'_i \text{ für } m_i = m''_h = m'_j \end{aligned}$$

TODO: Prosa

Lemma 3.11 Seien $\tau_1, \tau_2 \in \text{Type}$.

(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 $\tau, \tau_1, \tau_2 \in \text{Type}$. 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 Typeregeln 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 dass 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

$$\begin{array}{ll}
(\text{ID}) & \Gamma \triangleright_m id :: \tau \quad \text{falls } id \in \text{dom}(\Gamma) \wedge \Gamma(id) = \tau \\
(\text{CONST}) & \frac{c :: \tau}{\Gamma \triangleright_m c :: \tau} \\
(\text{APP-SUBSUME}) & \frac{\Gamma \triangleright_m e_1 :: \tau'_2 \rightarrow \tau \quad \Gamma \triangleright e_2 :: \tau_2 \quad \tau_2 \leq \tau'_2}{\Gamma \triangleright_m e_1 e_2 :: \tau} \\
(\text{ABSTR}) & \frac{\Gamma[\tau/x] \triangleright_m e :: \tau'}{\Gamma \triangleright_m \lambda x : \tau. e :: \tau \rightarrow \tau'} \\
(\text{REC-SUBSUME}) & \frac{\Gamma[\tau/x] \triangleright_m e :: \tau' \quad \tau' \leq \tau}{\Gamma \triangleright_m \mathbf{rec} x : \tau. e :: \tau} \\
(\text{LET}) & \frac{\Gamma \triangleright_m e_1 :: \tau_1 \quad \Gamma[\tau_1/x] \triangleright_m e_2 :: \tau_2}{\Gamma \triangleright_m \mathbf{let} x = e_1 \mathbf{in} e_2 :: \tau_2} \\
(\text{COND-SUBSUME}) & \frac{\Gamma \triangleright_m e_0 :: \mathbf{bool} \quad \Gamma \triangleright_m e_1 :: \tau_1 \quad \Gamma \triangleright_m e_2 :: \tau_2}{\Gamma \triangleright_m \mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 :: \tau_1 \vee \tau_2}
\end{array}$$

den folgenden Typregeln für Objekte

$$\begin{array}{lcl}
(\text{SEND}) & \frac{\Gamma \triangleright_m e :: \langle m : \tau; \phi \rangle}{\Gamma \triangleright_m e \# m :: \tau} & \\
(\text{OBJECT}) & \frac{\Gamma^*[\tau / \text{self}] \triangleright_m r :: \phi \quad \tau = \langle \phi \rangle}{\Gamma \triangleright_m \mathbf{object}(\text{self} : \tau) r \mathbf{end} :: \tau} & \\
(\text{DUPL-SUBSUME}) & \frac{\Gamma \triangleright_m \text{self} :: \tau \quad \forall i = 1 \dots n : \Gamma \triangleright_m a_i :: \tau_i \wedge \Gamma \triangleright_m e_i :: \tau'_i \wedge \tau'_i \leq \tau_i}{\Gamma \triangleright_m \{ \langle a_1 = e_1; \dots; a_n = e_n \rangle \} :: \tau} &
\end{array}$$

sowie den folgenden Typregeln für Reihen

$$\begin{array}{lcl}
(\text{EMPTY}) & \Gamma \triangleright_m \epsilon :: \emptyset & \\
(\text{ATTR}) & \frac{\Gamma^* \triangleright_m e :: \tau \quad \Gamma[\tau / a] \triangleright_m r_1 :: \phi}{\Gamma \triangleright_m \mathbf{val} a = e; r_1 :: \phi} & \\
(\text{METHOD-SUBSUME}) & \frac{\Gamma \triangleright_m \text{self} :: \langle m : \tau; \phi' \rangle \quad \Gamma \triangleright_m e :: \tau' \quad \Gamma \triangleright_m r_1 :: \phi \quad \tau' \leq \tau}{\Gamma \triangleright_m \mathbf{method} m = e; r_1 :: (m : \tau; \emptyset) \oplus \phi} &
\end{array}$$

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) $\forall \Gamma \in TEnv, e \in Exp, \tau \in Type : \Gamma \triangleright_m e :: \tau \Rightarrow \Gamma \triangleright e :: \tau$
- (b) $\forall \Gamma \in TEnv, r \in Row, \phi \in RType : \Gamma \triangleright_m r :: \phi \Rightarrow \Gamma \triangleright r :: \phi$

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 $\Gamma \triangleright_m e :: \tau$ und $\Gamma \triangleright_m r :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel (aus dem Minimal Typing Kalkül). Wie bereits erwähnt betrachten wir lediglich die interessanten Fälle.

- 1.) $\Gamma \triangleright_m e_1 e_2 :: \tau$ mit Typregel (APP-SUBSUME) kann nur aus Prämissen der Form $\Gamma \triangleright_m e_1 :: \tau'_2 \rightarrow \tau$, $\Gamma \triangleright_m e_2 :: \tau_2$ und $\tau_2 \leq \tau'_2$ folgen. Nach Induktionsvoraussetzung gilt also $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau$ und $\Gamma \triangleright e_2 :: \tau_2$, woraus sich das Typurteil $\Gamma \triangleright e_1 e_2 :: \tau$ wie folgt herleiten läßt:

$$\frac{\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau \quad \frac{\Gamma \triangleright e_2 :: \tau_2 \quad \tau_2 \leq \tau'_2}{\Gamma \triangleright e_2 :: \tau'_2} \text{SUBSUME}}{\Gamma \triangleright e_1 e_2 :: \tau} \text{APP}$$

- 2.) $\Gamma \triangleright_m \mathbf{rec} x : \tau.e :: \tau$ mit Typregel (REC-SUBSUME) bedingt $\Gamma[\tau/x] \triangleright_m e :: \tau'$ und $\tau' \leq \tau$. Mit Induktionsvoraussetzung folgt daraus $\Gamma[\tau/x] \triangleright e :: \tau'$, und die Behauptung läßt sich folgendermaßen herleiten:

$$\frac{\frac{\Gamma[\tau/x] \triangleright e :: \tau' \quad \tau' \leq \tau}{\Gamma[\tau/x] \triangleright e :: \tau} \text{SUBSUME}}{\Gamma \triangleright \mathbf{rec} x : \tau.e :: \tau} \text{REC}$$

- 3.) Für $\Gamma \triangleright_m \mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 :: \tau_1 \vee \tau_2$ gilt nach Voraussetzung $\Gamma \triangleright_m e_0 :: \mathbf{bool}$, $\Gamma \triangleright_m e_1 :: \tau_1$ und $\Gamma \triangleright_m e_2 :: \tau_2$. 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 $\Gamma \triangleright e_0 :: \mathbf{bool}$, $\Gamma \triangleright e_1 :: \tau_1$ und $\Gamma \triangleright e_2 :: \tau_2$. Damit läßt sich die Behauptung wie folgt herleiten:

$$\frac{\Gamma \triangleright e_0 :: \mathbf{bool} \quad \frac{\frac{\text{SUBSUME}}{\Gamma \triangleright e_1 :: \tau_1} \quad \tau_1 \leq \tau}{\Gamma \triangleright e_1 :: \tau} \quad \frac{\frac{\text{SUBSUME}}{\Gamma \triangleright e_2 :: \tau_2} \quad \tau_2 \leq \tau}{\Gamma \triangleright e_2 :: \tau}}{\Gamma \triangleright \mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 :: \tau} \text{COND}$$

- 4.) $\Gamma \triangleright_m \{\langle a_1 = e_1; \dots; a_n = e_n \rangle\} :: \tau$ mit Typregel (DUPL-SUBSUME) kann ausschliesslich aus Prämissen der Form

$$\Gamma \triangleright_m \mathbf{self} :: \tau$$

sowie

$$\forall i = 1, \dots, n : \Gamma \triangleright_m a_i :: \tau_i \wedge \Gamma \triangleright_m e_i :: \tau'_i \wedge \tau'_i \leq \tau_i$$

folgen. Nach Induktionsvoraussetzung gilt

$$\Gamma \triangleright self :: \tau$$

und

$$\forall i = 1, \dots, n : \Gamma \triangleright a_i :: \tau_i \wedge \Gamma \triangleright e_i :: \tau'_i \wedge \tau'_i \leq \tau_i,$$

und mit Typregel (SUBSUME) folgt

$$\forall i = 1, \dots, n : \Gamma \triangleright a_i :: \tau_i \wedge \Gamma \triangleright e_i :: \tau_i.$$

Daraus schliesslich folgt $\Gamma \triangleright \{\langle a_1 = e_1; \dots; a_n = e_n \rangle\} :: \tau$ mit Typregel (DUPL).

- 5.) Im Fall von $\Gamma \triangleright_m \mathbf{method} \ m = e_1; r_1 :: \phi$ mit Typregel (METHOD-SUBSUME) gilt nach Voraussetzung $\Gamma \triangleright_m self :: \langle m : \tau; \phi'' \rangle$, $\Gamma \triangleright_m e_1 :: \tau'$, $\Gamma \triangleright_m r_1 :: \phi'$ und $\tau' \leq \tau$, wobei $\phi = (m : \tau; \emptyset) \oplus \phi'$. Nach Induktionsvoraussetzung folgt somit $\Gamma \triangleright e_1 :: \tau'$ und $\Gamma \triangleright r_1 :: \phi'$. Wegen Typregel (SUBSUME) gilt ebenfalls $\Gamma \triangleright e_1 :: \tau$, und daraus schliesslich folgt mit (METHOD) $\Gamma \triangleright \mathbf{method} \ m = e_1; r_1 :: \phi$.

Die übrigen Fälle folgen trivialerweise aus der Übereinstimmung der Regeln (teilweise mit Induktionsvoraussetzung). \square

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

$$\begin{array}{l} \begin{array}{c} [] \triangleright_m e :: \tau \\ \xRightarrow{\text{Satz 3.5}} \\ [] \triangleright e :: \tau \end{array} \\ \xRightarrow{\text{Satz 3.3}} \text{die Berechnung von } e \text{ bleibt nicht stecken.} \end{array}$$

\square

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 $TEnv_\phi := \{\Gamma \in TEnv \mid \exists \phi', \phi'' \in RType : \Gamma(\text{self}) = \langle \phi' \rangle \wedge \phi' = \phi \oplus \phi''\}$

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 $\Gamma_1, \Gamma_2 \in TEnv$. Dann gilt:

- (a) $\Gamma_1 \leq \Gamma_2 \Rightarrow \Gamma_1^* \leq \Gamma_2^*$
- (b) $\forall \tau \in Type : \forall id \in Id : \Gamma_1 \leq \Gamma_2 \Rightarrow \Gamma_1[\tau/id] \leq \Gamma_2[\tau/id]$
- (c) $\forall \tau_1, \tau_2 \in Type : \forall id \in Id : \Gamma_1 \leq \Gamma_2 \wedge \tau_1 \leq \tau_2 \Rightarrow \Gamma_1[\tau_1/id] \leq \Gamma_2[\tau_2/id]$

Beweis:

- (a) Folgt trivialerweise, denn es gilt

$$dom(\Gamma_1^*) = dom(\Gamma_1) \cap Var = dom(\Gamma_2) \cap Var = dom(\Gamma_2^*)$$

und

$$\Gamma_1^*(id) = \Gamma_1(id) \leq \Gamma_2(id) = \Gamma_2^*(id)$$

für alle $id \in dom(\Gamma_1^*)$.

(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

$$\text{dom}(\Gamma_1[\tau_1/id]) = \text{dom}(\Gamma_1) \cup \{id\} = \text{dom}(\Gamma_2) \cup \{id\} = \text{dom}(\Gamma_2[\tau_2/id])$$

und

$$\Gamma_1[\tau_1/id](id) = \tau_1 \leq \tau_2 = \Gamma_2[\tau_2/id](id),$$

sowie

$$\forall id' \in \text{dom}(\Gamma_1) : \Gamma_1(id') \leq \Gamma_2(id'),$$

nach Voraussetzung, zusammenfassend also gilt $\Gamma_1[\tau_1/id] \leq \Gamma_2[\tau_2/id]$. \square

TODO: Prosa

Satz 3.7 (Vollständigkeit des Minimal Typing)

- (a) $\forall \tau \in \text{Type} : \forall \Gamma, \Gamma' \in \text{TEnv} : \forall e \in \text{Exp}_P :$
 $\Gamma \triangleright e :: \tau \wedge \Gamma' \leq \Gamma \wedge \Gamma' =_{\text{dom}(\Gamma) \cap \text{Attribute}} \Gamma \Rightarrow \exists \tau' \in \text{Type} : \Gamma' \triangleright_m e :: \tau' \wedge \tau' \leq \tau$
- (b) $\forall \phi \in \text{RType} : \forall \Gamma, \Gamma' \in \text{TEnv}_\phi : \forall r \in \text{Row} :$
 $\Gamma \triangleright r :: \phi \wedge \Gamma' \leq \Gamma \Rightarrow \Gamma' \triangleright_m r :: \phi$

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 $\Gamma \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel (der Programmiersprache $\mathcal{L}_o^{\text{sub}}$). Wir betrachten dazu exemplarisch die folgenden Fälle.

- 1.) $\Gamma \triangleright c :: \tau$ mit Typregel (CONST) kann nur mit $c :: \tau$ folgen. Also folgt $\Gamma' \triangleright_m c :: \tau$ mit Typregel (CONST) der Programmiersprache \mathcal{L}_o^m .
- 2.) Im Fall von $\Gamma \triangleright id :: \tau$ mit Typregel (ID) gilt nach Voraussetzung $id \in \text{dom}(\Gamma)$ und $\Gamma(id) = \tau$. Also gilt nach Definition 3.8 $id \in \text{dom}(\Gamma')$ und $\Gamma'(id) = \tau' \leq \tau$. Mit Typregel (ID) aus dem Minimal Typing Kalkül folgt somit $\Gamma' \triangleright_m id :: \tau'$.
- 3.) Für $\Gamma \triangleright e :: \tau$ mit Typregel (SUBSUME) existiert nach Voraussetzung ein $\tau' \in \text{Type}$ mit $\tau' \leq \tau$ und $\Gamma \triangleright e :: \tau'$. Nach Induktionsvoraussetzung existiert ein $\tau'' \leq \tau'$, so dass $\Gamma' \triangleright_m e :: \tau''$ gilt. Wegen (S-TRANS) schliesslich folgt $\tau'' \leq \tau$.
- 4.) $\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 :: \tau$ mit Typregel (LET) erfordert $\Gamma \triangleright e_1 :: \tau_1$ und $\Gamma[\tau_1/x] \triangleright e_2 :: \tau$. Nach Induktionsvoraussetzung existiert ein $\tau'_1 \in \text{Type}$ mit $\tau'_1 \leq \tau_1$, so dass

$$\Gamma' \triangleright_m e_1 :: \tau'_1$$

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

$$\Gamma'[\tau'_1/x] \triangleright_m e_2 :: \tau',$$

woraus dann mit Typregel (LET) von \mathcal{L}_o^m schliesslich

$$\Gamma' \triangleright_m \text{let } x = e_1 \text{ in } e_2 :: \tau'$$

folgt.

- 5.) Für $\Gamma \triangleright \lambda x : \tau_1. e :: \tau_1 \rightarrow \tau_2$ mit Typregel (ABSTR) muss gelten $\Gamma[\tau_1/x] \triangleright e :: \tau_2$ gelten. Nach Lemma 3.13 ist $\Gamma'[\tau_1/x] \leq \Gamma[\tau_1/x]$, also existiert nach Induktionsvoraussetzung ein $\tau'_2 \in \text{Type}$ mit $\tau'_2 \leq \tau_2$ und

$$\Gamma'[\tau_1/x] \triangleright_m e :: \tau'_2,$$

und mit Typregel (ABSTR) des Minimal Typing Kalküls folgt schliesslich

$$\Gamma' \triangleright_m \lambda x : \tau_1. e :: \tau_1 \rightarrow \tau'_2.$$

Es bleibt zu zeigen, dass $\tau_1 \rightarrow \tau'_2 \leq \tau_1 \rightarrow \tau_2$ gilt. Dies folgt einfach mit Subtyping-Regeln (S-ARROW), (S-REFL) und der Voraussetzung $\tau'_2 \leq \tau_2$.

- 6.) $\Gamma \triangleright \mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 :: \tau$ kann mit Typregel (COND) nur aus Prämissen der Form $\Gamma \triangleright e_0 :: \mathbf{bool}$, $\Gamma \triangleright e_1 :: \tau$ und $\Gamma \triangleright e_2 :: \tau$ folgen. Nach Induktionsvoraussetzung gilt also

$$\Gamma' \triangleright_m e_0 :: \mathbf{bool},$$

denn nach dem Subtyping-Lemma (Lemma 3.3) ist **bool** selbst der einzige Subtyp von **bool**. Weiterhin existiert nach Induktionsvoraussetzung ein $\tau_1 \in Type$ mit $\tau_1 \leq \tau$ und

$$\Gamma' \triangleright_m e_1 :: \tau_1,$$

sowie ein $\tau_2 \in Type$ mit $\tau_2 \leq \tau$ und

$$\Gamma' \triangleright_m e_2 :: \tau_2.$$

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

$$\Gamma' \triangleright_m \mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 :: \tau_1 \vee \tau_2$$

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 $\tau' \in Type$ 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 $\Gamma \triangleright \{a_1 = e_1; \dots; a_n = e_n\} :: \tau$ kann mit Typregel (DUPL) nur aus Prämissen der Form $\Gamma \triangleright self :: \tau$ sowie $\Gamma(a_i) = \tau_i$ und $\Gamma \triangleright e_i :: \tau_i$ für $i = 1, \dots, n$ folgen. Nach Induktionsvoraussetzung existiert also ein $\tau' \in Type$ mit $\tau' \leq \tau$ und

$$\Gamma' \triangleright_m self :: \tau'.$$

Darüberhinaus existieren $\tau'_1, \dots, \tau'_n \in \text{Type}$ mit $\tau'_i \leq \tau_i$ und

$$\Gamma' \triangleright_m e_i :: \tau'_i$$

für $i = 1, \dots, n$. Wegen $\Gamma' =_{\text{dom}(\Gamma) \cap \text{Attribute}} \Gamma$ gilt nach wie vor⁴

$$\Gamma'(a_i) = \tau_i$$

für $i = 1, \dots, n$ und somit folgt

$$\Gamma' \triangleright_m \{ \langle a_1 = e_1; \dots; a_n = e_n \rangle \} :: \tau'$$

mit Typregel (DUPL-SUBSUME) der Programmiersprache \mathcal{L}_o^m .

- 9.) $\Gamma \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \tau$ mit Typregel (OBJECT) kann nur aus Prämissen der Form $\Gamma^*[\tau/self] \triangleright r :: \phi$ und $\tau = \langle \phi \rangle$ folgen. Mit Lemma 3.13 folgt aus $\Gamma \leq \Gamma'$, dass gilt

$$\Gamma^*[\tau/self] \leq \Gamma'^*[\tau/self],$$

also folgt nach Induktionsvoraussetzung $\Gamma'^*[\tau/self] \triangleright_m r :: \phi$, denn

$$\Gamma'^*[\tau/self](self) = \tau \leq \langle \phi \rangle$$

gilt wegen (S-REFL) und somit ist $\Gamma'^*[\tau/self] \in \text{TE}nv_\phi$. Daraus schliesslich folgt

$$\Gamma' \triangleright_m \mathbf{object} (self : \tau) r \mathbf{end} :: \tau$$

mit der Typregel (OBJECT) aus dem Minimal Typing Kalkül, und $\tau \leq \tau$ gilt trivialerweise wegen Subtyping-Regel (S-REFL).

- 10.) Falls das Typurteil $\Gamma \triangleright e \# m :: \tau$ mit Typregel (SEND) hergeleitet wurde, muss gelten $\Gamma \triangleright e :: \langle m : \tau; \phi \rangle$. Nach Induktionsvoraussetzung existiert dann ein $\tau_m \in \text{Type}$ mit $\tau_m \leq \langle m : \tau; \phi \rangle$, so dass

$$\Gamma' \triangleright_m e :: \tau_m$$

gilt. Gemäß dem Subtyping-Lemma (3.3) existieren dann $\tau' \in \text{Type}, \phi' \in \text{RType}$ mit $\tau_m = \langle m : \tau'; \phi' \rangle$, $\tau' \leq \tau$ und $\langle \phi' \rangle \leq \langle \phi \rangle$, und es gilt

$$\Gamma' \triangleright_m e \# m :: \tau'$$

wegen Typregel (SEND) der Programmiersprache \mathcal{L}_o^m .

⁴An dieser Stelle wird deutlich, warum die Voraussetzung $\Gamma' =_{\text{dom}(\Gamma) \cap \text{Attribute}} \Gamma$ unbedingt notwendig ist.

- 11.) $\Gamma_\phi \triangleright \mathbf{method} \ m = e'; r' :: \phi$ mit Typregel (METHOD) bedingt $\Gamma_\phi \triangleright e' :: \tau'$ und $\Gamma_\phi \triangleright r' :: \phi'$, wobei $\phi = (m : \tau'; \emptyset) \oplus \phi'$. Nach Induktionsvoraussetzung existiert ein $\tau \in Type$ mit $\tau \leq \tau'$, so dass gilt

$$\Gamma'_\phi \triangleright_m e' :: \tau$$

und

$$\Gamma'_\phi \triangleright_m r' :: \phi',$$

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

$$\Gamma'_\phi \triangleright_m self :: \langle (m : \tau'; \emptyset) \oplus \phi' \oplus \phi'' \rangle.$$

Insgesamt folgt also

$$\Gamma'_\phi \triangleright_m \mathbf{method} \ m = e'; r' :: \phi$$

mit der Typregel (METHOD-SUBSUME) aus dem Minimal Typing Kalkül.

- 12.) Wenn $\Gamma_\phi \triangleright \mathbf{val} \ a = e'; r' :: \phi$ mit Typregel (ATTR) hergeleitet worden ist, gilt nach Voraussetzung $\Gamma_\phi^* \triangleright e' :: \tau$ und $\Gamma_\phi[\tau'/a] \triangleright r' :: \phi$. Nach Induktionsvoraussetzung existiert also ein $\tau' \in Type$ mit $\tau' \leq \tau$ und

$$\Gamma_\phi^* \triangleright_m e' :: \tau',$$

denn aus $\Gamma'_\phi \leq \Gamma_\phi$ folgt $\Gamma_\phi^* \leq \Gamma_\phi^*$ 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 $\Gamma'_\phi[\tau'/x] \in TEnv_\phi$. Zusammenfassend folgt also

$$\Gamma'_\phi \triangleright_m \mathbf{val} \ a = e'; r' :: \phi$$

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 3.4 *Sei $e \in \text{Exp}_P$ ein abgeschlossener Ausdruck. Dann gilt:*

$$\forall \tau \in \text{Type} : [] \triangleright e :: \tau \Rightarrow \exists \tau' \in \text{Type} : [] \triangleright_m e :: \tau' \wedge \tau' \leq \tau$$

Vereinfacht gesprochen gibt es zu jedem Typurteil der Programmiersprache $\mathcal{L}_o^{\text{sub}}$, ein äquivalentes in der Programmiersprache \mathcal{L}_o^m , 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 löst das Typsystem für die Programmiersprache \mathcal{L}_o^m , 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 $e \in \text{Exp}$ existiert höchstens ein Typ $\tau \in \text{Type}$ mit $\Gamma \triangleright_m e :: \tau$.*
- (b) *Für jede Typumgebung Γ und jede Reihe $r \in \text{Row}$ existiert höchstens ein Reihentyp $\phi \in \text{RType}$ 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 3.5 *Sei Γ eine Typumgebung.*

- (a) *Ist $e \in \text{Exp}$ wohlgetypt in Γ , dann existiert genau ein $\tau \in \text{Type}$ mit $\Gamma \triangleright_m e :: \tau$.*
- (b) *Ist $r \in \text{Row}$ wohlgetypt in Γ , dann existiert genau ein $\phi \in \text{RType}$ 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.

$$\text{(COERCE)} \quad \frac{\tau \leq \tau' \quad \Gamma \triangleright e :: \tau}{\Gamma \triangleright (e : \tau <: \tau') :: \tau'}$$

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 erweitern 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

$$\begin{aligned} & [] \triangleright e :: \tau \quad \text{im (COERCE)-Typsystem} \\ \Rightarrow & [] \triangleright e :: \tau \quad \text{im Typsystem von } \mathcal{L}_o^{sub} \\ \Rightarrow & e \text{ divergiert oder terminiert mit einem Wert (Satz 3.3).} \end{aligned}$$

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 $B \leq A$ 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 ist⁵.

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.

⁵Eine der vielen Stellen, an der sich das unliebsame C-Erbe bemerkbar macht.

4 Rekursive Typen

TODO: Diese Einleitung passt nicht zum Kapitel! Unbedingt anpassen!

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.

```
let point =  
  object (self)  
    val x = 1;  
    val y = 2;  
    method move =  $\lambda dx. \lambda dy. \{ \langle x = x + dx; y = y + dy \rangle \};$   
  end  
in point#move 2 1
```

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:

$$\tau_{point} = \langle move : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \tau_{point}; \emptyset \rangle$$

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

$$\begin{aligned} \tau^{raw} &::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit} \\ &\mid \tau_1^{raw} \rightarrow \tau_2^{raw} \\ &\mid \mu t. \tau_1^{raw} \\ &\mid \langle \phi^{raw} \rangle \end{aligned}$$

und die Menge $RType^{raw}$ aller syntaktisch herleitbaren Reihentypen ϕ^{raw} ist durch

$$\begin{aligned} \phi^{raw} &::= \emptyset \\ &\mid m : \tau^{raw}; \phi_1^{raw} \end{aligned}$$

definiert.

- (b) Die Menge $free(\tau^{raw})$ aller im Typ $\tau^{raw} \in Type^{raw}$ frei vorkommenden Typnamen ist induktiv durch

$$\begin{aligned} free(\tau_\beta) &= \emptyset \text{ für } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ free(\tau_1^{raw} \rightarrow \tau_2^{raw}) &= free(\tau_1^{raw}) \cup free(\tau_2^{raw}) \\ free(\mu t. \tau_1^{raw}) &= free(\tau_1^{raw}) \setminus \{t\} \\ free(\langle \phi^{raw} \rangle) &= free(\phi^{raw}) \end{aligned}$$

und die Menge $free(\phi^{raw})$ aller im Reihentyp $\phi^{raw} \in RType^{raw}$ frei vorkommenden Typnamen ist induktiv durch

$$\begin{aligned} free(\emptyset) &= \emptyset \\ free(m : \tau^{raw}; \phi_1^{raw}) &= free(\tau^{raw}) \cup free(\phi_1^{raw}) \end{aligned}$$

definiert.

- (c) Die Mengen $Type \subseteq Type^{raw}$ aller gültigen Typen τ und $RType^{raw} \subseteq RType$ aller gültigen Reihentypen ϕ der Programmiersprache \mathcal{L}_o^{rt} sind wie folgt definiert

$$\begin{aligned} Type &= \{\tau \in Type^{raw} \mid free(\tau) = \emptyset \wedge \forall t, t_1, \dots, t_n \in TName : \mu t. \mu t_1. \dots \mu t_n. t \notin \mathcal{P}(\tau)\} \\ RType &= \{\phi \in RType^{raw} \mid free(\phi) = \emptyset\}, \end{aligned}$$

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.

```
let point =
  object (self :  $\mu t. \langle move : \mathbf{int} \rightarrow \mathbf{int} \rightarrow t; \emptyset \rangle$ )
    val x = 1;
    val y = 2;
    method move =  $\lambda dx : \mathbf{int}. \lambda dy : \mathbf{int}. \{ \langle x = x + dx; y = y + dy \rangle \}$ ;
  end
in point # move 2 1
```

Intuitiv sollte sich für diesen Ausdruck der Typ des Punktobjekts herleiten lassen, also

$$\mu t. \langle move : \mathbf{int} \rightarrow \mathbf{int} \rightarrow t; \emptyset \rangle,$$

was dem zu erwartenden Ergebnis entsprechen würde.

Definition 4.2 Die Menge $TCons$ der Typkonstruktoren sei definiert durch

$$TCons = \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}, \rightarrow\} \cup \{\langle m_1; \dots; m_n \rangle \mid n \in \mathbb{N}, m_1, \dots, m_n \in Method\},$$

wobei die Methodennamen in $\langle m_1; \dots; m_n \rangle$ ähnlich wie bei den Typen als disjunkt angenommen werden.

TODO: Prosa

Definition 4.3

(a) Sei $root : Type \rightarrow TCons$ die induktiv wie folgt definierte Funktion:

$$\begin{aligned} root(\tau_\beta) &= \tau_\beta \quad \text{für } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ root(\tau_1 \rightarrow \tau_2) &= \rightarrow \\ root(\langle m_i : \tau_i^{i=1\dots n} \rangle) &= \langle m_i^{i=1\dots n} \rangle \\ root(\mu t. \tau) &= root(\tau[\mu t. \tau / t]) \end{aligned}$$

(b) Die Funktion $arity : Type \rightarrow \mathbb{N}$ sei induktiv definiert durch:

$$\begin{aligned} arity(\tau_\beta) &= 0 \quad \text{für } \tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \\ arity(\tau_1 \rightarrow \tau_2) &= 2 \\ arity(\langle m_i : \tau_i^{i=1\dots n} \rangle) &= n \\ arity(\mu t. \tau) &= arity(\tau[\mu t. \tau / t]) \end{aligned}$$

(c) Die partielle Funktion $child : Type \times \mathbb{N} \rightarrow Type$ ist induktiv definiert durch:

$$\begin{aligned} child(\tau_1 \rightarrow \tau_2, i) &= \begin{cases} \tau_i & \text{falls } 1 \leq i \leq 2 \\ \uparrow & \text{sonst} \end{cases} \\ child(\langle m_j : \tau_j^{j=1\dots n} \rangle, i) &= \begin{cases} \tau_i & \text{falls } 1 \leq i \leq n \\ \uparrow & \text{sonst} \end{cases} \\ child(\mu t. \tau, i) &= child(\tau[\mu t. \tau / t], i) \end{aligned}$$

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 $1 \leq i \leq arity(\tau)$.

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

$$\begin{aligned} \sim_0 &= Type^2 \\ \sim_{n+1} &= \{(\tau, \tau') \in Type^2 \mid root(\tau) = root(\tau') \\ &\quad \wedge \forall 1 \leq i \leq arity(\tau) : child_i(\tau) \sim_n child_i(\tau')\}, \end{aligned}$$

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 $\sim_0 = \text{Type}^2$ 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.

Transitivität: Seien $(\tau_1, \tau_2), (\tau_2, \tau_3) \in \sim_{n+1}$. Dann gilt $\text{root}(\tau_1) = \text{root}(\tau_2) = \text{root}(\tau_3)$, $\text{child}_i(\tau_1) \sim_n \text{child}_i(\tau_2)$ und $\text{child}_i(\tau_2) \sim_n \text{child}_i(\tau_3)$ für alle $i = 1, \dots, \text{arity}(\tau_2)$. Da \sim_n transitiv ist, folgt $\text{child}_i(\tau_1) \sim_n \text{child}_i(\tau_3)$ für alle $i = 1, \dots, \text{arity}(\tau_1)$, und somit $(\tau_1, \tau_3) \in \sim_{n+1}$.

Symmetrie: Sei $(\tau, \tau') \in \sim_{n+1}$. Dann ist nach Definition $\text{root}(\tau) = \text{root}(\tau')$ und es gilt $\text{child}_i(\tau) \sim_n \text{child}_i(\tau')$ für alle $i = 1, \dots, \text{arity}(\tau)$. Da \sim_n symmetrisch ist, folgt $\text{child}_i(\tau') \sim_n \text{child}_i(\tau)$, also $(\tau, \tau') \in \sim_{n+1}$.

Also ist \sim eine Äquivalenzrelation auf Type . □

Definition 4.5 Mit TypeRel bezeichnen wir die Menge der binären Relationen auf Type , also $\text{TypeRel} = \mathcal{P}(\text{Type} \times \text{Type})$.

Definition 4.6 Sei $\mathcal{E} : \text{TypeRel} \rightarrow \text{TypeRel}$ definiert durch

$$\begin{aligned} \mathcal{E}(R) &= \{(\tau, \tau') \in \text{Type}^2 \mid \text{root}(\tau) = \text{root}(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}\} \\ &\cup \{(\tau, \tau') \in \text{Type}^2 \mid \text{root}(\tau) = \text{root}(\tau') = \rightarrow \\ &\quad \wedge \forall i = 1, 2 : (\text{child}_i(\tau), \text{child}_i(\tau')) \in R\} \\ &\cup \{(\tau, \tau') \in \text{Type}^2 \mid \text{root}(\tau) = \text{root}(\tau') = \langle m_1; \dots; m_k \rangle \\ &\quad \wedge \forall i = 1, \dots, k : (\text{child}_i(\tau), \text{child}_i(\tau')) \in R\}, \end{aligned}$$

also eine totale Abbildung zwischen binären Typrelationen.

Die Idee ist nun zu zeigen, dass die Relation \sim ein Fixpunkt der generierenden Funktion \mathcal{E} ist. Dazu benutzen wir den Fixpunktsatz von Kleene, das heisst, wir müssen zunächst zeigen, dass $TypeRel$ eine dcpo ist, und die Abbildung \mathcal{E} auf $(TypeRel, \supseteq)$ stetig ist. Der Leser sei an dieser Stelle nochmals auf Abschnitt 1.1.1 verwiesen, in dem die notwendigen mathematischen Grundlagen zur Bereichstheorie dargestellt werden.

Lemma 4.2

- (a) $(TypeRel, \supseteq)$ ist eine dcpo mit kleinstem Element $Type \times Type$.
- (b) \mathcal{E} ist eine stetige Abbildung auf $(TypeRel, \supseteq)$.

Beweis:

- (a) Folgt unmittelbar nach Lemma 1.1.
- (b) Zunächst zeigen wir, dass \mathcal{E} eine monotone Abbildung auf $(TypeRel, \supseteq)$ ist. Seien dazu $R_1, R_2 \in TypeRel$ mit $R_1 \supseteq R_2$. Dann müssen wir zeigen, dass für alle $(\tau, \tau') \in \mathcal{E}(R_2)$ auch $(\tau, \tau') \in \mathcal{E}(R_1)$ gilt. Dazu unterscheiden wir nach der Form von $root(\tau)$:
 - 1.) $root(\tau) \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$
 Dann gilt $root(\tau) = root(\tau')$, und nach Definition 4.6 gilt $(\tau, \tau') \in \mathcal{E}(R)$ für alle $R \in TypeRel$, also insbesondere $(\tau, \tau') \in \mathcal{E}(R_1)$.
 - 2.) $root(\tau) = \rightarrow$
 Dann gilt $root(\tau) = root(\tau')$ und $(child_i(\tau), child_i(\tau')) \in R_2$ für $i = 1, 2$. Da $R_2 \subseteq R_1$ folgt unmittelbar $(child_i(\tau), child_i(\tau')) \in R_1$ und somit nach Definition 4.6 auch $(\tau, \tau') \in \mathcal{E}(R_1)$.
 - 3.) $root(\tau) = \langle m_1; \dots; m_k \rangle$
 Folgt völlig analog.

Um die Stetigkeit von \mathcal{E} zu beweisen, bleibt also zu zeigen, dass

$$\mathcal{E}(\bigsqcup \Delta) \supseteq \bigsqcup \mathcal{E}(\Delta)$$

für jede gerichtete Teilmenge $\Delta \subseteq TypeRel$ gilt, wobei das Supremum bezüglich \supseteq offensichtlich dem Durchschnitt entspricht. Sei dazu $\Delta \subseteq TypeRel$ gerichtet und $(\tau, \tau') \in \bigcap \mathcal{E}(\Delta) = \bigcap \{\mathcal{E}(R) \mid R \in \Delta\}$, das heisst $(\tau, \tau') \in \mathcal{E}(R)$ für alle $R \in \Delta$. Wir zeigen durch Fallunterscheidung nach der Form von $root(\tau)$, dass auch $(\tau, \tau') \in \mathcal{E}(\bigcap \Delta)$ gilt:

- 1.) $root(\tau) \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

Klar.

- 2.) $root(\tau) = \rightarrow$

Es gilt wieder $root(\tau) = root(\tau')$ und $(child_i(\tau), child_i(\tau')) \in R$ für $i = 1, 2$ und alle $R \in \Delta$. Also gilt insbesondere $(child_i(\tau), child_i(\tau')) \in \bigcap \Delta$ für $i = 1, 2$, und wegen Definition 4.6 folgt somit $(\tau, \tau') \in \mathcal{E}(\bigcap \Delta)$.

- 3.) $root(\tau) = \langle m_1; \dots; m_k \rangle$

Folgt wieder analog zum vorangegangenen Fall. \square

Damit nun lässt sich das zentrale Lemma über die Typrelation \sim formulieren und beweisen, welches uns einen einfacheren Umgang mit der Relation \sim ermöglicht, so dass wir nicht länger zeigen müssen, dass eine bestimmte Eigenschaft, die wir für \sim beweisen möchten, für alle \sim_n gilt.

Lemma 4.3 (Fixpunkt-Lemma für \mathcal{L}_o^{rt}) $\mu\mathcal{E} = \sim$

Mit anderen Worten, die Relation \sim ist der kleinste Fixpunkt der generierenden Funktion \mathcal{E} . Dies lässt sich mit dem Fixpunktsatz von Kleene leicht zeigen.

Beweis: Gemäß Lemma 4.2 ist $(TypeRel, \supseteq)$ eine dcpo mit einem kleinsten Element $Type \times Type$, und \mathcal{E} ist eine totale, stetige Abbildung auf $(TypeRel, \supseteq)$. Nach Satz 1.1 (Fixpunktsatz von Kleene) existiert also mit $\mu\mathcal{E} = \bigsqcup_{n \in \mathbb{N}} \mathcal{E}^n(Type \times Type)$ der kleinste Fixpunkt von \mathcal{E} .

Es bleibt noch zu zeigen, dass gerade die Typrelation \sim genau diesem Fixpunkt entspricht, also, dass $\sim = \bigsqcup_{n \in \mathbb{N}} \mathcal{E}^n(Type \times Type)$ gilt. Dazu zeigen wir zunächst durch vollständige Induktion über n , dass $\sim_n = \mathcal{E}^n(Type \times Type)$ gilt:

- $n = 0$

Klar, da $\sim_0 = Type \times Type = \mathcal{E}^0(Type \times Type)$.

- $n \rightsquigarrow n + 1$

Wegen Induktionsvoraussetzung gilt $\sim_n = \mathcal{E}^n(Type \times Type)$, und wir müssen zeigen, dass dann auch $\sim_{n+1} = \mathcal{E}(\sim_n)$ gilt:

„ \subseteq “ Sei $(\tau, \tau') \in \sim_{n+1}$, d.h. $root(\tau) = root(\tau')$ und $(child_i(\tau), child_i(\tau')) \in \sim_n$ für $i = 1, \dots, arity(\tau)$. Mit Fallunterscheidung nach der Form von $root(\tau)$ lässt sich leicht zeigen, dass auch $(\tau, \tau') \in \mathcal{E}(\sim_n)$ gilt.

„ \supseteq “ Sei nun $(\tau, \tau') \in \mathcal{E}(\sim_n)$. Dann ist je nach Form von $root(\tau)$ zu zeigen, dass auch $(\tau, \tau') \in \sim_{n+1}$ gilt, was aus offensichtlichen Gründen ziemlich trivial ist.

Insgesamt folgt dann $\sim = \bigcap_{n \in \mathbb{N}} \mathcal{E}^n(\text{Type} \times \text{Type}) = \bigsqcup_{n \in \mathbb{N}} \mathcal{E}^n(\text{Type} \times \text{Type})$, was es zu zeigen galt. \square

Korollar 4.2 \sim ist bezüglich \subseteq der größte Fixpunkt von \mathcal{E} .

Beweis: Klar. \square

Die Aussage des Fixpunkt-Lemmas lässt sich auch in einer einfacher anzuwendenden Form darstellen, mit der wir dann in den nachfolgenden Beweisen leichter argumentieren können:

Lemma 4.4 (Äquivalenz-Lemma für \mathcal{L}_o^{rt}) $\tau \sim \hat{\tau}$ gilt genau dann, wenn eine der folgenden Aussagen erfüllt ist:

- (a) $\tau = \mu t_1 \dots \mu t_n. \tau_\beta$ und $\hat{\tau} = \mu \hat{t}_1 \dots \mu \hat{t}_m. \tau_\beta$ mit $\tau_\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$
- (b) $\tau = \mu t_1 \dots \mu t_n. \tau_1 \rightarrow \tau_2$ und $\hat{\tau} = \mu \hat{t}_1 \dots \mu \hat{t}_m. \hat{\tau}_1 \rightarrow \hat{\tau}_2$ mit $\tau_1[\tau/t_i]_{i=1}^n \sim \hat{\tau}_1[\hat{\tau}/\hat{t}_i]_{i=1}^m$
und $\tau_2[\tau/t_i]_{i=1}^n \sim \hat{\tau}_2[\hat{\tau}/\hat{t}_i]_{i=1}^m$
- (c) $\tau = \mu t_1 \dots \mu t_n. \langle m_1 : \tau_1; \dots; m_k : \tau_k \rangle$ und $\hat{\tau} = \mu \hat{t}_1 \dots \mu \hat{t}_m. \langle m_1 : \hat{\tau}_1; \dots; m_k : \hat{\tau}_k \rangle$
mit $\tau_j[\tau/t_i]_{i=1}^n \sim \hat{\tau}_j[\hat{\tau}/\hat{t}_i]_{i=1}^m$ für alle $j = 1, \dots, k$

Dabei handelt es sich zum Beispiel bei τ_1, τ_2 um nicht unbedingt abgeschlossene Typen, in denen die Typnamen t_1, \dots, t_n frei vorkommen dürfen, was aber in der Folge bedeutet, dass in jedem Fall $\tau_1[\tau/t_i]_{i=1}^n, \tau_2[\tau/t_i]_{i=1}^n \in \text{Type}$ gilt, da τ ein abgeschlossener Typ ist. Entsprechendes gilt für die übrigen Typen $\hat{\tau}_1, \hat{\tau}_2, \dots$

Beweis: Folgt unmittelbar aus dem Fixpunkt-Lemma (4.3), da die Aussagen des Lemmas mit der Definition von \mathcal{E} übereinstimmen (4.6). \square

Um nun zum Beispiel zu zeigen, dass $\tau_1 \rightarrow \tau_2 \sim \tau'_1 \rightarrow \tau'_2$ gilt, genügt es zu zeigen, dass $\tau_1 \sim \tau'_1$ und $\tau_2 \sim \tau'_2$ gilt. Entsprechend gilt für Basistypen $\tau, \tau' \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$, dass $\tau \sim \tau'$ genau dann erfüllt ist, wenn $\tau = \tau'$ gilt.

Definition 4.7 (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

$$\begin{array}{l}
\text{(OBJECT')} \quad \frac{\Gamma^*[\tau/self] \triangleright_r r :: \phi \quad \tau \sim \langle \phi \rangle}{\Gamma \triangleright_e \mathbf{object}(self : \tau) r \mathbf{end} :: \tau} \\
\text{(EQUIV)} \quad \frac{\Gamma \triangleright_e e :: \tau' \quad \tau' \sim \tau}{\Gamma \triangleright_e e :: \tau}
\end{array}$$

herleiten lässt.

TODO: Beispiel für Typherleitung

TODO: Überleitung

4.1.1 Typsicherheit

TODO: Zunächst die obligatorischen Lemmata

Lemma 4.5 (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: 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. \square

Lemma 4.6 (Koinzidenzlemma für Lort)

- (a) $\forall \Gamma_1, \Gamma_2 \in TEnv, e \in Exp, \tau \in Type : \Gamma_1 \triangleright e :: \tau \wedge \Gamma_1 =_{free(e)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright e :: \tau$
- (b) $\forall \Gamma_1, \Gamma_2 \in TEnv, r \in Row, \phi \in RType : \Gamma_1 \triangleright r :: \phi \wedge \Gamma_1 =_{free(r)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright r :: \phi$

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. \square

Lemma 4.7 (Typurteile und Substitution) Sei $id \in Attribute \cup Var$, $\Gamma \in TEnv$, $\tau \in Type$ und $e \in Exp$. Dann gilt:

- (a) $\forall e' \in Exp : \forall \tau' \in Type : \Gamma[\tau/id] \triangleright e' :: \tau' \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright e'[e/id] :: \tau'$
- (b) $\forall r \in Row : \forall \phi \in RType : \Gamma[\tau/id] \triangleright r :: \phi \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright r[e/id] :: \phi$

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 $\tau'' \in Type$ 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.) $\Gamma[\tau/id] \triangleright \mathbf{object} (self : \tau') r \mathbf{end} :: \tau'$ mit Typregel (OBJECT').

Dann existiert ein $\phi \in RType$ mit $(\Gamma[\tau/id])^*[\tau'/self] \triangleright r :: \phi$ und $\langle \phi \rangle \sim \tau'$.

Falls $id \in Var$, dann gilt nach Definition $(\Gamma[\tau/id])^* = \Gamma^*[\tau/id]$. Andererseits gilt für $id \in Attribute$, dass $(\Gamma[\tau/id])^* = \Gamma^*$, also nach Lemma 4.5 $id \notin free(r)$. Demzufolge gilt $\Gamma^* =_{free(r)} \Gamma^*[\tau/id]$. Zusammenfassend folgt also

$$\Gamma^*[\tau/id][\tau'/self] \triangleright r :: \phi,$$

und da $self \notin Attribute \cup Var$ folgt weiter

$$\Gamma^*[\tau'/self][\tau/id] \triangleright r :: \phi.$$

Darauf lässt sich nun die Induktionsvoraussetzung anwendung und wir erhalten

$$\Gamma^*[\tau'/self] \triangleright r[e/id] :: \phi,$$

woraus sich wegen $\langle \phi \rangle \sim \tau'$ mit Typregel (OBJECT') schliesslich

$$\Gamma \triangleright \mathbf{object} (self : \tau') r[e/id] \mathbf{end} :: \tau'$$

ergibt, was nach Definition der Substitution (2.8) dem Typurteil

$$\Gamma \triangleright (\mathbf{object} (self : \tau') r \mathbf{end})[e/id] :: \tau'$$

entspricht.

Die übrigen Fälle verlaufen entsprechend wie im Beweis des Lemmas für die Programmiersprache \mathcal{L}_o^t . \square

Zusätzlich zum Substitutionslemma für Attribute und Variablen (4.7) benötigen wir wieder ein spezielles Lemma, welches die Typerhaltung der *self*-Substitution sichert (siehe Lemma 2.9).

Lemma 4.8 (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) $\Gamma^* \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau$
- (3) $\forall a \in \text{dom}(\Gamma) \cap \text{Attribute}, \tau_a \in Type : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright e[\mathbf{object}(self:\tau) r \mathbf{end}/self] :: \tau'$

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

- (1) $\Gamma[\tau/self] \triangleright r' :: \phi$
- (2) $\Gamma^* \triangleright \mathbf{object}(self : \tau) r \oplus r' \mathbf{end} :: \tau$
- (3) $\forall a \in \text{dom}(\Gamma) \cap \text{Attribute}, \tau_a \in Type : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright r'[\mathbf{object}(self:\tau) r \oplus r' \mathbf{end}/self] :: \phi$

Beweis: Der Beweis entspricht im wesentlichen dem Beweis von Lemma 2.9 für die Programmiersprache \mathcal{L}_o^t . Wie zuvor folgt gemäß Lemma 4.5 (Typumgebungen und frei vorkommende Namen), dass die Substitution stets definiert ist. Der Beweis erfolgt dann durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[\tau/self] \triangleright e :: \tau'$ und $\Gamma[\tau/self] \triangleright r' :: \phi$, und Fallunterscheidung nach der letzten Typregel in der Herleitung. Wir betrachten hier lediglich die Fälle der (EQUIV)- und (OBJECT')-Regeln.

1.) $\Gamma[\tau/self] \triangleright e :: \tau'$ mit Typregel (EQUIV).

Nach Voraussetzung existiert ein $\tau'' \in Type$ mit $\Gamma[\tau/self] \triangleright e :: \tau''$ und $\tau'' \sim \tau'$. Darauf lässt sich direkt die Induktionsvoraussetzung anwenden, und es folgt

$$\Gamma \triangleright e[\mathbf{object}(self:\tau) r \mathbf{end}/self] :: \tau''.$$

Wegen $\tau'' \sim \tau'$ folgt unmittelbar

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

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

$$(\mathbf{object} (self : \tau') r'' \mathbf{end})[\mathbf{object} (self : \tau') r'' \mathbf{end} / self] = \mathbf{object} (self : \tau') r'' \mathbf{end},$$

und es gilt $self \notin \text{free}(\mathbf{object} (self : \tau') r'' \mathbf{end})$ gemäß Definition 2.5. Mit Lemma 4.6 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 für die Programmiersprache \mathcal{L}_o^t . \square

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 4.9 (Umkehrung der Typrelation)

- (a) Wenn $\Gamma \triangleright op :: \tau$, dann gilt $\tau \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ für $op \in \{+, -, *\}$ oder $\tau \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ 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 $\tau' \sim \tau_2 \rightarrow \tau$.
- (c) Wenn $\Gamma \triangleright e_1 e_2 :: \tau$, dann gilt $\Gamma \triangleright e_1 :: \tau_2 \rightarrow \tau$ und $\Gamma \triangleright e_2 :: \tau_2$.
- (d) Wenn $\Gamma \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \tau'$, dann gilt $\Gamma^*[\tau/self] \triangleright r :: \phi$ 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 aus der Definition, da die Relation \sim reflexiv ist.
 - 2.) Wenn $\Gamma \triangleright op :: \tau$ mit Typregel (EQUIV) hergeleitet wurde, existiert nach Voraussetzung ein $\tau' \in Type$ 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 $\tau' = \tau_2 \rightarrow \tau$. Also $\tau' \sim \tau_2 \rightarrow \tau$, da \sim reflexiv ist.
 - 2.) Im Fall von (EQUIV) existiert wieder ein $\tau'' \in Type$, 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 $\tau'' \sim \tau_2 \rightarrow \tau$. Wegen der Transitivität 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 $\tau' \in Type$ mit $\Gamma \triangleright e_1 e_2 :: \tau'$ und $\tau' \sim \tau$. Nach Induktionsvoraussetzung existiert dann ein $\tau_2 \in Type$ mit $\Gamma \triangleright e_1 :: \tau_2 \rightarrow \tau'$ und $\Gamma \triangleright e_2 :: \tau_2$. Wegen $\tau' \sim \tau$ gilt nach Lemma 4.4 (Äquivalenz-Lemma) ebenfalls $\tau_2 \rightarrow \tau' \sim \tau_2 \rightarrow \tau$, und mit Anwendung der (EQUIV)-Regel folgt schliesslich $\Gamma \triangleright e_1 :: \tau_2 \rightarrow \tau$.
- (d) Das Typurteil $\Gamma \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau'$ 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. \square

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“)

$$(a) \forall \Gamma \in TEnv : \forall e, e' \in Exp : \forall \tau \in Type : \Gamma^* \triangleright e :: \tau \wedge e \rightarrow e' \Rightarrow \Gamma^* \triangleright e' :: \tau$$

$$(b) \forall \Gamma \in TEnv : \forall r, r' \in Row : \forall \phi \in RType : \Gamma \triangleright r :: \phi \wedge r \rightarrow r' \Rightarrow \Gamma \triangleright r' :: \phi$$

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 $\Gamma^* \triangleright e :: \tau$ 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 eindeutig rückwärtsanwendbar sind¹.

Wir betrachten exemplarisch die folgenden Fälle.

- 1.) Für (CONST), (ID), (ABSTR) und (EMPTY) gilt gemäß Lemma 2.3 $e \not\rightarrow$ bzw. $r \not\rightarrow$.
- 2.) $\Gamma^* \triangleright e :: \tau$ mit Typregel (EQUIV).

Nach Voraussetzung existiert ein $\tau' \in Type$, so dass $\Gamma^* \triangleright e :: \tau'$ mit $\tau' \sim \tau$ gilt. Mit Induktionsvoraussetzung folgt daraus

$$\Gamma^* \triangleright e' :: \tau',$$

woraus dann wegen $\tau' \sim \tau$ mit Typregel (EQUIV) das erwartete Ergebnis

$$\Gamma^* \triangleright e' :: \tau$$

folgt.

- 3.) $\Gamma^* \triangleright e_1 e_2 :: \tau$ mit Typregel (APP).

Nach Voraussetzung gilt $\Gamma^* \triangleright e_1 :: \tau_2 \rightarrow \tau$ und $\Gamma^* \triangleright e_2 :: \tau_2$. Der small step $e_1 e_2 \rightarrow e'$ 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.) $e_1 e_2 \rightarrow e'$ mit (APP-LEFT) impliziert $e' = e'_1 e_2$ und $e_1 \rightarrow e'_1$. Nach Induktionsvoraussetzung folgt $\Gamma^* \triangleright e'_1 :: \tau_2 \rightarrow \tau$, also $\Gamma^* \triangleright e'_1 e_2 :: \tau$ mit Typregel (APP).
- 2.) $e_1 e_2 \rightarrow e'$ mit small step Regel (APP-RIGHT) kann nur aus $e_1 \in Val$, $e_2 \rightarrow e'_2$ und $e' = e_1 e'_2$ folgen. Die Behauptung folgt dann mit Induktionsvoraussetzung und Typregel (APP).

¹Genauer gesagt, kann auf jeden Ausdrucks nicht nur die eigentlich für diesen Ausdruck vorgesehene Typregel, sondern ebenfalls die (EQUIV)-Regel, angewandt werden.

- 3.) Wenn $e_1 e_2 \rightarrow e'$ mit (BETA-V) hergeleitet wurde, gilt $e_1 = \lambda x : \tau_1.e'_1$ und $e_2 \in Val$, sowie $e' = e'_1[e_2/x]$. Nach Lemma 4.9 gilt

$$\Gamma^*[\tau_1/x] \triangleright e'_1 :: \tau'$$

mit $\tau_1 \rightarrow \tau' \sim \tau_2 \rightarrow \tau$. Nach Definition von \sim (4.4) also insbesondere $\tau_1 \sim \tau_2$ und $\tau' \sim \tau$. Mit Typregel (EQUIV) folgt aus $\Gamma^* \triangleright e_2 :: \tau_2$ und $\tau_2 \sim \tau_1$, dass auch

$$\Gamma^* \triangleright e_2 :: \tau_1$$

gilt. Nach Substitutionslemma (4.7) folgt somit

$$\Gamma^* \triangleright e'_1[e_2/x] :: \tau',$$

und wegen $\tau' \sim \tau$ schliesslich

$$\Gamma^* \triangleright e'_1[e_2/x] :: \tau$$

mit Typregel (EQUIV).

- 4.) $e_1 e_2 \rightarrow e'$ mit small step Regel (OP) bedingt $e_1 = op\ v_1$, $e_2 = v_2 \in Val$ und $e' = op^I(v_1, v_2)$. Mit Lemma 4.9 folgt, dass

$$\Gamma^* \triangleright op :: \tau_1 \rightarrow \tau_2 \rightarrow \tau$$

und

$$\Gamma^* \triangleright v_1 :: \tau_1$$

gilt. Nach Lemma 4.9 folgt weiterhin, dass $\tau_1 \sim \mathbf{int}$, $\tau_2 \sim \mathbf{int}$, und $\tau \sim \mathbf{int}$ für $op \in \{+, -, *\}$ bzw. $\tau \sim \mathbf{bool}$ für $op \in \{\leq, \geq, <, >, =\}$.

Sei also $op \in \{+, -, *\}$, dann gilt $op^I(v_1, v_2) \in Int$. Mit Typregel (CONST) folgt

$$\Gamma^* \triangleright op^I(v_1, v_2) :: \mathbf{int},$$

und wegen $\tau \sim \mathbf{int}$ folgt schliesslich

$$\Gamma^* \triangleright op^I(v_1, v_2) :: \tau$$

mit Typregel (EQUIV).

Für $op \in \{\leq, \geq, <, >, =\}$ folgt die Behauptung analog.

- 4.) $\Gamma^* \triangleright \mathbf{object}(self : \tau) \ r \ \mathbf{end} :: \tau$ mit Typregel (OBJECT').

Der small step $e \rightarrow e'$ kann nur mit der small step Regel (OBJECT-EVAL) aus $r \rightarrow r'$ folgen, und es gilt $e' = \mathbf{object}(self : \tau) r' \mathbf{end}$. Nach Voraussetzung existiert ein $\phi \in RType$ mit

$$\Gamma^*[\tau /_{self}] \triangleright r :: \phi$$

und $\langle \phi \rangle \sim \tau$. Mit Induktionsvoraussetzung folgt daraus

$$\Gamma^*[\tau /_{self}] \triangleright r' :: \phi,$$

also mit Typregel (OBJECT') die Behauptung.

5.) $\Gamma^* \triangleright e_1 \# m :: \tau$ mit Typregel (SEND).

Dann gilt $\Gamma^* \triangleright e_1 :: \langle m : \tau; \phi \rangle$. Der small step $e_1 \# m \rightarrow e'$ kann nur mit den small step Regeln (SEND-EVAL) oder (SEND-UNFOLD) hergeleitet worden sein.

- 1.) Der small step $e_1 \# m \rightarrow e'_1 \# m$ mit Regel (SEND-EVAL) kann nur aus $e_1 \rightarrow e'_1$ folgen. Nach Induktionsvoraussetzung gilt

$$\Gamma^* \triangleright e'_1 :: \langle m : \tau; \phi \rangle,$$

also folgt

$$\Gamma^* \triangleright e'_1 \# m :: \tau$$

mit Typregel (SEND).

- 2.) $e_1 \# m \rightarrow e'$ mit (SEND-UNFOLD) bedingt $e_1 = \mathbf{object}(self : \tau_1) \omega \mathbf{end}$ und $e' = \omega[e_1 /_{self}] \# m$. Nach Lemma 4.9 gilt

$$\Gamma^*[\tau_1 /_{self}] \triangleright \omega :: \phi_1$$

mit $\langle \phi_1 \rangle \sim \langle m : \tau; \phi \rangle \sim \tau_1$. Also existieren nach Definition 4.4 $\tau' \in Type$ und $\phi'_1 \in RType$ mit $\phi_1 = (m : \tau'; \phi'_1)$ und $\tau' \sim \tau$. Nach Voraussetzung gilt schon

$$\Gamma^* \triangleright \mathbf{object}(self : \tau_1) \omega \mathbf{end} :: \langle m : \tau; \phi \rangle,$$

und nach Definition 2.18 gilt $\Gamma^* = (\Gamma^*)^*$, somit also

$$(\Gamma^*)^* \triangleright \mathbf{object}(self : \tau_1) \omega \mathbf{end} :: \langle m : \tau; \phi \rangle.$$

Wegen $\langle m : \tau; \phi \rangle \sim \tau_1$ folgt daraus

$$(\Gamma^*)^* \triangleright \mathbf{object}(self : \tau_1) \omega \mathbf{end} :: \tau_1$$

mit Typregel (EQUIV). Darauf lässt sich nun das spezielle *self*-Substitutions-Lemma (4.8) anwenden, und wir erhalten

$$\Gamma^* \triangleright \omega[\mathbf{object}(self:\tau_1) \ \omega \ \mathbf{end} /_{self}] :: \phi_1.$$

Da $\phi_1 = (m : \tau'; \phi'_1)$ folgt weiter mit Typregel (SEND')

$$\Gamma^* \triangleright \omega[\mathbf{object}(self:\tau_1) \ \omega \ \mathbf{end} /_{self}] \# m :: \tau',$$

und wegen $\tau' \sim \tau$ folgt schliesslich mit Typregel (EQUIV) die Behauptung.

Die übrigen Fälle verlaufen analog. \square

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 4.10 (Canonical Forms) *Sei $v \in Val$, $\tau \in Type$, und gelte $[] \triangleright v :: \tau$.*

- (a) *Wenn $\tau \sim \mathbf{int}$, dann gilt $v \in Int$.*
- (b) *Wenn $\tau \sim \tau_1 \rightarrow \tau_2$, 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 $v = \mathbf{object}(self : \tau') \ \omega \ \mathbf{end}$.*

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 $[] \triangleright v :: \tau$ mit $\tau \sim \mathbf{int}$ 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 \mathbf{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 $v \in \text{Const}$. Die einzige in Frage kommende Regel hier ist (INT), denn von den möglichen Typen von Konstanten ist nach Lemma 4.4 (Äquivalenz-Lemma) einzig **int** äquivalent zu **int**. Die Regel (INT) wiederum impliziert $v \in \text{Int}$, 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. \square

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 $\mathcal{L}_o^{\text{sub}}$ (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) $\forall e \in \text{Exp}, \tau \in \text{Type} : [] \triangleright e :: \tau \Rightarrow (e \in \text{Val} \vee \exists e' \in \text{Exp} : e \rightarrow e')$
- (b) $\forall \Gamma \in \text{TEnv}, r \in \text{Row}, \phi \in \text{RType} : \Gamma^+ \triangleright r :: \phi \Rightarrow (r \in \text{RVal} \vee \exists r' \in \text{Row} : r \rightarrow r')$

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 $e \in \text{Val}$, und für (EMPTY) gilt $r \in \text{RVal}$.
- 2.) $[] \triangleright e :: \tau$ mit Typregel (EQUIV).

Nach Voraussetzung existiert ein $\tau' \in \text{Type}$, 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 $[] \triangleright e_1 :: \tau_1 \rightarrow \tau$ und $[] \triangleright e_2 :: \tau_2$, und es ist nach der Form der Teilausdrücke e_1, e_2 zu unterscheiden.

Falls $e_1 \notin \text{Val}$, so existiert nach Induktionsvoraussetzung ein $e'_1 \in \text{Exp}$ mit $e_1 \rightarrow e'_1$, und somit ein small step $e_1 e_2 \rightarrow e'_1 e_2$ mit small step Regel (APP-LEFT).

Entsprechend existiert für $e_1 \in \text{Val}$ und $e_2 \notin \text{Val}$ ein small step mit Regel (APP-RIGHT).

Somit bleibt noch der Fall $e_1, e_2 \in Val$ zu betrachten. Nach Canonical Forms Lemma (4.10) sind wegen $[] \triangleright e_1 :: \tau_2 \rightarrow \tau$ und der Reflexivität von \sim für e_1 die folgenden drei Fälle zu unterscheiden:

1.) $e_1 \in Op$

Dann gilt nach Definition 2.4 $e_1 e_2 \in Val$.

2.) $e_1 = op\ v_1$ mit $op \in Op$ und $v_1 \in Val$

Nach Lemma 4.9 existiert ein $\tau_1 \in Type$ mit $[] \triangleright op :: \tau_1 \rightarrow \tau_2 \rightarrow \tau$ und $[] \triangleright v_1 :: \tau_1$. Nach Definition 2.17 folgt mit dem Äquivalenz-Lemma (4.4) $\tau_1 \sim \mathbf{int}$ und $\tau_2 \sim \mathbf{int}$, also $v_1, e_2 \in Int$ nach Lemma 4.10. Somit existiert ein small step $op\ v_1\ e_2 \rightarrow op^I(v_1, e_2)$ mit small step Regel (OP).

3.) Für $e_1 = \lambda x : \tau'_2. e'_1$ existiert ein small step $(\lambda x : \tau'_2. e'_1)\ e_2 \rightarrow e'_1[e_2/x]$ mit Regel (BETA-V), da nach Voraussetzung $e_2 \in Val$.

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 $e_1 \in Val$ oder es existiert ein $e'_1 \in Exp$ mit $e_1 \rightarrow e'_1$.

Sei also $e_1 \in Val$, dann gilt nach Lemma 4.10, dass $e_1 = \mathbf{object}\ (self : \langle \phi \rangle) \ \omega\ \mathbf{end}$. Also existiert ein small step

$$(\mathbf{object}\ (self : \langle \phi \rangle) \ \omega\ \mathbf{end}) \# m \rightarrow \omega[e_1/self] \# m$$

mit small step Regel (SEND-UNFOLD).

Ist andererseits $e_1 \notin Val$, so existiert ein small step

$$e_1 \# m \rightarrow e'_1 \# m$$

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 $(\Gamma^+)^* \triangleright e' :: \tau'$ und $\Gamma^+ \triangleright r' :: \phi'$ folgen, wobei zu beachten gilt, dass $(\Gamma^+)^* = []$. Nach Induktionsvoraussetzung gilt dann entweder $e' \in Val$ oder es existiert ein $e'' \in Exp$ mit $e' \rightarrow e''$. Ebenso gilt entweder $r' \in RVal$ oder existiert ein $r'' \in Row$ mit $r' \rightarrow r''$.

Seien $e' \in Val$ und $r' \in RVal$, dann gilt $(\mathbf{val}\ a = e';\ r') \in RVal$ nach Definition 2.4.

Falls $e' \notin Val$, also $e' \rightarrow e''$, so existiert ein small step

$$\mathbf{val}\ a = e';\ r' \rightarrow \mathbf{val}\ a = e'';\ r'$$

mit small step Regel (ATTR-LEFT).

Analog existiert für $e' \in \text{Val}$ und $r' \notin \text{RVal}$, also $r' \rightarrow r''$, ein small step

$$\mathbf{val} \ a = e'; r' \rightarrow \mathbf{val} \ a = e'; r''$$

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 objekt-orientierten Programmiersprache kombinieren lassen.

TODO: Mehr Prosa

Definition 4.8 Die Relationen \lesssim_n seien wie folgt induktiv definiert

$$\begin{aligned} \lesssim_0 &= \text{Type}^2 \\ \lesssim_{n+1} &= \{(\tau, \tau') \in \text{Type}^2 \mid \text{root}(\tau) = \text{root}(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}\} \\ &\cup \{(\tau, \tau') \in \text{Type}^2 \mid \text{root}(\tau) = \text{root}(\tau') = \rightarrow \\ &\quad \wedge \text{child}_1(\tau') \lesssim_n \text{child}_1(\tau) \wedge \text{child}_2(\tau) \lesssim_n \text{child}_2(\tau')\} \\ &\cup \{(\tau, \tau') \in \text{Type}^2 \mid \text{root}(\tau) = \langle m_1; \dots; m_{k+l} \rangle \wedge \text{root}(\tau') = \langle m_1; \dots; m_k \rangle \\ &\quad \wedge \forall 1 \leq i \leq k : \text{child}_i(\tau) \lesssim_n \text{child}_i(\tau')\} \end{aligned}$$

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 Subtyperegeln (Sm-Refl, Sm-Arrow und Sm-Object)

Lemma 4.11 *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 $\lesssim_0 = \text{Type}^2$ 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 $\text{root}(\tau_2)$:

- 1.) $\text{root}(\tau_2) \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

Dann gilt $\text{root}(\tau_1) = \text{root}(\tau_2) = \text{root}(\tau_3)$, also auch $(\tau_1, \tau_3) \in \lesssim_{n+1}$.

- 2.) $\text{root}(\tau_2) = \rightarrow$

Dann gilt wieder $\text{root}(\tau_1) = \text{root}(\tau_2) = \text{root}(\tau_3)$. Weiterhin gelten die folgenden Ungleichungen:

$$\begin{array}{llll} \text{child}_1(\tau_2) & \lesssim_n & \text{child}_1(\tau_1) & \text{child}_2(\tau_1) \lesssim_n \text{child}_2(\tau_2) \\ \text{child}_1(\tau_3) & \lesssim_n & \text{child}_1(\tau_2) & \text{child}_2(\tau_2) \lesssim_n \text{child}_2(\tau_3) \end{array}$$

Wegen der nach Induktionsvoraussetzung geltenden Transitivität von \lesssim_n folgt damit, dass auch die Ungleichungen

$$\begin{array}{ll} \text{child}_1(\tau_3) & \lesssim_n \text{child}_1(\tau_1) \\ \text{child}_2(\tau_1) & \lesssim_n \text{child}_2(\tau_3) \end{array}$$

gelten, und somit $(\tau_1, \tau_3) \in \lesssim_{n+1}$ nach Definition 4.8.

3.) $root(\tau_2) = \langle m_1; \dots; m_k \rangle$

Dann existieren für τ_1 und τ_3 nach Definition $j, k \in \mathbb{N}$ mit $j \leq k \leq l$, so dass $root(\tau_1) = \langle m_1; \dots; m_l \rangle$ und $root(\tau_3) = \langle m_1; \dots; m_j \rangle$. 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. \square

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 4.12 $\forall \tau, \tau' \in Type : \tau \lesssim \tau' \wedge \tau' \lesssim \tau \Leftrightarrow \tau \sim \tau'$

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

$$\forall 1 \leq i \leq \text{arity}(\tau) : \text{child}_i(\tau) \lesssim_n \text{child}_i(\tau') \wedge \text{child}_i(\tau') \lesssim_n \text{child}_i(\tau).$$

gilt. Damit folgt insgesamt die Behauptung mit einer einfachen Fallunterscheidung nach der Form von $\text{root}(\tau)$ und Anwenden der Definition von \lesssim_{n+1} (4.8).

„ \Rightarrow “ Auch diese Implikation beweisen wir durch vollständige Induktion über n :

- $n = 0$

Klar, da $\lesssim_0 = \text{Type} \times \text{Type} = \sim_0$.

- $n \rightsquigarrow n + 1$

Seien $\tau, \tau' \in \text{Type}$, und gelte sowohl $\tau \lesssim_{n+1} \tau'$ als auch $\tau' \lesssim_{n+1} \tau$. Dann zeigen wir abhängig von der Form von $\text{root}(\tau)$, dass auch $\tau \sim_{n+1} \tau'$ gilt:

- 1.) $\text{root}(\tau) = \text{root}(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

Trivial.

- 2.) $\text{root}(\tau) = \text{root}(\tau') = \rightarrow$

Dann gelten nach Voraussetzung die folgenden Ungleichungen:

$$\begin{array}{lll} \text{child}_1(\tau') \lesssim_n \text{child}_1(\tau) & \text{child}_2(\tau) \lesssim_n \text{child}_2(\tau') \\ \text{child}_1(\tau) \lesssim_n \text{child}_1(\tau') & \text{child}_2(\tau') \lesssim_n \text{child}_2(\tau) \end{array}$$

Darauf nun lässt sich die Induktionsvoraussetzung anwenden, und wir erhalten

$$\text{child}_1(\tau) \sim_n \text{child}_1(\tau')$$

und

$$\text{child}_2(\tau) \sim_n \text{child}_2(\tau'),$$

woraus dann nach Definition von \sim_{n+1} unmittelbar die Behauptung folgt.

- 3.) $\text{root}(\tau) = \langle m_1; \dots; m_k \rangle, \text{root}(\tau') = \langle m_1; \dots; m_l \rangle$

Da sowohl $\tau \lesssim_{n+1} \tau'$ wie auch $\tau' \lesssim_{n+1} \tau$ gilt, muss $k = l$ gelten, und die Behauptung folgt dann analog zum vorangegangenen Fall. \square

Genaugenommen ist das Lemma sogar eine stärkere Aussage. Für die eigentliche Antisymmetrie würde eine Implikation, wie im folgenden Korollar dargestellt, genügen.

Korollar 4.3 $\forall \tau, \tau' \in \text{Type} : \tau \lesssim \tau' \wedge \tau' \lesssim \tau \Rightarrow \tau \sim \tau'$

Beweis: Folgt direkt aus Lemma 4.12. \square

TODO: Prosa

Definition 4.9 Sei $\mathcal{S} : TypeRel \rightarrow TypeRel$ definiert durch

$$\begin{aligned} \mathcal{S}(R) = & \{(\tau, \tau') \in Type^2 \mid root(\tau) = root(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}\} \\ \cup & \{(\tau, \tau') \in Type^2 \mid root(\tau) = root(\tau') = \rightarrow \\ & \quad \wedge (child_1(\tau'), child_1(\tau)) \in R \wedge (child_2(\tau), child_2(\tau')) \in R\} \\ \cup & \{(\tau, \tau') \in Type^2 \mid root(\tau) = \langle m_1; \dots; m_{k+l} \rangle \wedge root(\tau') = \langle m_1; \dots; m_k \rangle \\ & \quad \wedge \forall 1 \leq i \leq k : (child_i(\tau), child_i(\tau')) \in R\}, \end{aligned}$$

also eine totale Abbildung auf binären Typrelationen.

Wir wollen wieder zeigen, dass die Typrelation \lesssim ein Fixpunkt der Abbildung \mathcal{S} ist. Für die Programmiersprache \mathcal{L}_o^{rt} hatten wir bereits bewiesen, dass $(TypeRel, \supseteq)$ eine dcpo mit einem kleinstem Element $Type \times Type$ ist (Lemma 4.2). Für die Anwendung des Fixpunktsatzes von Kleene bleibt hier also lediglich zu zeigen, dass die Abbildung \mathcal{S} stetig ist.

Lemma 4.13 \mathcal{S} ist eine stetige Abbildung auf $(TypeRel, \supseteq)$.

Beweis: Wie zuvor im Beweis des Lemmas für die Programmiersprache \mathcal{L}_o^{rt} (Lemma 4.2) zeigen wir zunächst, dass \mathcal{S} eine monotone Abbildung auf $(TypeRel, \supseteq)$ ist. Seien dazu wieder $R_1, R_2 \in TypeRel$ mit $R_1 \supseteq R_2$. Dann ist zu zeigen, dass für alle $(\tau, \tau') \in \mathcal{S}(R_2)$ auch $(\tau, \tau') \in \mathcal{S}(R_1)$ gilt. Wir unterscheiden dazu wieder nach der Form von $root(\tau)$.

1.) $root(\tau) \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

Dann ist $root(\tau) = root(\tau')$, und nach Definition 4.9 gilt $(\tau, \tau') \in \mathcal{S}(R)$ für alle $R \in TypeRel$, also insbesondere $(\tau, \tau') \in \mathcal{S}(R_1)$.

2.) $root(\tau) = \rightarrow$

Nach Definition 4.9 gilt wieder $root(\tau) = root(\tau')$, $(child_1(\tau'), child_1(\tau)) \in R_2$ sowie $(child_2(\tau), child_2(\tau')) \in R_2$. Also insbesondere $(child_1(\tau'), child_1(\tau)) \in R_1$ und $(child_2(\tau), child_2(\tau')) \in R_1$, denn es gilt $R_2 \subseteq R_1$, und somit folgt die Behauptung unmittelbar nach Definition.

3.) $root(\tau) = \langle m_1; \dots; m_{k+l} \rangle$

Folgt analog zum vorangegangenen Fall.

Damit bleibt zum Beweis der Stetigkeit von \mathcal{S} zu zeigen, dass

$$\mathcal{S}(\bigsqcup \Delta) \supseteq \bigsqcup \mathcal{S}(\Delta)$$

für alle gerichteten Teilmengen $\Delta \subseteq TypeRel$ gilt. Sei also wieder $\Delta \subseteq TypeRel$ eine gerichtete Teilmenge und sei $(\tau, \tau') \in \bigcap \mathcal{S}(\Delta) = \bigcap \{\mathcal{S}(R) \mid R \in \Delta\}$, also $(\tau, \tau') \in \mathcal{S}(R)$ für alle $R \in \Delta$. Durch Fallunterscheidung nach der Form von $root(\tau)$ zeigen wir, dass dann auch $(\tau, \tau') \in \mathcal{S}(\bigcap \Delta)$ gilt:

- 1.) $root(\tau) \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

Klar.

- 2.) $root(\tau) = \rightarrow$

Dann ist $root(\tau) = root(\tau')$ und für alle $R \in \Delta$ gilt $(child_1(\tau'), child_1(\tau)) \in R$ sowie $(child_2(\tau), child_2(\tau')) \in R$. Das bedeutet aber insbesondere, dass dann $(child_1(\tau'), child_1(\tau)), (child_2(\tau), child_2(\tau')) \in \bigcap \Delta$ gilt, also folgt nach Definition 4.9, dass auch $(\tau, \tau') \in \mathcal{S}(\bigcap \Delta)$ gilt.

- 3.) $root(\tau) = \langle m_1; \dots; m_{k+l} \rangle$

Folgt wieder analog zum vorherigen Fall. \square

Damit lässt sich nun das Fixpunkt-Lemma für die Relation \lesssim formulieren und beweisen, analog zum Fixpunkt-Lemma für die Programmiersprache \mathcal{L}_o^{rt} (4.3), welches uns einen einfacheren Umgang mit der Subtyprelation \lesssim ermöglichen soll.

Lemma 4.14 (Fixpunkt-Lemma für \mathcal{L}_o^{srt}) $\mu\mathcal{S} = \lesssim$

In Worten also, die Subtyprelation \lesssim ist der kleinste Fixpunkt der generierenden Funktion \mathcal{S} .

Beweis: Nach Lemma 4.2 ist $(TypeRel, \supseteq)$ ein semantischer Bereich, und nach Lemma 4.13 ist die totale Abbildung \mathcal{S} stetig auf $(TypeRel, \supseteq)$. Damit lässt sich Satz 1.1 (Fixpunktsatz von Kleene) anwenden, nach dem mit $\mu\mathcal{S} = \bigsqcup_{n \in \mathbb{N}} \mathcal{S}^n(Type \times Type)$ der kleinste Fixpunkt von \mathcal{S} existiert. Es bleibt also zu zeigen, dass die Relation \lesssim gerade diesem Fixpunkt entspricht.

Dazu zeigen wir zunächst durch vollständige Induktion, dass $\lesssim_n = \mathcal{S}^n(Type \times Type)$ gilt:

- $n = 0$

Klar, da $\lesssim_0 = Type \times Type = \mathcal{S}^0(Type \times Type)$.

- $n \rightsquigarrow n + 1$

Nach Induktionsvoraussetzung gilt $\lesssim_n = \mathcal{S}^n(Type \times Type)$. 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 $\lesssim = \bigcap_{n \in \mathbb{N}} \mathcal{S}^n(Type \times Type) = \bigsqcup_{n \in \mathbb{N}} \mathcal{S}^n(Type \times Type)$, was zu zeigen war. \square

Korollar 4.4 Die Relation \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 4.15 (Subtyping-Lemma für \mathcal{L}_o^{srt}) $\tau \lesssim \tau'$ 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) $\tau \sim \tau_1 \rightarrow \tau_2$ und $\tau' \sim \tau'_1 \rightarrow \tau'_2$ mit $\tau'_1 \lesssim \tau_1$ und $\tau_2 \lesssim \tau'_2$
- (c) $\tau \sim \langle m_1 : \tau_1; \dots; m_{k+l} : \tau_{k+l} \rangle$ und $\tau' \sim \langle m_1 : \tau'_1; \dots; m_k : \tau'_k \rangle$ mit $\tau_i \lesssim \tau'_i$ für alle $i = 1, \dots, k$

Beweis: Folgt unmittelbar aus der Tatsache, dass die Subtyprelation \lesssim Fixpunkt der stetigen Abbildung \mathcal{S} ist (Lemma 4.14), denn die Aussagen des Lemmas entsprechen genau der Definition der Abbildung \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 4.10 (Gültige Typurteile für \mathcal{L}_o^{srt}) Ein Typurteil $\Gamma \triangleright_e e :: \tau$ oder $\Gamma \triangleright_r r :: \phi$ heisst *gültig* für \mathcal{L}_o^{srt} , wenn es sich mit den Typregeln der Programmiersprache \mathcal{L}_o^{rt} aus Definition 4.7 mit Ausnahme der (EQUIV)-Regel, sowie der folgenden neuen Typregel

$$(\text{SUBSUME}') \quad \frac{\Gamma \triangleright_e e :: \tau' \quad \tau' \lesssim \tau}{\Gamma \triangleright_e e :: \tau}$$

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 4.16 (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: 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. \square

Lemma 4.17 (Koinzidenzlemma für \mathcal{L}_o^{srt})

- (a) $\forall \Gamma_1, \Gamma_2 \in TEnv, e \in Exp, \tau \in Type : \Gamma_1 \triangleright e :: \tau \wedge \Gamma_1 =_{free(e)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright e :: \tau$
- (b) $\forall \Gamma_1, \Gamma_2 \in TEnv, r \in Row, \phi \in RType : \Gamma_1 \triangleright r :: \phi \wedge \Gamma_1 =_{free(r)} \Gamma_2 \Rightarrow \Gamma_2 \triangleright r :: \phi$

Beweis: Ebenfalls wie gehabt, wobei der Fall der (SUBSUME')-Regel direkt mit Induktionsvoraussetzung folgt. \square

Wie zuvor benötigen wir wieder zwei Substitutionslemmata, ein allgemeines für Attribute und Variablen, und ein spezielles für die *self*-Substitution.

Lemma 4.18 (Typurteile und Substitution) Sei $id \in Attribute \cup Var$, $\Gamma \in TEnv$, $\tau \in Type$ und $e \in Exp$. Dann gilt:

- (a) $\forall e' \in Exp : \forall \tau' \in Type : \Gamma[\tau/id] \triangleright e' :: \tau' \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright e'[e/id] :: \tau'$
- (b) $\forall r \in Row : \forall \phi \in RType : \Gamma[\tau/id] \triangleright r :: \phi \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright r[e/id] :: \phi$

Beweis: Ähnlich wie im Beweis des Lemmas für die Programmiersprache \mathcal{L}_o^{rt} (4.7), wobei der Fall der (SUBSUME')-Regel ebenfalls trivialerweise mit Induktionsvoraussetzung folgt. \square

Lemma 4.19 (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) $\Gamma^* \triangleright \mathbf{object}(self : \tau) \ r \ \mathbf{end} :: \tau$
- (3) $\forall a \in \text{dom}(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright e[\mathbf{object}(self:\tau) \ r \ \mathbf{end}/self] :: \tau'$

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

- (1) $\Gamma[\tau/self] \triangleright r' :: \phi$
- (2) $\Gamma^* \triangleright \mathbf{object}(self : \tau) \ r \oplus r' \ \mathbf{end} :: \tau$
- (3) $\forall a \in \text{dom}(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright r'[\mathbf{object}(self:\tau) \ r \oplus r' \ \mathbf{end}/self] :: \phi$

Beweis: Der Beweis entspricht ebenfalls im wesentlichen den Beweisen des Lemmas für zuvor behandelte Programmiersprachen. Wie früher folgt gemäß Lemma 4.16, dass die Substitution in jedem Fall definiert ist. Der eigentliche Beweis erfolgt dann durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[\tau/self] \triangleright e :: \tau'$ und $\Gamma[\tau/self] \triangleright r' :: \phi$, 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 $\tau'' \in Type$ 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. \square

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 Lemma, 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 4.20 (Umkehrung der Typrelation)

- (a) Wenn $\Gamma \triangleright op :: \tau$, dann gilt $\tau \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ für $op \in \{+, -, *\}$ oder $\tau \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ 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 $\tau_1 \rightarrow \tau_2 \lesssim \tau$.
- (c) Wenn $\Gamma \triangleright e_1 e_2 :: \tau$, dann gilt $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau$, $\Gamma \triangleright e_2 :: \tau_2$ und $\tau_2 \lesssim \tau'_2$.
- (d) Wenn $\Gamma \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau'$, dann gilt $\Gamma^*[\tau/self] \triangleright r :: \phi$ 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 $\tau' \in Type$ mit $\Gamma \triangleright op :: \tau'$ und $\tau' \lesssim \tau$. Nach Induktionsvoraussetzung gilt $\tau' \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ für $op \in \{+, -, *\}$ oder $\tau' \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ für $op \in \{\leq, \geq, <, >, =\}$.

Wir betrachten ausschliesslich den Fall $\tau' \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$, der Fall $\tau' \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ folgt dann analog. Wegen Lemma 4.12 folgt $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int} \lesssim \tau'$ und wegen der Transitivität von \lesssim folgt $\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int} \lesssim \tau$. Nach Subtyping-Lemma (4.15) existieren dann $\tau_1, \tau_2 \in Type$ mit $\tau \sim \tau_1 \rightarrow \tau_2$, $\tau_1 \lesssim \mathbf{int}$ und $\mathbf{int} \rightarrow \mathbf{int} \lesssim \tau_2$. Nach erneuter Anwendung des Lemmas erhalten wir $\tau'_2, \tau''_2 \in Type$ mit $\tau'_2 \lesssim \mathbf{int}$ und $\mathbf{int} \lesssim \tau''_2$. Wiederum mit Lemma 4.15 folgt schliesslich $\tau_1 \sim \tau'_2 \sim \tau''_2 \sim \mathbf{int}$. Somit gilt schlussendlich $\tau \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$, 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 $\tau = \tau_1 \rightarrow \tau_2$. Da \lesssim reflexiv ist, folgt $\tau_1 \rightarrow \tau_2 \lesssim \tau$.
 - 2.) Falls $\Gamma \triangleright \lambda x : \tau_1. e :: \tau$ mit Typregel (SUBSUME') hergeleitet worden ist, existiert nach Voraussetzung ein $\tau' \in \text{Type}$ 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 $\tau_1 \rightarrow \tau_2 \lesssim \tau'$. Wegen der Transitivität von \lesssim folgt damit $\tau_1 \rightarrow \tau_2 \lesssim \tau$, 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 $\Gamma \triangleright e_1 :: \tau_2 \rightarrow \tau$ 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 $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau'$, $\Gamma \triangleright e_2 :: \tau_2$ und $\tau_2 \lesssim \tau'_2$. Wegen $\tau' \lesssim \tau$ folgt somit schliesslich $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau$ mit Typregel (SUBSUME') aus $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau'$.
- (d) Wiederum ist klar, dass das Typurteil $\Gamma \triangleright \mathbf{object}(\text{self} : \tau) r \mathbf{end} :: \tau'$ ausschliesslich mit den Typregeln (OBJECT') und (SUBSUME') hergeleitet worden sein kann, und somit nur diese Fälle zu betrachten sind:
- 1.) Falls $\Gamma \triangleright \mathbf{object}(\text{self} : \tau) r \mathbf{end} :: \tau'$ mit einer Anwendung der Typregel (OBJECT') hergeleitet wurde, so gilt nach Voraussetzung $\Gamma^*[\tau/\text{self}] \triangleright r :: \phi$ mit $\langle \phi \rangle \sim \tau$ und $\tau = \tau'$, woraus unmittelbar $\tau \lesssim \tau'$ folgt.
 - 2.) Sollte andererseits $\Gamma \triangleright \mathbf{object}(\text{self} : \tau) r \mathbf{end} :: \tau'$ mit Typregel (SUBSUME') hergeleitet worden sein, so existiert nach Voraussetzung ein $\tau'' \in \text{Type}$ mit $\Gamma \triangleright \mathbf{object}(\text{self} : \tau) r \mathbf{end} :: \tau''$ und $\tau'' \lesssim \tau'$. Darauf lässt sich nun die Induktionsvoraussetzung anwenden und wir erhalten $\Gamma^*[\tau/\text{self}] \triangleright r :: \phi$ mit $\langle \phi \rangle \sim \tau$ und $\tau \lesssim \tau''$. Wegen der Transitivität von \lesssim folgt schliesslich $\tau \lesssim \tau'$. \square

Wie bereits im Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_o^{\text{sub}}$ und $\mathcal{L}_o^{\text{rt}}$ 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“)

$$(a) \forall \Gamma \in TEnv : \forall e, e' \in Exp : \forall \tau \in Type : \Gamma^* \triangleright e :: \tau \wedge e \rightarrow e' \Rightarrow \Gamma^* \triangleright e' :: \tau$$

(b) $\forall \Gamma \in TEnv : \forall r, r' \in Row : \forall \phi \in RType : \Gamma \triangleright r :: \phi \wedge r \rightarrow r' \Rightarrow \Gamma \triangleright r' :: \phi$

Beweis: Der Beweis erfolgt selbstverständlich wieder durch simultane Induktion über die Länge der Typherleitungen von $\Gamma^* \triangleright e :: \tau$ 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 $e \not\rightarrow$ bzw. $r \not\rightarrow$.
- 2.) $\Gamma^* \triangleright e :: \tau$ mit Typregel (SUBSUME')

Dann gilt nach Voraussetzung, dass ein $\tau' \in Type$ existiert, mit $\Gamma^* \triangleright e :: \tau'$ und $\tau' \lesssim \tau$. Nach Induktionsvoraussetzung folgt

$$\Gamma^* \triangleright e' :: \tau',$$

und wegen $\tau' \lesssim \tau$ folgt daraus mit Anwendung von Typregel (SUBSUME') unmittelbar

$$\Gamma^* \triangleright e' :: \tau,$$

was zu zeigen war.

- 3.) $\Gamma^* \triangleright e_1 e_2 :: \tau$ mit Typregel (APP)

Dann muss gelten $\Gamma^* \triangleright e_1 :: \tau_2 \rightarrow \tau$ und $\Gamma^* \triangleright e_2 :: \tau_2$. Der small step $e_1 e_2 \rightarrow e'$ 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.) $e_1 e_2 \rightarrow e'$ mit (APP-LEFT) impliziert $e' = e'_1 e_2$ und $e_1 \rightarrow e'_1$. Darauf lässt sich die Induktionsvoraussetzung anwenden, und wir erhalten

$$\Gamma^* \triangleright e'_1 :: \tau_2 \rightarrow \tau,$$

woraus dann mit Typregel (APP) das erwartete Ergebnis

$$\Gamma^* \triangleright e'_1 e_2 :: \tau$$

folgt.

- 2.) $e_1 e_2 \rightarrow e'$ mit small step Regel (APP-RIGHT) kann nur aus $e_1 \in Val$, $e_2 \rightarrow e'_2$ und $e' = e_1 e'_2$ folgen. Die Behauptung folgt dann mit Induktionsvoraussetzung und Typregel (APP).

- 3.) Falls $e_1 e_2 \rightarrow e'$ mit (BETA-V) hergeleitet wurde, gilt $e_1 = \lambda x : \tau_1. e'_1$ und $e_2 \in \text{Val}$, sowie $e' = e'_1[e_2/x]$. Nach dem vorangegangenen Lemma (4.20) gilt also

$$\Gamma^*[\tau_1 / x] \triangleright e'_1 :: \tau'$$

mit $\tau_1 \rightarrow \tau' \lesssim \tau_2 \rightarrow \tau$. Nach Subtyping-Lemma (4.15) gilt also $\tau_2 \lesssim \tau_1$ und $\tau' \lesssim \tau$, das bedeutet wegen $\Gamma^* \triangleright e_2 :: \tau_2$ folgt mit Typregel (SUBSUME'), dass auch

$$\Gamma^* \triangleright e_2 :: \tau_1$$

gilt. Damit lässt sich nun auf $\Gamma^*[\tau_1 / x] \triangleright e'_1 :: \tau'$ das Substitutions-Lemma (4.18) anwenden, und wir erhalten

$$\Gamma^* \triangleright e'_1[e_2/x] :: \tau',$$

woraus dann wegen $\tau' \lesssim \tau$ mit Typregel (SUBSUME') schliesslich

$$\Gamma^* \triangleright e'_1[e_2/x] :: \tau$$

folgt, was zu zeigen war.

- 4.) Für $e_1 e_2 \rightarrow e'$ mit small step Regel (OP) muss gelten $e_1 = op\ v_1$, $e_2 = v_2 \in \text{Val}$ und $e' = op^I(v_1, v_2)$. Gemäß Lemma 4.20 folgt daraus, dass

$$\Gamma^* \triangleright op :: \tau_1 \rightarrow \tau_2 \rightarrow \tau$$

und

$$\Gamma^* \triangleright v_1 :: \tau_1$$

gilt. Weiterhin folgt nach Lemma 4.20, 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 $op^I(v_1, v_2) \in \text{Int}$. Daraus folgt mit Typregel (CONST), dass

$$\Gamma^* \triangleright op^I(v_1, v_2) :: \mathbf{int}$$

gilt, und wegen $\tau \sim \mathbf{int}$ gilt nach Lemma 4.12 auch $\mathbf{int} \lesssim \tau$, also folgt schliesslich

$$\Gamma^* \triangleright op^I(v_1, v_2) :: \tau$$

mit Typregel (SUBSUME').

4.) $\Gamma^* \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \tau$ mit Typregel (OBJECT')

Das kann nur aus Prämissen der Form $(\Gamma^*)^*[\tau/self] \triangleright r :: \phi$ 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

$$(\Gamma^*)^*[\tau/self] \triangleright r' :: \phi,$$

und mit Typregel (OBJECT') folgt dann unmittelbar die Behauptung.

5.) $\Gamma^* \triangleright e_1 \# m :: \tau$ mit Typregel (SEND)

Dann gilt nach Voraussetzung $\Gamma^* \triangleright e_1 :: \langle m : \tau; \phi \rangle$. 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 (4.20) gilt

$$(\Gamma^*)^*[\tau'/self] \triangleright \omega :: \phi'$$

mit $\langle \phi' \rangle \sim \tau'$ und $\tau' \lesssim \langle m : \tau; \phi \rangle$. Das wiederum bedeutet, dass mit Typregel (SUBSUME')

$$\Gamma^* \triangleright \mathbf{object} (self : \tau') \omega \mathbf{end} :: \tau'$$

folgt. Damit sind alle Voraussetzungen für die Anwendung des speziellen Substitutions-Lemmas (4.19) erfüllt, und wir erhalten

$$\Gamma^* \triangleright \omega[\mathbf{object} (self : \tau') \omega \mathbf{end} / self] :: \phi'.$$

Da $\langle \phi' \rangle \sim \tau'$ und $\tau' \lesssim \langle m : \tau; \phi \rangle$ existieren also nach Subtyping-Lemma (4.15) $\tau'' \in Type$ und $\phi'' \in RType$ 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')

$$\Gamma^* \triangleright \omega[\mathbf{object} (self : \tau') \omega \mathbf{end} / self] \# m :: \tau'',$$

denn $(\Gamma^*)^* = \Gamma^*$, und wegen $\tau'' \lesssim \tau$ folgt schliesslich mit Typregel (SUBSUME') die Behauptung.

Die restlichen Fälle verlaufen ähnlich. \square

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 4.21 (Canonical Forms) *Sei $v \in Val$, $\tau \in Type$, und gelte $[] \triangleright v :: \tau$.*

- (a) *Wenn $\tau \lesssim \mathbf{int}$, dann gilt $v \in Int$.*
- (b) *Wenn $\tau \lesssim \tau_1 \rightarrow \tau_2$, 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 $\tau \lesssim \langle \phi \rangle$, 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} (4.21). \square

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) $\forall e \in Exp, \tau \in Type : [] \triangleright e :: \tau \Rightarrow (e \in Val \vee \exists e' \in Exp : e \rightarrow e')$
- (b) $\forall \Gamma \in TEnv, r \in Row, \phi \in RType : \Gamma^+ \triangleright r :: \phi \Rightarrow (r \in RVal \vee \exists r' \in Row : r \rightarrow r')$

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 $e \in Val$, während für (EMPTY) gilt $r \in RVal$.

2.) $[] \triangleright e :: \tau$ mit Typregel (SUBSUME')

Nach Voraussetzung existiert ein $\tau' \in Type$ mit $[] \triangleright e :: \tau'$ und $\tau' \lesssim \tau$. Also folgt die Behauptung unmittelbar mit Induktionsvoraussetzung.

3.) $[] \triangleright e_1 e_2 :: \tau$ mit Typregel (APP)

Das kann nur aus Prämissen der Form $[] \triangleright e_1 :: \tau_2 \rightarrow \tau$ 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 $e_1 \notin Val$ lässt sich wie früher leicht zeigen, dass ein small step mit (APP-LEFT) existiert. Ebenso einfach zeigt man, dass für $e_1 \in Val$ und $e_2 \notin Val$ ein small step mit (APP-RIGHT) existiert.

Es bleibt also der Fall $e_1, e_2 \in Val$ zu betrachten. Da \lesssim reflexiv ist, lässt sich das Canonical Forms Lemma (4.21) anwenden, demzufolge drei Fälle zu unterscheiden sind:

1.) $e_1 \in Op$

Trivial.

2.) $e_1 = op v_1$ mit $op \in Op$ und $v_1 \in Val$

Dann existiert nach Lemma 4.20 ein $\tau_1 \in Type$ mit $[] \triangleright op :: \tau_1 \rightarrow \tau_2 \rightarrow \tau$ und $[] \triangleright v_1 :: \tau_1$. Wie früher folgt daraus leicht, dass $v_1, e_2 \in Int$, und somit ein small step $op v_1 e_2 \rightarrow op^I(v_1, e_2)$ mit small step Regel (OP) existiert.

3.) $e_1 = \lambda x : \tau'_2. e'_1$ mit $x \in Var$, $\tau'_2 \in Type$ und $e'_1 \in Exp$

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 $e_1 \in Val$ gilt oder ein small step $e_1 \rightarrow e'_1$ existiert.

Sei also $e_1 \in Val$, dann gilt nach Lemma 4.21, dass $e_1 = \mathbf{object}(self : \langle \phi \rangle) \omega \mathbf{end}$. Also existiert ein small step

$$(\mathbf{object}(self : \langle \phi \rangle) \omega \mathbf{end}) \# m \rightarrow \omega^{[e_1 / self]} \# m$$

mit small step Regel (SEND-UNFOLD).

Ist andererseits $e_1 \notin Val$, so existiert ein small step

$$e_1 \# m \rightarrow e'_1 \# m$$

mit small step Regel (SEND-EVAL).

Die restlichen Fälle verlaufen analog. \square

Wie zuvor folgt dann mit Hilfe der Preservation- und Progress-Theoreme die Typsicherheit der Programmiersprache \mathcal{L}_o^{srt} . Der Vollständigkeit halber 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. \square

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 ist. 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 (Abstrakte Syntax von \mathcal{L}_c)

- (a) Die Menge *Body* aller *Klassenrümpfe* b von \mathcal{L}_c ist durch die folgende kontextfreie Grammatik definiert:

$$b ::= \text{inherit } a_1, \dots, a_k \text{ from } e; b_1 \\ | r$$

Für a_1, \dots, a_k schreiben wir kurz A und betrachten A gleichzeitig als Menge, also $A = \{a_1, \dots, a_k\}$.

- (b) Die Menge *Exp* aller *Ausdrücke* e von \mathcal{L}_c erhält man durch folgende Erweiterung der kontextfreien Grammatik von \mathcal{L}_o .

$$e ::= \text{class } (self) b \text{ end} \\ | \text{new } e_1$$

TODO: Damit ist klar, dass $Row \subseteq Body$, d.h. wenn etwas für alle $b \in Body$ gezeigt wird, ist damit auch für alle $r \in Row$ gezeigt. Sämtliche **inherit**-Anweisungen stehen stets am Anfang einer Klasse (vor der Reihe).

5.2 Operationelle Semantik der Sprache \mathcal{L}_c

TODO: Prosa.

Definition 5.2 (Werte und Reihenwerte) Die Menge $Val \subseteq Exp$ aller *Werte* v von \mathcal{L}_c erhält man, indem man die Produktion

$$v ::= \text{class } (self) r \text{ end}$$

zur kontextfreien Grammatik von \mathcal{L}_o (vgl. Definition 2.4) hinzunimmt.

5.2.1 Frei vorkommende Namen und Substitution

Definition 5.3 (Frei vorkommende Namen)

- (a) Die Menge $free(e)$ aller im Ausdruck $e \in Exp$ frei vorkommenden Namen erhält man durch folgende Verallgemeinerung von Definition 2.5.

$$\begin{aligned} free(\mathbf{class}(self) \ b \ \mathbf{end}) &= free(b) \setminus \{self\} \\ free(\mathbf{new} \ e) &= free(e) \end{aligned}$$

- (b) Die Menge $free(b)$ aller im Klassenrumpf $b \in Body$ frei vorkommenden Namen ist wie folgt definiert:

$$free(\mathbf{inherit} \ A \ \mathbf{from} \ e; \ b) = \{self\} \cup free(e) \cup free(b) \setminus A$$

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 Klassennamen/Supernamen, würde aber die Betrachtung deutlich aufwendiger machen).

Die Menge Exp^* 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 (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 wie folgt erweitern.

Definition 5.4 (Substitution)

- (a) Für $e' \in Exp$, $e \in Exp^*$ und $id \in Id$ erhält man den Ausdruck $e'[e/id]$, der aus e' durch *Substitution* von e für id entsteht, durch folgende Erweiterung von Definition 2.8:

$$\begin{aligned} \mathbf{class}(self : \tau) \ b \ \mathbf{end}[e/id] &= \begin{cases} \mathbf{class}(self) \ b \ \mathbf{end} & \text{falls } id = self \\ \mathbf{class}(self) \ b[e/id] \ \mathbf{end} & \text{sonst} \end{cases} \\ (\mathbf{new} \ e')[e/id] &= \mathbf{new} \ e'[e/id] \end{aligned}$$

- (b) Für $b \in Body$, $e \in Exp^*$ und $id \in Id$ erhält man den Klassenrumpf $b[e/id]$, der aus b durch *Substitution* von e für id entsteht, durch folgende Verallgemeinerung von Definition 2.8:

$$(\mathbf{inherit} \ A \ \mathbf{from} \ e'; \ b)[e/id] = \begin{cases} \mathbf{inherit} \ A \ \mathbf{from} \ e'[e/id]; \ b & \text{falls } id \in A \\ \mathbf{inherit} \ A \ \mathbf{from} \ e'[e/id]; \ b[e/id] & \text{sonst} \end{cases}$$

5.2.2 Small step Semantik

TODO: Einleitung

Definition 5.5 (Gültige small steps für \mathcal{L}_c) Ein small step, $e \rightarrow_e e'$ mit $e, e' \in \text{Exp}$, $b \rightarrow_b b'$ mit $b, b' \in \text{Body}$ oder $r \rightarrow_r r'$ mit $r, r' \in \text{Row}$, 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

$$\begin{aligned}
 (\text{CLASS-EVAL}) \quad & \frac{b \rightarrow_b b'}{\text{class}(\text{self}) \ b \ \text{end} \rightarrow_e \text{class}(\text{self}) \ b' \ \text{end}} \\
 (\text{NEW-EVAL}) \quad & \frac{e \rightarrow_e e'}{\text{new } e \rightarrow_e \text{new } e'} \\
 (\text{NEW-EXEC}) \quad & \text{new}(\text{class}(\text{self}) \ r \ \text{end}) \rightarrow_e \text{object}(\text{self}) \ r \ \text{end}
 \end{aligned}$$

und den neuen small step Regeln für Klassenrumpfe

$$\begin{aligned}
 (\text{INHERIT-RIGHT}) \quad & \frac{b \rightarrow_b b'}{\text{inherit } A \ \text{from } e; \ b \rightarrow_b \text{inherit } A \ \text{from } e; \ b'} \\
 (\text{INHERIT-LEFT}) \quad & \frac{e \rightarrow_e e'}{\text{inherit } A \ \text{from } e; \ r \rightarrow_b \text{inherit } A \ \text{from } e'; \ r} \\
 (\text{INHERIT-EXEC}) \quad & \text{inherit } A \ \text{from} \ (\text{class}(\text{self}) \ r_1 \ \text{end}); \ r_2 \rightarrow_b r_1 \oplus r_2 \quad \text{falls } \text{dom}_a(r_1) = A
 \end{aligned}$$

herleiten lässt.

TODO: Erläuterung der small step Regeln.

Für die erweiterte small step Semantik wollen wir nun wieder zeigen, dass sie eindeutig ist, das heisst, für jeden Ausdruck, jede Reihe und jeden Klassenrumpf höchstens ein small step existiert. Dazu zeigen wir zunächst wieder, dass für Werte und Reihenwerte keine small steps existieren.

Lemma 5.1

- (a) $v \not\rightarrow$ für alle $v \in \text{Val}$.
- (b) $\omega \not\rightarrow$ für alle $\omega \in \text{RVal}$.

Beweis: Für die Werte der Sprache \mathcal{L}_o wurde diese Eigenschaft bereits bewiesen (Lemma 2.3), und es ist ziemlich offensichtlich, dass keine der neuen Regeln einen small step

für einen dieser Werte ermöglicht. Damit genügt es zu zeigen, dass auch für die einzige neue Form eines Wertes, nämlich $\mathbf{class}(self) \ r \ \mathbf{end} \in Val$, kein small step existiert.

Für $v = \mathbf{class}(self) \ r \ \mathbf{end}$ kommt nur ein small step mit der neuen Regel (CLASS-EVAL) in Frage. (CLASS-EVAL) setzt jedoch voraus, dass für r ein small step mit der Relation \rightarrow_b existiert. Diese wiederum beschränkt sich aber auf Klassenrümpfe, so dass folglich hier kein small step in Frage kommt für r . Also existiert auch für $\mathbf{class}(self) \ r \ \mathbf{end}$ kein small step. \square

TODO: lala

Satz 5.1 (Eindeutigkeit des Übergangsschritts)

- (a) Für jeden Ausdruck $e \in Exp$ existiert höchstens ein $e' \in Exp$ mit $e \rightarrow_e e'$.
- (b) Für jede Reihe $r \in Row$ existiert höchstens ein $r' \in Row$ mit $r \rightarrow_r r'$.
- (c) Für jeden Klassenrumpf $b \in Body$ existiert höchstens ein $b' \in Body$ mit $b \rightarrow_b b'$.

Beweis: Wie für die Programmiersprache \mathcal{L}_o beweisen wir den Satz durch simultane Induktion über die Grösse von Ausdrücken e , Reihen r und Klassenrümpfen b , mit Fallunterscheidung nach der jeweiligen Form von e , r oder b .

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

Wegen Lemma 5.1 existiert in diesen Fällen kein small step für e .

- 2.) **TODO**

Die übrigen Fälle verlaufen analog. \square

TODO: Prosa

5.3 Typsystem

TODO: Einleitung

5.3.1 Syntax der Sprache \mathcal{L}_c^t

TODO: Erläuterung zu den Ideen.

Definition 5.6 (Typen)

- (a) Die Menge *Type* aller Typen τ von \mathcal{L}_c^t ist durch folgende Erweiterung der kontextfreien Grammatik von \mathcal{L}_o^t (Definition 2.14) definiert:

$$\tau ::= \zeta(\tau_1 : \phi)$$

- (b) Die Menge *RType* aller Reihentypen ϕ von \mathcal{L}_c^t definieren wir durch Hinzunahme der Produktion

$$\phi ::= a : \tau; \phi_1$$

zur kontextfreien Grammatik der Programmiersprache \mathcal{L}_o^t (Definition 2.14), wobei wie für Methodennamen, auch für Attributnamen gilt, dass alle Vorkommen in einem Reihentyp paarweise verschieden sein müssen.

- (c) Desweiteren seien Funktionen dom_a und dom_m auf *RType* definiert, die jeweils die in einem Reihentyp vorhandenen Attribut- oder Methodennamen auflisten, sowie eine Funktion *methods*, die aus einem Reihentyp alle Typinformationen über Attribute entfernt.

Ein Reihentyp enthält nun neben Typinformationen über die Methoden in einer Reihe auch Typinformationen über die Attribute in einer Reihe. Desweiteren werden Reihentypen in \mathcal{L}_c^t nicht nur als Typen für Reihen, sondern ebenfalls als Typen für Klassenrümpfe verwendet. Hintergrund dieser Änderung ist die Feststellung, dass im Rahmen von Vererbung, die Typen von Klassen, im Gegensatz zu den Objekttypen, auch Informationen über die in ihnen vorhandenen Attribute aufweisen müssen.

Die Vereinigung von Reihentypen, geschrieben als $\phi_1 \oplus \phi_2$, ist wie zuvor definiert (Definition 2.15), wobei für Reihentypen nun auch die Typen gemeinsamer Attribute übereinstimmen müssen. Insbesondere gilt nach wie vor, dass die Vereinigung zweier inkompatibler Reihentypen nicht definiert ist, also in einer Implementation des Typsystems zu einem Typfehler führt.

5.3.2 Typsystem der Sprache \mathcal{L}_c^t

TODO: Prosa, Erweiterung von Definition 4.2

Definition 5.7

- (a) Die Menge *SType* \subseteq *Type* aller *einfachen Typen* τ_s ist wie definiert als Einschränkung der Menge *Type* aller Typen auf die primitiven Typen und die Klassentypen, also:

$$SType = \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \cup \{\zeta(\tau : \phi) \mid \tau \in Type, \phi \in RType\}$$

- (b) Die Menge $TCons$ der Typkonstruktoren wird neu definiert durch

$$\begin{aligned} TCons &= SType \cup \{\rightarrow\} \\ &\cup \{ \langle a_1; \dots; a_k; m_1; \dots; m_n \rangle \mid k, n \in \mathbb{N}, \\ &\quad a_1, \dots, a_k \in Attribute, m_1, \dots, m_n \in Method \}, \end{aligned}$$

wobei wie zuvor die Attribut- und Methodennamen in $\langle a_1; \dots; a_k; m_1; \dots; m_n \rangle$ als disjunkt angenommen werden.

Die Idee der einfachen Typen $\tau_s \in SType$ ist, dass auf diesen Typen kein Subtyping definiert ist. Für die primitiven Datentypen war das bisher stets der Fall. Neu hinzukommt, dass auch auf Klassentypen kein Subtyping möglich ist. Der Grund dafür wird später im Beweis der Typsicherheit ersichtlich. Intuitiv sollte man sich einfach vorstellen, dass jegliche in Klassentypen enthaltenen Typinformationen immer erhalten bleiben müssen. Es sollte anhand der Ergebnisse aus den zuvor betrachteten Programmiersprachen klar sein, dass für eine objekt-orientierte Programmiersprache Subtyping auf Objekttypen ausreichend ist.

Da die Mengen $Type$ und $TCons$ erweitert worden sind, müssen auch die Funktionen $root$, $arity$ und $child$ aus Definition 4.3 entsprechend erweitert werden.

Definition 5.8

- (a) Die Funktion $root : Type \rightarrow TCons$ wird wie folgt erweitert:

$$\begin{aligned} root(\langle a_1 : \hat{\tau}_1; \dots; a_k : \hat{\tau}_k; m_1 : \tau_1; \dots; m_n : \tau_n \rangle) &= \langle a_1; \dots; a_k; \dots; m_1; \dots; m_n \rangle \\ root(\zeta(\tau : \phi)) &= \zeta(\tau : \phi) \end{aligned}$$

- (b) Die Funktion $arity : Type \rightarrow \mathbb{N}$ wird entsprechend erweitert durch:

$$\begin{aligned} arity(\langle a_1 : \hat{\tau}_1; \dots; a_k : \hat{\tau}_k; m_1 : \tau_1; \dots; m_n : \tau_n \rangle) &= k + n \\ arity(\zeta(\tau : \phi)) &= 0 \end{aligned}$$

- (c) Die partielle Funktion $child : Type \times \mathbb{N} \rightharpoonup Type$ wird erweitert durch:

$$child_i(\langle a_1 : \hat{\tau}_1; \dots; a_k : \hat{\tau}_k; m_1 : \tau_1; \dots; m_n : \tau_n \rangle) = \begin{cases} \hat{\tau}_i & \text{falls } 1 \leq i \leq k \\ \tau_{k+i} & \text{falls } k < i \leq k + n \end{cases}$$

TODO: Prosa

Definition 5.9 Die Relationen \sim_n wird wieder wie folgt induktiv definiert

$$\begin{aligned}\sim_0 &= Type^2 \\ \sim_{n+1} &= \{(\tau, \tau') \in Type^2 \mid \text{root}(\tau) = \text{root}(\tau') \\ &\quad \wedge \forall 1 \leq i \leq \text{arity}(\tau) : \text{child}_i(\tau) \sim_n \text{child}_i(\tau')\},\end{aligned}$$

und die Relation \sim wieder durch

$$\sim = \bigcap_{n \in \mathbb{N}} \sim_n$$

als Durchschnitt aller Relationen \sim_n .

Wie zuvor lässt sich zeigen, dass die Relation \sim eine Äquivalenzrelation auf $Type$ ist, und ebenfalls wie zuvor kann man zeigen, dass eine entsprechende generierende Abbildung von $TypeRel$ nach $TypeRel$ existiert, die \sim als kleinsten Fixpunkt besitzt. Entsprechend gilt das folgende Lemma:

Lemma 5.2 (Äquivalenz-Lemma für \mathcal{L}_c^t) $\tau \sim \hat{\tau}$ gilt genau dann, wenn eine der folgenden Aussagen zutrifft:

- (a) $\tau = \mu t_1 \dots \mu t_n. \tau_s$ und $\hat{\tau} = \mu \hat{t}_1 \dots \mu \hat{t}_m. \tau_s$ mit $\tau_s \in SType$
- (b) $\tau = \mu t_1 \dots \mu t_n. \tau_1 \rightarrow \tau_2$ und $\hat{\tau} = \mu \hat{t}_1 \dots \mu \hat{t}_m. \hat{\tau}_1 \rightarrow \hat{\tau}_2$ mit $\tau_1[\tau/t_i]_{i=1}^n \sim \hat{\tau}_1[\hat{\tau}/\hat{t}_i]_{i=1}^m$ und $\tau_2[\tau/t_i]_{i=1}^n \sim \hat{\tau}_2[\hat{\tau}/\hat{t}_i]_{i=1}^m$
- (c) $\tau = \mu t_1 \dots \mu t_n. \langle a_1 : \tau_1; \dots; a_h : \tau_h; m_{h+1} : \tau_{h+1}; \dots; m_k : \tau_k \rangle$ und $\hat{\tau} = \mu \hat{t}_1 \dots \mu \hat{t}_m. \langle a_1 : \hat{\tau}_1; \dots; a_h : \hat{\tau}_h; m_{h+1} : \hat{\tau}_{h+1}; \dots; m_k : \hat{\tau}_k \rangle$ mit $\tau_j[\tau/t_i]_{i=1}^n \sim \hat{\tau}_j[\hat{\tau}/\hat{t}_i]_{i=1}^m$ für alle $j = 1, \dots, k$

Beweis: Lässt sich analog zur Äquivalenz-Lemma für die Programmiersprache \mathcal{L}_o^{rt} (4.4) beweisen, indem man eine entsprechende generierende Funktion angibt, und mit dem Fixpunktsatz von Kleene (1.1) zeigt, dass die Relation \sim ein Fixpunkt dieser Funktion ist. \square

Weiterhin definieren wir auch für die Programmiersprache \mathcal{L}_c^t auf der Menge $Type$ eine Subtyprelation \lesssim .

Definition 5.10 Die Relationen \lesssim_n werden wie zuvor induktiv durch

$$\begin{aligned}\lesssim_0 &= Type^2 \\ \lesssim_{n+1} &= \{(\tau, \tau') \in Type^2 \mid \text{root}(\tau) = \text{root}(\tau') \in SType\} \\ &\cup \{(\tau, \tau') \in Type^2 \mid \text{root}(\tau) = \text{root}(\tau') = \rightarrow \\ &\quad \wedge \text{child}_1(\tau') \lesssim_n \text{child}_1(\tau) \wedge \text{child}_2(\tau) \lesssim_n \text{child}_2(\tau')\} \\ &\cup \{(\tau, \tau') \in Type^2 \mid \text{root}(\tau) = \langle a_1; \dots; a_h; m_{h+1}; \dots; m_{k+l} \rangle \\ &\quad \wedge \text{root}(\tau') = \langle a_1; \dots; a_h; m_{h+1}; \dots; m_k \rangle \\ &\quad \wedge \forall 1 \leq i \leq k : \text{child}_i(\tau) \lesssim_n \text{child}_i(\tau')\}\end{aligned}$$

definiert, und die Relation \lesssim ist wie zuvor durch

$$\lesssim = \bigcap_{n \in \mathbb{N}} \lesssim_n$$

als Durchschnitt aller Relationen \lesssim_n definiert.

Analog zur Subtyping-Relation der Programmiersprache \mathcal{L}_o^{srt} (Definition 4.8) lässt sich auch hier zeigen, dass die Relation \lesssim eine Quasiordnung ist, welche antisymmetrisch bezüglich der Relation \sim ist.

Lemma 5.3

- (a) Die Relation \lesssim ist eine Quasiordnung.
- (b) $\forall \tau, \tau' \in \text{Type} : \tau \lesssim \tau' \wedge \tau' \lesssim \tau \Leftrightarrow \tau \sim \tau'$

Beweis: Wie zuvor in den Beweisen der Lemmata für die Programmiersprache \mathcal{L}_o^{srt} (Lemma 4.11 und 4.12). \square

TODO: Prosa

Lemma 5.4 (Subtyping-Lemma für \mathcal{L}_c^t) $\tau \lesssim \tau'$ gilt genau dann, wenn eine der folgenden Aussagen zutrifft:

- (a) $\tau \sim \tau' \sim \tau_s$ mit $\tau_s \in S\text{Type}$
- (b) $\tau \sim \tau_1 \rightarrow \tau_2$ und $\tau' \sim \tau'_1 \rightarrow \tau'_2$ mit $\tau'_1 \lesssim \tau_1$ und $\tau_2 \lesssim \tau'_2$
- (c) $\tau \sim \langle a_1 : \tau_1; \dots; a_n : \tau_n; m_{n+1} : \tau_{n+1}; \dots; m_{k+l} : \tau_{k+l} \rangle$
und $\tau' \sim \langle a_1 : \tau'_1; \dots; a_n : \tau'_n; m_{n+1} : \tau'_{n+1}; \dots; m_k : \tau'_k \rangle$
mit $\tau_i \lesssim \tau'_i$ für alle $i = 1, \dots, k$

Beweis: Erfolgt wie im Fall des Subtyping-Lemmas für die Programmiersprache \mathcal{L}_o^{srt} (4.15), indem eine dem Lemma entsprechende generierende Funktion auf TypeRel konstruiert wird, und anschliessend mit den Fixpunktsatz von Kleene (1.1) gezeigt wird, dass die Relation \lesssim ein Fixpunkt dieser Funktion ist. \square

Nach dem wir nun die Relationen \sim und \lesssim für die Programmiersprache \mathcal{L}_c^t passend erweitert haben und die notwendigen Aussagen gezeigt haben, die einen einfachen Umgang mit diesen Relationen ermöglichen, können wir nun Typregeln aufstellen, die festlegen, wie gültige Typurteile für Programme der Sprache \mathcal{L}_c^t hergeleitet werden können:

Definition 5.11 (Gültige Typurteile für \mathcal{L}_c^t) Ein Typurteil $\Gamma \triangleright_e e :: \tau$ oder $\Gamma \triangleright_r b :: \phi$ heisst *gültig* für \mathcal{L}_c^t , wenn es sich mit den Typregeln für die funktionale Kernsprache aus Definition 2.20, sowie den Typregeln für Objekte und Klassen

$$\begin{array}{ll}
(\text{SEND}) & \frac{\Gamma \triangleright_e e :: \langle m : \tau; \phi \rangle}{\Gamma \triangleright_e e \# m :: \tau} \\
(\text{SEND}') & \frac{\Gamma \triangleright_r \omega :: (m : \tau; \phi)}{\Gamma \triangleright_e \omega \# m :: \tau} \\
(\text{OBJECT''}) & \frac{\Gamma^*[\tau / \text{self}] \triangleright_r r :: \phi \quad \tau \sim \langle \text{methods}(\phi) \rangle}{\Gamma \triangleright_e \mathbf{object}(\text{self} : \tau) r \mathbf{end} :: \tau} \\
(\text{DUPL}) & \frac{\Gamma \triangleright_e \text{self} :: \tau \quad \forall i = 1 \dots n : \Gamma \triangleright_e a_i :: \tau_i \wedge \Gamma \triangleright_e e_i :: \tau_i}{\Gamma \triangleright_e \{ \langle a_1 = e_1; \dots; a_n = e_n \rangle \} :: \tau} \\
(\text{CLASS}) & \frac{\Gamma^*[\tau / \text{self}] \triangleright_r b :: \phi \quad \tau \lesssim \langle \text{methods}(\phi) \rangle}{\Gamma \triangleright_e \mathbf{class}(\text{self} : \tau) b \mathbf{end} :: \zeta(\tau : \phi)} \\
(\text{NEW}) & \frac{\Gamma \triangleright_e e :: \zeta(\tau : \phi) \quad \tau \sim \langle \text{methods}(\phi) \rangle}{\Gamma \triangleright_e \mathbf{new} e :: \tau} \\
(\text{SUBSUME'}) & \frac{\Gamma \triangleright_e e :: \tau' \quad \tau' \lesssim \tau}{\Gamma \triangleright_e e :: \tau}
\end{array}$$

und den verallgemeinerten Typregeln für Reihen und Klassenrumpfe

$$\begin{array}{ll}
(\text{EMPTY}) & \Gamma \triangleright_r \epsilon :: \emptyset \\
(\text{ATTR'}) & \frac{\Gamma^* \triangleright_e e :: \tau \quad \Gamma[\tau / a] \triangleright_r r_1 :: \phi}{\Gamma \triangleright_r \mathbf{val} a = e; r_1 :: (a : \tau; \emptyset) \oplus \phi} \\
(\text{METHOD}) & \frac{\Gamma \triangleright_e e :: \tau \quad \Gamma \triangleright_r r_1 :: \phi}{\Gamma \triangleright_r \mathbf{method} m = e; r_1 :: (m : \tau; \emptyset) \oplus \phi} \\
(\text{INHERIT}) & \frac{\Gamma \triangleright_e \text{self} :: \tau \quad \Gamma^* \triangleright_e e :: \zeta(\tau : \phi) \quad \text{dom}_a(\phi) = A \quad \Gamma[\phi(a) / a]_{a \in A} \triangleright_r b :: \phi'}{\Gamma \triangleright_r \mathbf{inherit} A \mathbf{from} e; b :: \phi \oplus \phi'}
\end{array}$$

herleiten lässt. Wie zuvor gilt, dass die spezielle Typregel (SEND') lediglich für den Beweis der Typsicherheit von \mathcal{L}_c^t benötigt wird, also in einem möglichen Typechecker nicht implementiert werden muss.

5.3.3 Typsicherheit

TODO: *Val* mit Typen versehen, ebenso die small step Regeln auf die neue Syntax bringen.

TODO: Die üblichen Lemmata

Lemma 5.5 (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) $\forall \Gamma \in TEnv, b \in Body, \phi \in RType : \Gamma \triangleright b :: \phi \Rightarrow free(b) \subseteq dom(\Gamma)$

Beweis: Wie im Beweis des Lemmas für die Programmiersprache \mathcal{L}_o^{srt} (4.16) erfolgt der Beweis durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma \triangleright e :: \tau$ und $\Gamma \triangleright b :: \phi$, mit Fallunterscheidung nach der letzten Typregel in der Herleitung. Wir betrachten lediglich die neuen Fälle für die Programmiersprache \mathcal{L}_c^t :

- 1.) $\Gamma \triangleright \mathbf{new} e :: \tau$ mit Typregel (NEW)

Für diesen Fall folgt die Behauptung mit Induktionsvoraussetzung unmittelbar aus der ersten Prämisse der (NEW)-Regel.

- 2.) $\Gamma \triangleright \mathbf{class} (self : \tau) b \mathbf{end} :: \zeta(\tau : \phi)$ mit Typregel (CLASS)

Dann gilt nach Voraussetzung $\Gamma^*[\tau/self] \triangleright b :: \phi$ mit $\tau \lesssim \langle methods(\phi) \rangle$. Nach Induktionsvoraussetzung gilt dann

$$free(b) \subseteq dom(\Gamma^*[\tau/self]),$$

und wegen

$$dom(\Gamma^*[\tau/self]) = dom(\Gamma^*) \cup \{self\} \subseteq dom(\Gamma) \cup \{self\}$$

folgt nach den Gesetzen der Mengenlehre, dass

$$free(b) \setminus \{self\} \subseteq dom(\Gamma)$$

gilt. Die Behauptung folgt dann trivialerweise nach Definition 5.3 (Frei vorkommende Namen).

- 3.) $\Gamma \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \phi$ mit Typregel (OBJECT'')

Die veränderte Prämisse von (OBJECT'') im Vergleich zu (OBJECT') hat, wie man sich leicht klar machen kann, keinen Einfluss auf den Ablauf des Beweises für diesen Fall, so dass dieser Fall vollkommen identisch zu den vorangegangenen Beweisen dieses Lemmas verläuft.

4.) $\Gamma \triangleright \mathbf{val} \ a = e; r :: \phi$ mit Typregel (ATTR')

Wie im Fall der (OBJECT'')-Regel hat die Änderung der (ATTR')-Regel keinen Einfluss auf den Ablauf dieses Beweises.

5.) $\Gamma \triangleright \mathbf{inherit} \ A \ \mathbf{from} \ e; b :: \phi \oplus \phi'$ mit Typregel (INHERIT)

Dieses Typurteil kann nur aus Prämissen der Form $\Gamma \triangleright self :: \tau$, $\Gamma^* \triangleright e :: \zeta(\tau : \phi)$, $dom_a(\phi) = A$ und $\Gamma[\phi^{(a)}/a]_{a \in A} \triangleright b :: \phi'$ folgen. Das bedeutet, mit Induktionsvoraussetzung folgt

$$\begin{aligned} free(self) &\subseteq dom(\Gamma), \\ free(e) &\subseteq dom(\Gamma^*) \subseteq dom(\Gamma) \quad \text{und} \\ free(b) &\subseteq dom(\Gamma[\phi^{(a)}/a]_{a \in A}) \subseteq dom(\Gamma) \cup A. \end{aligned}$$

Letzteres lässt sich nach den Gesetzen der Mengenlehre umformen zu

$$free(b) \setminus A \subseteq dom(\Gamma),$$

also erhalten wir insgesamt

$$\{self\} \cup free(e) \cup [free(b) \setminus A] \subseteq dom(\Gamma),$$

was nach Definition 5.3 (Frei vorkommende Namen) genau der Behauptung entspricht.

Die übrigen Fälle folgen wie zuvor für die Programmiersprache \mathcal{L}_o^{srt} . \square

TODO: Koinzidenzlemma

Lemma 5.6 (Koinzidenzlemma für \mathcal{L}_c^t)

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

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

Beweis: Den Beweis führen wir wie bereits für die vorangegangenen Programmiersprachen durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma_1 \triangleright e :: \tau$ und $\Gamma_1 \triangleright b :: \phi$, mit Fallunterscheidung nach der zuletzt angewandten Typregel. Wir betrachten wieder nur die Fälle für die neuen oder geänderten Typregeln:

- 1.) $\Gamma_1 \triangleright \mathbf{new} e :: \tau$ mit Typregel (NEW)

Dann gilt nach Voraussetzung $\Gamma_1 \triangleright e :: \zeta(\tau : \phi)$ und $\tau \sim \langle \mathit{methods}(\phi) \rangle$. Nach Induktionsvoraussetzung folgt daraus

$$\Gamma_2 \triangleright e :: \zeta(\tau : \phi),$$

und damit also die Behauptung mit Typregel (NEW).

- 2.) $\Gamma_1 \triangleright \mathbf{class}(\mathit{self} : \tau) b \mathbf{end} :: \zeta(\tau : \phi)$ mit Typregel (CLASS)

Das kann nur aus Prämissen der Form $\Gamma_1^*[\tau/\mathit{self}] \triangleright b :: \phi$ und $\tau \lesssim \langle \mathit{methods}(\phi) \rangle$ folgen. Gemäß Lemma 2.4 gilt $\Gamma_1^* =_{\text{free}(e)} \Gamma_2^*$ und mit erneuter Anwendung des Lemmas folgt $\Gamma_1^*[\tau/\mathit{self}] =_{\text{free}(e)} \Gamma_2^*[\tau/\mathit{self}]$. Da $\text{free}(b) \subseteq \text{free}(e)$ folgt schliesslich, wieder mit Lemma 2.4, dass auch $\Gamma_1^*[\tau/\mathit{self}] =_{\text{free}(b)} \Gamma_2^*[\tau/\mathit{self}]$ gilt. Damit können wir nun also die Induktionsvoraussetzung anwenden und erhalten

$$\Gamma_2^*[\tau/\mathit{self}] \triangleright b :: \phi,$$

woraus wiederum wegen $\tau \lesssim \langle \mathit{methods}(\phi) \rangle$ mit Typregel (CLASS) unmittelbar die Behauptung folgt.

- 3.) $\Gamma_1 \triangleright \mathbf{object}(\mathit{self} : \tau) r \mathbf{end} :: \tau$ mit Typregel (OBJECT')

Wie man sich leicht überlegen kann, folgt dieser Fall völlig analog zum vorhergehenden Fall der (CLASS)-Regel.

- 4.) $\Gamma_1 \triangleright \mathbf{val} a = e; r :: (a : \tau; \emptyset) \oplus \phi$ mit Typregel (ATTR')

Hier ist offensichtlich, dass der Fall identisch zum Beweis des Lemmas für die vorangegangenen Programmiersprachen abläuft, da sich die Veränderungen von (ATTR) zu (ATTR') auf den resultierenden Typ beschränken.

- 5.) $\Gamma_1 \triangleright \mathbf{inherit} A \mathbf{from} e; b :: \phi \oplus \phi'$ mit Typregel (INHERIT)

Hier muss nach Voraussetzung $\Gamma_1 \triangleright \mathit{self} :: \tau$, $\Gamma_1^* \triangleright e :: \zeta(\tau : \phi)$, $\text{dom}_a(\phi) = A$ und $\Gamma_1^{[\phi(a)/a]}_{a \in A} \triangleright b :: \phi'$ gelten. Auf die ersten beiden Prämissen lässt sich unmittelbar die Induktionsvoraussetzung anwenden, und wir erhalten

$$\Gamma_2 \triangleright \mathit{self} :: \tau$$

und

$$\Gamma_2^* \triangleright e :: \zeta(\tau : \phi).$$

Wegen Lemma 2.4 gilt

$$\Gamma_1[\phi^{(a)}/a]_{a \in A} =_{free(b)} \Gamma_2[\phi^{(a)}/a]_{a \in A},$$

denn, ebenfalls wegen Lemma 2.4, gilt $free(b) \subseteq free(e)$, also $\Gamma_1 =_{free(b)} \Gamma_2$. Das bedeutet, wir können nun auch für letzte Prämisse der (INHERIT)-Regel die Induktionsvoraussetzung anwenden, und erhalten

$$\Gamma_2[\phi^{(a)}/a]_{a \in A} \triangleright b :: \phi'.$$

Aus diesen Ergebnissen folgt dann mit Typregel (INHERIT), dass auch

$$\Gamma_2 \triangleright \mathbf{inherit} \ A \ \mathbf{from} \ e; \ b :: \phi \oplus \phi'$$

gilt, was zu zeigen war.

Die übrigen Fälle folgen wie in den Beweisen des Lemmas für die vorangegangenen Programmiersprachen (vgl. Lemma 2.6, 3.5, 4.6 und 4.17). \square

TODO: Prosa

Lemma 5.7 (Typurteile und Substitution) *Sei $id \in Id \setminus Self$, $\Gamma \in TEnv$, $\tau \in Type$ und $e \in Exp$. Dann gilt:*

$$(a) \ \forall e' \in Exp : \forall \tau' \in Type : \Gamma[\tau/id] \triangleright e' :: \tau' \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright e'[e/id] :: \tau'$$

$$(b) \ \forall b \in Body : \forall \phi \in RType : \Gamma[\tau/id] \triangleright b :: \phi \wedge \Gamma^* \triangleright e :: \tau \Rightarrow \Gamma \triangleright b[e/id] :: \phi$$

Beweis: Wie in den Beweisen des Lemmas für die vorangegangenen Sprachen, zeigen wir die Behauptung wieder durch simultane Induktion über die Länge der Typherleitungen $\Gamma[\tau/id] \triangleright e' :: \tau'$ und $\Gamma[\tau/id] \triangleright b :: \phi$, wobei wir jeweils nach der zuletzt angewandten Typregel unterscheiden. Wie zuvor betrachten wir lediglich die Fälle für die neuen oder geänderten Typregeln:

- 1.) $\Gamma[\tau/id] \triangleright \mathbf{new} \ e' :: \tau'$ mit Typregel (NEW)

Dann gilt nach Voraussetzung $\Gamma[\tau/id] \triangleright e' :: \zeta(\tau' : \phi')$ mit $\tau' \sim \langle methods(\phi') \rangle$. Daraus folgt unmittelbar die Behauptung mit Induktionsvoraussetzung, Typregel (NEW) und Definition 5.4.

2.) $\Gamma[\tau/id] \triangleright \mathbf{class} (self : \tau') b \mathbf{end} :: \zeta(\tau' : \phi')$ mit Typregel (CLASS)

Das kann nur aus Prämissen der Form $(\Gamma[\tau/id])^*[\tau'/self] \triangleright b :: \phi'$ und $\tau' \lesssim \langle methods(\phi') \rangle$ folgen. Abhängig von der Form von id folgt mit Lemma 2.4 und Lemma 5.6, dass

$$\Gamma^*[\tau/id][\tau'/self] \triangleright b :: \phi'$$

gilt, und da $id \neq self$ folgt weiter

$$\Gamma^*[\tau'/self][\tau/id] \triangleright b :: \phi'.$$

Wegen $(\Gamma^*[\tau'/self])^* = \Gamma^*$ lässt sich nun die Induktionsvoraussetzung anwenden, und wir erhalten

$$\Gamma^*[\tau'/self] \triangleright b[e/id] :: \phi'.$$

Die Behauptung folgt dann mit Typregel (CLASS) und Definition 5.4.

3.) $\Gamma[\tau/id] \triangleright \mathbf{object} (self : \tau') r \mathbf{end} :: \tau'$ mit Typregel (OBJECT)

Folgt offensichtlich analog zum Fall der (CLASS)-Regel.

4.) $\Gamma[\tau/id] \triangleright \mathbf{val} a = e'; r :: (a : \tau'; \emptyset) \oplus \phi$ mit (ATTR')

Dann gilt $(\Gamma[\tau/id])^* \triangleright e' :: \tau'$ und $\Gamma[\tau/id][\tau'/a] \triangleright r :: \phi$ nach Voraussetzung.

Für die erste Prämisse folgt abhängig von der konkreten Form von id wegen Lemma 2.4 und dem Koinzidenzlemma (5.6), dass auch

$$\Gamma^*[\tau/id] \triangleright e' :: \tau'$$

gilt. Da $(\Gamma^*)^* = \Gamma^*$, lässt sich hierauf dann die Induktionsvoraussetzung anwenden, so dass wir

$$\Gamma^* \triangleright e'[e/id] :: \tau'$$

erhalten.

Für die zweite Prämisse müssen wir zwei Fälle unterscheiden:

1.) $id \neq a$

Dann gilt $\Gamma[\tau/id][\tau'/a] = \Gamma[\tau'/a][\tau/id]$ und die Behauptung folgt mit Induktionsvoraussetzung und Anwendung von Typregel (ATTR') auf die beiden Zwischenergebnisse der Prämissen.

2.) $id = a$

In diesem Fall gilt $\Gamma[\tau/id][\tau'/a] = \Gamma[\tau'/a]$. Nach Definition der Substitution (5.4) gilt für Attributdeklarationen

$$(\mathbf{val} \ a = e'; r)[e/a] = (\mathbf{val} \ a = e'[e/a]; r),$$

also sind offensichtlich bereits alle Voraussetzung erfüllt, um mit Typregel (ATTR') die Behauptung herzuleiten.

Insgesamt haben wir also gezeigt, dass unter den gegebenen Voraussetzungen auch

$$\Gamma \triangleright (\mathbf{val} \ a = e'; r)[e/id] :: (a : \tau'; \emptyset) \oplus \phi$$

gilt.

5.) $\Gamma[\tau/id] \triangleright \mathbf{inherit} \ A \ \mathbf{from} \ e'; b :: \phi \oplus \phi'$ mit Typregel (INHERIT)

Dieses Typurteil kann nur aus Voraussetzungen der Form $\Gamma[\tau/id] \triangleright self :: \tau'$, $(\Gamma[\tau/id])^* \triangleright e' :: \zeta(\tau' : \phi)$, $dom_a(\phi) = A$ und $\Gamma[\tau/id][\phi^{(a)}/a]_{a \in A} \triangleright b :: \phi'$ folgen.

TODO

Die restlichen Fälle verlaufen identisch zu den Beweisen dieses Lemmas für die zuvor betrachteten Programmiersprachen (vgl. Lemma 2.7, 3.6, 4.7 und 4.18). \square

Es sollte den Leser nur sehr bedingt überraschen, dass wir auch für die Programmiersprache \mathcal{L}_c^t neben dem allgemeinen Substitutionslemma, ein spezielles Lemma für die Substitution von Ausdrücken für *self* benötigen. Nach wie vor beschränkt sich die Nutzung dieses Lemmas auf das Auffalten von Objekten, so dass das Lemma und der Beweis im wesentlichen von den vorhergehenden Programmiersprachen übernommen werden können.

Lemma 5.8 (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 $\tau' \in Type$:

$$(1) \ \Gamma[\tau/self] \triangleright e :: \tau'$$

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

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

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

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

- (1) $\Gamma[\tau/\text{self}] \triangleright r' :: \phi$
- (2) $\Gamma^* \triangleright \mathbf{object}(\text{self} : \tau) \ r \oplus r' \ \mathbf{end} :: \tau$
- (3) $\forall a \in \text{dom}(\Gamma) \cap \text{Attribute}, \tau_a \in \text{Type} : \Gamma^* \triangleright r(a) :: \tau_a \Leftrightarrow \Gamma \triangleright a :: \tau_a$
 $\Rightarrow \Gamma \triangleright r'[\mathbf{object}(\text{self}:\tau) \ r \oplus r' \ \mathbf{end} / \text{self}] :: \phi$

Insbesondere trifft das Lemma keine Aussage über *self*-Substitution in Klassenrumpfen, stattdessen beschränkt es sich ausschliesslich auf Ausdrücke und Reihen. Die Idee dahinter ist sehr einfach: Klassenrumpfe können nur direkt innerhalb von Klassen vorkommen. *self*-Substitution tritt jedoch nur beim Auffalten von Objekten auf. Zwar kann eine Klasse innerhalb eines Objekts auftreten, jedoch endet nach Definition 5.4 die *self*-Substitution vor der Klasse und wird nicht fortgeführt in den Klassenrumpf.

Beweis: Wie zuvor gilt, dass vermöge Lemma 5.5 die Substitution stets definiert ist. Der Beweis erfolgt dann ebenfalls wie zuvor durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[\tau/\text{self}] \triangleright e :: \tau'$ und $\Gamma[\tau/\text{self}] \triangleright r' :: \phi$, mit Fallunterscheidung nach der zuletzt angewandten Typregel. Wir beschränken uns wieder auf die Fälle der neuen oder geänderten Typregeln:

- 1.) Für (CLASS) und (OBJECT'') ist – wie bereits angedeutet – nichts zu zeigen, da die Behauptung in diesen Fälle unmittelbar mit Definition 5.4 folgt.
- 2.) $\Gamma[\tau/\text{self}] \triangleright \mathbf{new} \ e' :: \tau'$ mit Typregel (NEW)

Dann gilt nach Voraussetzung $\Gamma[\tau/\text{self}] \triangleright e' :: \zeta(\tau' : \phi')$ mit $\tau' \sim \langle \text{methods}(\phi') \rangle$. Darauf lässt sich dann die Induktionsvoraussetzung anwenden, und es folgt

$$\Gamma \triangleright e'[\mathbf{object}(\text{self}:\tau) \ r \ \mathbf{end} / \text{self}] :: \zeta(\tau' : \phi'),$$

woraus dann mit Typregel (NEW) und Definition 5.4 schliesslich die Behauptung folgt.

- 3.) $\Gamma[\tau/\text{self}] \triangleright \mathbf{val} \ a = e'; r'' :: (a : \tau'; \emptyset) \oplus \phi'$ mit Typregel (ATTR')

In diesem Fall muss $(\Gamma[\tau/\text{self}])^* \triangleright e' :: \tau'$ und $\Gamma[\tau/\text{self}][\tau'/a] \triangleright r'' :: \phi'$ gelten.

Da $(\Gamma[\tau/\text{self}])^* = \Gamma^*$ folgt nach Lemma 5.5, dass $\text{self} \notin \text{free}(e')$. Also gilt

$$e' = e'[\mathbf{object}(\text{self}:\tau) \ r \oplus (\mathbf{val} \ a=e'; r'') \ \mathbf{end} / \text{self}],$$

und damit ist für die erste Prämisse der Behauptung nichts weiter zu zeigen.

Für die zweite Prämisse gilt $\Gamma[\tau'/a][\tau/self] \triangleright r'' :: \phi'$, denn $a \neq self$. Da weiter $(\Gamma[\tau'/a])^* = \Gamma^*$, $(\mathbf{val} \ a = e'; r'')(a) = e'$ und $(\Gamma[\tau'/a])(a) = \tau'$ gilt, folgt schliesslich mit Induktionsvoraussetzung, dass auch

$$\Gamma[\tau'/a] \triangleright r''[\mathbf{object} \ (self:\tau) \ r \oplus (\mathbf{val} \ a=e'; r'') \ \mathbf{end} /_{self}] :: \phi'$$

gilt, und somit zusammenfassend, dass mit Typregel (ATTR') und Definition 5.4 die Behauptung folgt.

Die weiteren Fälle verhalten sich ähnlich zu den Beweisen dieses Lemmas für die bereits bekannten Programmiersprachen (vgl. Lemma 2.9, 3.7, 4.8 und 4.19). \square

Neben den bekannten Lemmata benötigen wir für den Beweis der Typerhaltung für die Programmiersprache \mathcal{L}_c^t ein neues Lemma, welches zusichert, dass der Übergang zu einem grösseren Typ für einen Eintrag in der Typumgebungen keinen Einfluss auf den Typ hat, der für Reihen oder Klassenrumpfe hergeleitet werden kann. (**TODO:** Eine ähnliche Aussage wurde bereits für die Programmiersprache \mathcal{L}_c^m bewiesen).

Lemma 5.9 (Subtyping und Typumgebungen) *Sei $\Gamma \in TEnv$, $id \in Id$ und seien $\tau, \tau' \in Type$ mit $\tau' \lesssim \tau$. Dann gilt:*

- (a) $\forall e \in Exp, \hat{\tau} \in Type : \Gamma[\tau/id] \triangleright e :: \hat{\tau} \Rightarrow \Gamma[\tau'/id] \triangleright e :: \hat{\tau}$
- (b) $\forall b \in Body, \phi \in RType : \Gamma[\tau/id] \triangleright b :: \phi \Rightarrow \Gamma[\tau'/id] \triangleright b :: \phi$

Beweis: Wir führen den Beweis wie üblich durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[\tau/id] \triangleright e :: \hat{\tau}$ und $\Gamma[\tau/id] \triangleright b :: \phi$, wobei wir jeweils nach der zuletzt angewandten Typregel in der Herleitung unterscheiden. Dazu betrachten wir folgende Fälle:

- 1.) $\Gamma[\tau/id] \triangleright c :: \hat{\tau}$ mit Typregel (CONST)

Nach Voraussetzung gilt $c :: \hat{\tau}$, also folgt unmittelbar $\Gamma[\tau'/id] \triangleright c :: \hat{\tau}$ mit Typregel (CONST).

- 2.) $\Gamma[\tau/id] \triangleright id' :: \hat{\tau}$ mit Typregel (ID)

Falls $id = id'$, so gilt $\Gamma[\tau/id](id') = \tau$, also insbesondere $\hat{\tau} = \tau$. Das heisst, es gilt weiter $\Gamma[\tau'/id](id') = \tau'$, und da nach Annahme $\tau' \lesssim \tau$ gilt, folgt die Behauptung mit Typregel (SUBSUME").

Andererfalls, für $id \neq id'$, gilt $\Gamma[\tau/id](id') = \Gamma[\tau'/id](id') = \hat{\tau}$, also ist nichts zu zeigen.

3.) $\Gamma[\tau/id] \triangleright e_1 e_2 :: \hat{\tau}$ mit Typregel (APP)

Das kann nur aus Prämissen der Form $\Gamma[\tau/id] \triangleright e_1 :: \hat{\tau}_2 \rightarrow \hat{\tau}$ und $\Gamma[\tau/id] \triangleright e_2 :: \hat{\tau}_2$ folgen. Nach Induktionsvoraussetzung gilt dann auch

$$\Gamma[\tau'/id] \triangleright e_1 :: \hat{\tau}_2 \rightarrow \hat{\tau},$$

sowie

$$\Gamma[\tau'/id] \triangleright e_2 :: \hat{\tau}_2.$$

Damit folgt unmittelbar

$$\Gamma[\tau'/id] \triangleright e_1 e_2 :: \hat{\tau}$$

mit Typregel (APP).

4.) $\Gamma[\tau/id] \triangleright e :: \hat{\tau}$ mit Typregel (SUBSUME'')

In diesem Fall existiert nach Voraussetzung ein $\hat{\tau}' \in Type$ mit $\hat{\tau}' \lesssim \hat{\tau}$ und es gilt $\Gamma[\tau/id] \triangleright e :: \hat{\tau}'$. Nach Induktionsannahme gilt dann insbesondere

$$\Gamma[\tau'/id] \triangleright e :: \hat{\tau}',$$

also folgt wegen $\hat{\tau}' \lesssim \hat{\tau}$ mit Typregel (SUBSUME'') die Behauptung.

5.) $\Gamma[\tau/id] \triangleright \mathbf{inherit} \ A \ \mathbf{from} \ e; b :: \phi \oplus \phi'$ mit Typregel (INHERIT)

Dieses Typurteil kann ausschliesslich aus den Voraussetzungen $\Gamma[\tau/id] \triangleright self :: \hat{\tau}$, $(\Gamma[\tau/id])^* \triangleright e :: \zeta(\hat{\tau} : \phi)$, $dom_a(\phi) = A$ und $\Gamma[\tau/id][\phi(a)/a]_{a \in A} \triangleright b :: \phi'$ folgen. Für die erste Prämisse folgt unmittelbar mit Induktionsvoraussetzung, dass auch

$$\Gamma[\tau'/id] \triangleright self :: \hat{\tau}$$

gilt.

Falls $id \in Var$, so gilt $(\Gamma[\tau/id])^* = \Gamma^*[\tau/id]$, also folgt nach Induktionsvoraussetzung

$$\Gamma^*[\tau'/id] \triangleright e :: \zeta(\hat{\tau} : \phi),$$

und wegen $\Gamma^*[\tau'/id] = (\Gamma[\tau'/id])^*$ folgt somit

$$(\Gamma[\tau'/id])^* \triangleright e :: \zeta(\hat{\tau} : \phi).$$

Andererseits, für $id \in Id \setminus Var$, gilt $(\Gamma[\tau/id])^* = \Gamma^* = (\Gamma[\tau'/id])^*$, also ist nichts zu zeigen.

Für die letzte Prämisse folgt mit einer analogen Argumentation, dass auch

$$\Gamma[\tau'/id][\phi^{(a)}/a]_{a \in A} \triangleright b :: \phi'$$

gilt.

Somit folgt insgesamt mit Typregel (INHERIT) aus den bisherigen Ergebnissen, dass

$$\Gamma[\tau'/id] \triangleright \mathbf{inherit} \ A \ \mathbf{from} \ e; \ b :: \phi \oplus \phi'$$

gilt, was zu zeigen war.

6.) $\Gamma[\tau/id] \triangleright \mathbf{val} \ a = e; \ r :: (a : \hat{\tau}; \emptyset) \oplus \phi$ mit Typregel (ATTR')

Das bedingt $(\Gamma[\tau/id])^* \triangleright e :: \hat{\tau}$ und $\Gamma[\tau/id][\hat{\tau}/a] \triangleright r :: \phi$. Wie bereits im Fall der (INHERIT)-Regel, folgt leicht mit Fallunterscheidungen nach der Form von id , dass auch

$$(\Gamma[\tau'/id])^* \triangleright e :: \hat{\tau}$$

und

$$\Gamma[\tau'/id][\hat{\tau}/a] \triangleright r :: \phi$$

gilt, und daraus folgt mit Typregel (ATTR') unmittelbar die Behauptung.

Die restlichen Fälle verlaufen ähnlich. □

TODO: Ein weiteres Lemma für die Preservation

Lemma 5.10 (Reihenkonkatenation) *Seien $r_1, r_2 \in Row$ mit $dom_a(r_1) \cap dom_a(r_2) = \emptyset$, $\phi_1, \phi_2 \in RType$ mit $\phi_1 \oplus \phi_2$ definiert, und $\Gamma \in TEnv$ eine beliebige Typumgebung. Dann gilt:*

$$\Gamma \triangleright r_1 :: \phi_1 \ \wedge \ \Gamma[\phi_1^{(a)}/a]_{a \in dom_a(\phi_1)} \triangleright r_2 :: \phi_2 \ \Rightarrow \ \Gamma \triangleright r_1 \oplus r_2 :: \phi_1 \oplus \phi_2$$

Beweis: Das Lemma lässt sich einfach durch Induktion über die Länge von r_1 beweisen, mit Fallunterscheidung nach der syntaktischen Form von r_1 :

1.) $r_1 = \epsilon$

Dann gilt $r_1 \oplus r_2 = r_2$. Das Typurteil $\Gamma \triangleright r_1 :: \phi_1$ kann nur mit Typregel (EMPTY) aus $\phi_1 = \emptyset$ folgen, also gilt $\phi_1 \oplus \phi_2 = \phi_2$. Weiterhin ist $\text{dom}_a(\phi_1) = \text{dom}_a(\emptyset) = \emptyset$, also gilt insbesondere $\Gamma[\phi_1^{(a)}/a]_{a \in \text{dom}_a(\phi_1)} = \Gamma$, und somit folgt $\Gamma \triangleright r_1 \oplus r_2 :: \phi_1 \oplus \phi_2$.

2.) $r_1 = (\mathbf{val} \ a' = e; r)$

Das Typurteil $\Gamma \triangleright \mathbf{val} \ a' = e; r :: \phi_1$ kann nur mit Typregel (ATTR') aus Prämissen der Form $\Gamma^* \triangleright e :: \tau$ und $\Gamma[\tau/a'] \triangleright r :: \phi$ mit $\phi_1 = (a' : \tau; \emptyset) \oplus \phi$ folgen. Wenn $\text{dom}_a(r_1) \cap \text{dom}_a(r_2) = \emptyset$ gilt auch insbesondere $\text{dom}_a(r) \cap \text{dom}_a(r_2) = \emptyset$, und offensichtlich ist $\phi \oplus \phi_2$ definiert. Da $a' \in \text{dom}_a(\phi_1) = \text{dom}_a((a' : \tau; \emptyset) \oplus \phi)$, gilt dann auch $\Gamma[\tau/a'][\phi_1^{(a)}/a]_{a \in \text{dom}_a(\phi_1)} \triangleright r_2 :: \phi_2$. Darauf lässt sich nun die Induktionsvoraussetzung anwenden, und wir erhalten

$$\Gamma[\tau/a'] \triangleright r \oplus r_2 :: \phi \oplus \phi_2.$$

Daraus folgt schliesslich mit Typregel (ATTR'), dass insgesamt

$$\Gamma \triangleright \mathbf{val} \ a' = e; r \oplus r_2 :: (a' : \tau; \emptyset) \oplus \phi \oplus \phi_2$$

gilt, was exakt der Behauptung entspricht, denn es gilt $r_1 = (\mathbf{val} \ a' = e; r)$ und $\phi_1 = (a' : \tau; \emptyset) \oplus \phi$.

3.) $r_1 = (\mathbf{method} \ m = e; r)$

Folgt analog zum vorhergehenden Fall. □

Wie im Fall der Programmiersprachen \mathcal{L}_o^t , \mathcal{L}_o^{rt} und \mathcal{L}_o^{srt} , benötigen wir für die Beweise der Preservation- und Progress-Theoreme noch ein weiteres Lemma, das einer Umkehrung der Typrelation entspricht. Wie zuvor beschränken wir uns auf die Aussagen des Lemmas, die in den weiteren Beweisen benutzt werden, statt das vollständige Lemma zu formulieren.

Lemma 5.11 (Umkehrung der Typrelation)

- (a) Wenn $\Gamma \triangleright op :: \tau$, dann gilt $\tau \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ für $op \in \{+, -, *\}$ oder $\tau \sim \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$ für $op \in \{\leq, \geq, <, >, =\}$.
- (b) Wenn $\Gamma \triangleright id :: \tau$, dann gilt $\Gamma(id) = \tau'$ und $\tau' \lesssim \tau$.
- (c) Wenn $\Gamma \triangleright \lambda x : \tau_1. e :: \tau$, dann gilt $\Gamma[\tau_1/x] \triangleright e :: \tau_2$ und $\tau_1 \rightarrow \tau_2 \lesssim \tau$.

- (d) Wenn $\Gamma \triangleright e_1 e_2 :: \tau$, dann gilt $\Gamma \triangleright e_1 :: \tau'_2 \rightarrow \tau$, $\Gamma \triangleright e_2 :: \tau_2$ und $\tau_2 \lesssim \tau'_2$.
- (e) Wenn $\Gamma \triangleright \mathbf{object} (self : \tau) r \mathbf{end} :: \tau'$, dann gilt $\Gamma^*[\tau/self] \triangleright r :: \phi$ mit $\langle \phi \rangle \sim \tau$ und $\tau \lesssim \tau'$.
- (f) Wenn $\Gamma \triangleright \mathbf{class} (self : \tau) b \mathbf{end} :: \zeta(\tau' : \phi)$, dann gilt $\Gamma^*[\tau/self] \triangleright b :: \phi$ und $\tau = \tau'$ mit $\tau \lesssim \langle \text{methods}(\phi) \rangle$.

Beweis: TODO □

Dank des vorangegangenen Lemmas lassen sich Anwendungen der (SUBSUME')-Regel in den folgenden Beweisen im wesentlichen ignorieren, so dass wir nun das Preservation-Theorem formulieren und leicht beweisen können. ■

Satz 5.2 (Typerhaltung, „Preservation“)

- (a) $\forall \Gamma \in TEnv : \forall e, e' \in Exp : \forall \tau \in Type : \Gamma^* \triangleright e :: \tau \wedge e \rightarrow_e e' \Rightarrow \Gamma^* \triangleright e' :: \tau$
- (b) $\forall \Gamma \in TEnv : \forall r, r' \in Row : \forall \phi \in RType : \Gamma \triangleright r :: \phi \wedge r \rightarrow_r r' \Rightarrow \Gamma \triangleright r' :: \phi$
- (c) $\forall \Gamma \in TEnv : \forall b, b' \in Body : \forall \phi \in RType : \Gamma \triangleright b :: \phi \wedge b \rightarrow_b b' \Rightarrow \Gamma \triangleright b' :: \phi$

Beweis: Wie üblich beweisen wir die typerhaltung durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma^* \triangleright e :: \tau$, $\Gamma \triangleright r :: \phi$ und $\Gamma \triangleright b :: \phi$, mit Fallunterscheidung nach der zuletzt angewendeten Typregel. Dazu betrachten wir die folgenden Fälle:

- 1.) Für (CONST), (ID), (ABSTR) und (EMPTY) gilt gemäß Lemma 5.1 $e \not\rightarrow$ bzw. $r \not\rightarrow$.
- 2.) $\Gamma^* \triangleright e :: \tau$ mit Typregel (SUBSUME')

Nach Voraussetzung existiert ein $\tau' \in Type$ mit $\Gamma^* \triangleright e :: \tau'$ und $\tau' \lesssim \tau$. Nach Induktionsannahme gilt also weiter

$$\Gamma^* \triangleright e' :: \tau',$$

und wegen $\tau' \lesssim \tau$ folgt daraus mit Anwendung von Typregel (SUBSUME'), dass auch

$$\Gamma^* \triangleright e' :: \tau$$

gilt.

3.) $\Gamma^* \triangleright \mathbf{class}(\mathit{self} : \tau) \ b \ \mathbf{end} :: \zeta(\tau : \phi)$ mit Typregel (CLASS)

Dann gilt nach Voraussetzung $(\Gamma^*)^*[\tau/\mathit{self}] \triangleright b :: \phi$ mit $\tau \lesssim \langle \mathit{methods}(\phi) \rangle$. Dann kann der small step $\mathbf{class}(\mathit{self} : \tau) \ b \ \mathbf{end} \rightarrow_e e'$ ausschliesslich mit (CLASS-EVAL) hergeleitet worden sein, was wiederum bedingt, dass ein small step $b \rightarrow_b b'$ existiert und $e' = \mathbf{class}(\mathit{self} : \tau) \ b' \ \mathbf{end}$ gilt. Nach Induktionsannahme gilt also

$$(\Gamma^*)^*[\tau/\mathit{self}] \triangleright b' :: \phi,$$

woraus dann unmittelbar mit Typregel (CLASS) folgt, dass auch

$$\Gamma^* \triangleright \mathbf{class}(\mathit{self} : \tau) \ b' \ \mathbf{end} :: \zeta(\tau : \phi)$$

gilt.

4.) $\Gamma^* \triangleright \mathbf{new} \ e :: \tau$ mit Typregel (NEW)

Also müssen die Prämissen $\Gamma^* \triangleright e :: \zeta(\tau : \phi)$ und $\tau \sim \langle \mathit{methods}(\phi) \rangle$ erfüllt sein. Der small step $\mathbf{new} \ e \rightarrow_e e'$ kann lediglich mit einer der small step Regeln (NEW-EVAL) und (NEW-EXEC) hergeleitet worden sein. Wir unterscheiden also nach der letzten small step Regel in der Herleitung, und zeigen jeweils, dass die Behauptung gilt:

1.) $\mathbf{new} \ e \rightarrow_e e'$ mit small step Regel (NEW-EVAL)

Dann existiert ein $e'' \in \mathit{Exp}$, so dass $e' = \mathbf{new} \ e''$ und $e \rightarrow_e e'$ gilt. Die Behauptung folgt dann unmittelbar mit Induktionsvoraussetzung und Typregel (NEW).

2.) $\mathbf{new} \ e \rightarrow_e e'$ mit small step Regel (NEW-EXEC)

Dann gilt $e = \mathbf{class}(\mathit{self} : \tau') \ r \ \mathbf{end}$ und $e' = \mathbf{object}(\mathit{self} : \tau') \ r \ \mathbf{end}$. Nach Lemma 5.11 muss also

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

gelten, mit $\tau' = \tau$. Wegen $\tau \sim \langle \mathit{methods}(\phi) \rangle$ folgt daraus mit Typregel (OBJECT") schliesslich die Behauptung

$$\Gamma^* \triangleright \mathbf{object}(\mathit{self} : \tau) \ r \ \mathbf{end} :: \tau.$$

5.) $\Gamma \triangleright \mathbf{val} \ a = e; \ r :: (a : \tau; \emptyset) \oplus \phi$ mit Typregel (ATTR')

Dann gilt nach Voraussetzung $\Gamma^* \triangleright e :: \tau$ und $\Gamma[\tau/a] \triangleright r :: \phi$. Der small step $\mathbf{val} \ a = e; \ r \rightarrow_r r'$ kann nur mit einer der small step Regeln (ATTR-LEFT) oder (ATTR-RIGHT) hergeleitet worden sein:

- 1.) **val** $a = e$; $r \rightarrow_r r'$ mit small step Regel (ATTR-LEFT)

Dann existiert ein small step $e \rightarrow_e e'$ und es muss $r' = (\mathbf{val} \ a = e'; r)$ gelten. Dann folgt die Behauptung mit Induktionsvoraussetzung und Typregel (ATTR').

- 2.) **val** $a = e$; $r \rightarrow_r r'$ mit small step Regel (ATTR-RIGHT)

In diesem Fall existiert ein small step $r \rightarrow_r r''$ und es gilt $e \in Val$ und $r' = (\mathbf{val} \ a = e; r'')$. Hier folgt die Behauptung ebenfalls leicht mit Induktionsvoraussetzung und Typregel (ATTR').

- 6.) $\Gamma \triangleright \mathbf{inherit} \ A \ \mathbf{from} \ e; b :: \phi \oplus \phi'$ mit Typregel (INHERIT)

Das kann nur aus Prämissen der Form $\Gamma \triangleright self :: \tau, \Gamma^* \triangleright e :: \zeta(\tau : \phi), dom_a(\phi) = A$ und $\Gamma[\phi^{(a)}/a]_{a \in A} \triangleright b :: \phi'$ folgen. Dann kann der small step **inherit** $A \ \mathbf{from} \ e; b \rightarrow_b b'$ ausschliesslich mit einer der small step Regel (INHERIT-RIGHT), (INHERIT-LEFT) oder (INHERIT-EXEC) hergeleitet worden sein. Entsprechend unterscheiden wir nach der zuletzt angewandten small step Regel:

- 1.) **inherit** $A \ \mathbf{from} \ e; b \rightarrow_b b'$ mit small step Regel (INHERIT-LEFT)

Dann existiert nach Voraussetzung ein small step $e \rightarrow_e e'$, und es gilt $b \in Row$ sowie $b' = \mathbf{inherit} \ A \ \mathbf{from} \ e'; b$. Nach Induktionsannahme gilt

$$\Gamma^* \triangleright e' :: \zeta(\tau : \phi)$$

und mit Typregel (INHERIT) folgt somit unmittelbar, dass auch

$$\Gamma \triangleright \mathbf{inherit} \ A \ \mathbf{from} \ e'; b :: \phi \oplus \phi'$$

gilt.

- 2.) **inherit** $A \ \mathbf{from} \ e; b \rightarrow_b b'$ mit small step Regel (INHERIT-RIGHT)

Dann muss gelten $b' = \mathbf{inherit} \ A \ \mathbf{from} \ e; b''$ und die Behauptung folgt ähnlich einfach, wie im vorangegangenen Fall.

- 3.) **inherit** $A \ \mathbf{from} \ e; b \rightarrow_b b'$ mit small step Regel (INHERIT-EXEC)

Das bedingt $e = \mathbf{class}(self : \hat{\tau}) \ r_1 \ \mathbf{end}$, $b = r_2 \in Row$ und $b' = r_1 \oplus r_2$. Wegen Lemma 5.11 existiert ein $\tau' \in Type$ mit $\Gamma(self) = \tau'$ und $\tau' \lesssim \tau$. Ebenfalls wegen Lemma 5.11 gilt $\hat{\tau} = \tau$ und $(\Gamma^*)^*[\tau/self] \triangleright r_1 :: \phi$. Hieraus lässt sich wegen $(\Gamma^*)^*[\tau/self] =_{free(r_1)} \Gamma[\tau/self]$ und Lemma 5.9 (Subtyping und Typumgebungen) herleiten, dass auch

$$\Gamma[\tau'/self] \triangleright r_1 :: \phi$$

und somit, dass

$$\Gamma \triangleright r_1 :: \phi$$

gilt. Darauf nun lässt sich Lemma 5.10 (Reihenkonkatenation) anwenden, und wir erhalten

$$\Gamma \triangleright r_1 \oplus r_2 :: \phi \oplus \phi',$$

und damit ist die Behauptung gezeigt.

Die restlichen Fälle folgen wie in den Beweisen dieses Satzes für die vorangegangenen Programmiersprachen (vgl. Satz 2.5, 3.1, 4.1 und 4.4). \square

Damit ist gezeigt, dass auch für die Programmiersprache \mathcal{L}_c^t gilt, dass small steps typerhaltend sind. Zum Beweis der Typsicherheit fehlt nun lediglich das Progress-Theorem. Dazu formulieren wir auch für \mathcal{L}_c^t wieder zunächst ein Canonical Forms Lemma, und zeigen dann, mithilfe dieses Lemmas, dass für jeden abgeschlossenen, wohlgetypten Ausdruck ein small step existiert, sofern er nicht bereits ein Wert ist. Da wir im Beweis des Progress-Theorems lediglich die neuen und geänderten Typregeln betrachten, kommen wir bei der Formulierung des Canonical Forms Lemmas mit einer einzigen Aussage aus¹:

Lemma 5.12 (Canonical Forms) *Sei $v \in \text{Val}$, $\tau \in \text{Type}$, und gelte $[] \triangleright v :: \tau$.*

(a) *Wenn $\tau \lesssim \zeta(\tau' : \phi')$, dann gilt $v = \mathbf{class}(\mathbf{self} : \tau') \mathbf{r} \mathbf{end}$.*

Beweis:

(a) Nach Lemma 5.4 und Lemma 5.2 gilt $\tau = \mu t_1 \dots \mu t_n. \zeta(\tau' : \phi')$, das heisst für das Typurteil $[] \triangleright v :: \tau$ kommen lediglich die Typregeln (CLASS) und (SUBSUME) in Frage, und die Behauptung folgt leicht durch Induktion über die Länge der Herleitung des Typurteils mit Fallunterscheidung nach der zuletzt angewandten Typregel. \square

Mit dieser Erkenntnis können wir nun auch für die Programmiersprache \mathcal{L}_c^t ein Progress-Theorem formulieren und beweisen, wobei die Formulierung des Satzes erweitert werden muss im Vergleich zur Formulierung des Satzes für die Programmiersprache \mathcal{L}_o^{st} (Satz 4.5).

Satz 5.3 (Existenz des Übergangsschritts, „Progress“)

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

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

¹Denn, wie üblich beschränken wir uns auf die Aussagen des Lemmas, die wir auch tatsächlich im Beweis des Progress-Satzes benutzen.

(c) $\forall \Gamma \in TEnv, b \in Body \setminus Row, \phi \in RType : \Gamma^+ \triangleright b :: \phi \Rightarrow \exists b' \in Body : b \rightarrow_b b'$

Für echte Klassenrümpfe $b \in Body \setminus Row$ existiert keine Vorstellung von einem Wert, wie das für Ausdrücke und Reihen der Fall ist. Stattdessen werden Klassenrümpfe zu Reihen ausgewertet, entsprechend ist bei der Formulierung des Satzes darauf zu achten, dass für echte Klassenrümpfe stets ein small step existieren muss.

Beweis: Wie üblich führen wir den Beweis durch simultane Induktion über die Länge der Typherleitungen $[] \triangleright e :: \tau$, $\Gamma^+ \triangleright r :: \phi$ und $\Gamma^+ \triangleright b :: \phi$, und unterscheiden jeweils nach der letzten Typregel in der Herleitung. Wir betrachten lediglich die folgenden Fälle für die neuen oder geänderten Typregeln:

1.) $[] \triangleright e :: \tau$ mit Typregel (SUBSUME")

Das Typurteil kann nur aus Prämissen der Form $[] \triangleright e :: \tau'$ und $\tau' \lesssim \tau$ folgen. Also folgt die Behauptung unmittelbar mit Induktionsvoraussetzung.

2.) $[] \triangleright \mathbf{object}(self : \tau) r \mathbf{end} :: \tau$ mit Typregel (OBJECT")

Nach Voraussetzung gilt also $[self : \tau] \triangleright r :: \phi$ mit $\tau \sim \langle methods(\phi) \rangle$, woraus nach Induktionsannahme folgt, dass r entweder bereits ein Reihenwert ist, oder ein small step $r \rightarrow_r r'$ existiert.

Gilt bereits $r \in RVal$, so gilt damit auch $(\mathbf{object}(self : \tau) r \mathbf{end}) \in Val$.

Falls andererseits ein small step $r \rightarrow_r r'$ existiert, so lässt sich für den Gesamtausdruck ein small step $\mathbf{object}(self : \tau) r \mathbf{end} \rightarrow_e \mathbf{object}(self : \tau) r' \mathbf{end}$ mit Regel (OBJECT-EVAL) herleiten.

3.) $[] \triangleright \mathbf{class}(self : \tau) b \mathbf{end} :: \zeta(\tau : \phi)$ mit Typregel (CLASS)

Das bedeutet, es gilt $[self : \tau] \triangleright b :: \phi$ mit $\tau \lesssim \langle methods(\phi) \rangle$. Dann müssen wir unterscheiden, ob b ein echter Klassenrumpf ist, oder ob b bereits eine Reihe ist.

Wenn $b \in Row$, so ist nichts zu zeigen, denn dann gilt $(\mathbf{class}(self : \tau) b \mathbf{end}) \in Val$.

Sollte $b \in Body \setminus Row$ gelten, so existiert nach Induktionsvoraussetzung ein small step $b \rightarrow_b b'$, also existiert insgesamt mit Regel (CLASS-EVAL) ein small step $\mathbf{class}(self : \tau) b \mathbf{end} \rightarrow_e \mathbf{class}(self : \tau) b' \mathbf{end}$.

4.) $[] \triangleright \mathbf{new} e :: \tau$ mit Typregel (NEW)

Dann gilt $[] \triangleright e :: \zeta(\tau : \phi)$ mit $\tau \sim \langle methods(\phi) \rangle$. Nach Induktionsvoraussetzung ist e entweder bereits ein Wert, oder es existiert ein $e' \in Type$ mit $e \rightarrow_e e'$.

Falls e bereits ein Wert ist, so existiert nach Lemma 5.12 (Canonical Forms) ein $r \in Row$ mit $e = \mathbf{class}(self : \tau) r \mathbf{end}$. Dann existiert mit Regel (NEW-EXEC) ein small step $\mathbf{new}(\mathbf{class}(self : \tau) r \mathbf{end}) \rightarrow_e \mathbf{object}(self : \tau) r \mathbf{end}$.

Existiert andererseits für e ein small step $e \rightarrow_e e'$, dann lässt sich mit Regel (NEW-EVAL) ein small step $\mathbf{new} e \rightarrow_e \mathbf{new} e'$ herleiten.

5.) $\Gamma^+ \triangleright \mathbf{inherit} A \mathbf{from} e; b :: \phi \oplus \phi'$ mit Typregel (INHERIT)

Nach Voraussetzung gilt $\Gamma^+ \triangleright self :: \tau$, $(\Gamma^+)^* \triangleright e :: \zeta(\tau : \phi)$, $dom_a(\phi) = A$ und $\Gamma^+[\phi^{(a)}/a]_{a \in A} \triangleright b :: \phi'$.

Wie zuvor müssen wir unterscheiden, ob b ein echter Klassenrumpf, oder bereits eine Reihe ist. Ist $b \in Body \setminus Row$, so existiert bereits nach Induktionsannahme ein $b' \in Body$ mit $b \rightarrow_b b'$. Also existiert insgesamt ein small step $\mathbf{inherit} A \mathbf{from} e; b \rightarrow_b \mathbf{inherit} A \mathbf{from} e; b'$ mit Regel (INHERIT-RIGHT).

Anderenfalls ist b bereits eine Reihe. Wegen $(\Gamma^+)^* = []$ folgt dann nach Induktionsvoraussetzung, dass e entweder bereits ein Wert ist, oder ein small step $e \rightarrow_e e'$ existiert. Letzteres bedeutet, dass insgesamt für den Klassenrumpf ein small step $\mathbf{inherit} A \mathbf{from} e; b \rightarrow_b \mathbf{inherit} A \mathbf{from} e'; b$ mit small step Regel (INHERIT-LEFT) existiert.

Damit bleibt lediglich der Fall zu betrachten, dass $e \in Val$ und $b = r_2 \in Row$ gilt. Nach Lemma 5.12 (Canonical Forms) gilt dann wieder $e = \mathbf{class}(self : \tau) r_1 \mathbf{end}$, und mit Lemma 5.11 folgt $[self : \tau] \triangleright r_1 :: \phi$. Offensichtlich gilt dann wegen $dom_a(\phi) = A$ auch $dom_a(r_1) = A$. Somit existiert insgesamt ein small step $\mathbf{inherit} A \mathbf{from} \mathbf{class}(self : \tau) r_1 \mathbf{end}; r_2 \rightarrow_b r_1 \oplus r_2$ mit Regel (INHERIT-EXEC).

Die übrigen Fälle verlaufen ähnlich zum Beweis des Satzes für die zuvor betrachteten Programmiersprachen (vgl. Satz 2.6, 3.2, 4.2 und 4.5). \square

Basierend auf diesen Ergebnissen lässt sich nun analog zu den vorherigen Programmiersprachen, mithilfe der Preservation- und Progress-Sätze, die Typsicherheit von \mathcal{L}_c^t beweisen.

Satz 5.4 (Typsicherheit, „Safety“) *Wenn $[] \triangleright e :: \tau$, dann bleibt die Berechnung für e nicht stecken.*

Beweis: Wie bereits für die zuvor betrachteten Programmiersprachen, folgt die Typsicherheit unmittelbar aus den Preservation- und Progress-Sätzen. \square

A Big step Interpreter

TODO: Ocaml Version des big steppers.

Index

Sonderzeichen

α -Konversion 14

A

Abgeschlossenheit 13

Attributname 21

Ausdruck 12, 22, 133

B

Berechnung 29

Berechnungsfolge 29

big step 33

 Semantik 16

big step Semantik 33

F

Frei vorkommende Namen 24, 134

Frei vorkommende Variablen 13

G

gebundene Umbenennung 14

K

Klassenrumpf 133

M

Methodenname 21

N

Name 21

O

Objektname 21

Operationelle Semantik 15

R

recursive types 98

Reihe 22

Reiheneinsetzung 25

Reihentyp 38, 137

Reihenwert 23

Rekursive Typen 98

S

small step 17, 27 f., 135

 Regeln 17

 Semantik 16

structural typing 66

Substitution 13, 25, 134

Subtyping 59

 Coercion 66

 Subsumption 66

T

Typ 38

Type 137

Typname 98

Typeregeln

für die Sprache \mathcal{L}_c^t	141
für die Sprache \mathcal{L}_o^m	83
für die Sprache \mathcal{L}_o^{rt}	104
für die Sprache \mathcal{L}_o^{srt}	122
für die Sprache \mathcal{L}_o^{sub}	66
für die Sprache \mathcal{L}_o^t	42

Typsicherheit

Preservation ...	54, 73, 109, 126, 153
Progress	56, 76, 114, 130, 156
Safety	58, 78, 86, 116, 132, 158
Subtyping	67, 123

Typumgebung

Typurteile

für Ausdrücke	42
für Konstante	40
für Reihen	42

V

Variable

Vereinigung

Vererbung

W

Wert

Literaturverzeichnis

- [DP88] DÖRFLER, Willibald ; PESCHEK, Werner: *Einführung in die Mathematik für Informatiker*. Carl Hanser Verlag, 1988. – ISBN 3-446-15112-5
- [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 21. Juli 2007

(Benedikt Meurer)