Universität Siegen
Institut für Theoretische Informatik
Fachbereich Mathematik

SS 2007

Diplomarbeit

Theoretische Grundlagen der Objektorientierung

Benedikt Meurer

25. September 2007

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

Inhaltsverzeichnis

Einleitung

> Mathematics is the gate and key to the sciences... Neglect of mathematics works injury to all knowledge, since one who is ignorant of it cannot know the other sciences of the things of this world. And what is worst, those who are thus ignorant are unable to perceive their own ignorance and so do not seek a remedy.
>
> *(Roger Bacon)*

In den vergangenen zwanzig Jahren hat sich das objekt-orientierte Sprachparadigma zum Standarddenkmodell in der Softwaretechnik entwickelt, was nicht zuletzt mit der jährlich exponentiell steigenden Komplexität der zu lösenden Aufgaben zu erklären ist. Es existieren fast keine Industriezweige oder Wissenschaften, die heutzutage nicht bereits in hohem Maße von der Softwaretechnik abhängig sind.

Parallel zur steigenden Komplexität wächst aber auch die Forderung seitens der Abnehmer nach hoher Qualität und geringer Fehlerwahrscheinlichkeit. Und gerade diese widersprüchliche Ausgangslage hat dazu geführt, dass sich aufgrund der sehr guten Modularisierung, die mit objekt-orientierten Programmiersprachen und Werkzeugen erreichbar ist, die objekt-orientierte Denkweise als Standardkonzept für die Entwicklung nichttrivialer Software durchgesetzt hat.

Allerdings tritt das objekt-orientierte Paradigma selten – mit Ausnahme von exotischen Programmiersprachen wie Eiffel oder SmallTalk – in Reinform auf, sondern ist i.d.R. gepaart mit imperativen oder funktionalen Konzepten. Beispielsweise basieren Java und C++ auf einer imperativen Kernsprache, während O'Caml, wie alle Sprachen der ML-Familie, auf einem funktionalen Kern basiert.

Leider existiert jedoch bis heute keine vollständige Theorie, welche alle wichtigen objekt-orientierten Konzepte beinhaltet. Zwar existieren für O'Caml Ansätze die Programmiersprache formal zu beschreiben und bestimmte Eigenschaften zu beweisen, jedoch umfassen diese Betrachtungen jeweils nur einen Teil der Sprache und sind oft zu informal gehalten. Ähnlich verhält es sich mit theoretischen Betrachtungen von Programmiersprachen wie Java oder C++.

Im Rahmen dieser Arbeit soll nun eine vollständige Theorie für eine objekt-orientierte Programmiersprache entwickelt werden, welche u.a. die objekt-orientierten Kernkonzepte Datenkapselung (engl. *information hiding*), Subtyping und Vererbung (häufig als *Subclassing* bezeichnet) unterstützt. Als Grundlage für die Betrachtungen dient eine einfache funktionale Programmiersprache, die aus mathematischer Sicht leicht zugänglich ist.

Insbesondere interessant ist natürlich in diesem Zusammenhang, ob und wie sich die bekannten objekt-orientierten Konzepte in einer (statisch) typsicheren Programmiersprache unterbringen lassen. Statisch typsicher bedeutet dabei, dass bei der statischen Typüberprüfung, die während der Compilephase durchgeführt wird, bereits alle Fehler entdeckt werden, die in einem naiven Interpreter zur Laufzeit fehlerhaftes Verhalten verursachen

würden. D.h. eine statisch typsichere Programmiersprache benötigt i.d.R. keine Laufzeittypüberprüfung. Wir werden später sehen, wie sich diese Eigenschaft formal handhaben lässt.

Da in diesem Dokument nur eine kurze Einführung in die zugrundeliegende Thematik gegeben wird, sollte der Leser vertraut sein mit den Grundlagen der Computerprogrammierung und insbesondere mit den Grundlagen funktionaler Programmiersprachen. Eine gute Einführung in die zugrunde liegende Thematik bietet unter anderem [ASS01].

Die Arbeit ist wie folgt aufgebaut:

- In Kapitel 1 wird kurz die grundlegende Mathematik wiederholt und eine einfache funktionale Programmiersprache vorgestellt, welche als Grundlage für die nachfolgend vorgestellten objekt-orientierten Programmiersprachen dient.
- In Kapitel 2 wird zunächst eine Erweiterung der rein funktionalen Programmiersprache um objekt-orientierte Konzepte betrachtet. Für diese neue Sprache wird dann bewiesen, dass sie typsicher ist.
- Kapitel 3 und Kapitel 4 erweitern das Typsystem der einfachen objekt-orientierten Programmiersprache um die Konzepte Subtyping und rekursive Typen, wobei zunächst diese Konzepte separat eingeführt werden, und anschliessend eine Programmiersprache entwickelt wird, welche beide Konzepte umfasst.
- Schließlich wird in Kapitel 5 das Typsystem und die Semantik der Sprache um das objekt-orientierte Kernkonzept Vererbung erweitert, und es wird wieder gezeigt, dass diese neue Sprache nach wie vor typsicher ist.

1 Grundlagen

Zu Anfang stellen wir zunächst die zugrunde liegende Mathematik und die einfache funktionale Programmiersprache vor, auf der die in diesem Dokument dargestellten Programmiersprachen basieren. Sollte der Leser bereits mit den theoretischen Grundlagen von Programmiersprachen der ML-Familie vertraut sein, kann er dieses Kapitel auch überspringen.

1.1 Mathematische Grundlagen

In diesem Abschnitt 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 Wiederholung oder Gedächtnisstütze dienen.

Definition 1.1 (Potenzmenge) Sei A eine beliebige Menge. Die Potenzmenge $\mathcal{P}(A)$ ist die Menge, für die gilt:

(a) $\forall B \subseteq A : B \in \mathcal{P}(A)$

(b) $\forall B \in \mathcal{P}(A) : B \subseteq A$

Definition 1.2 (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 1.3 (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.

In diesem Dokument werden wir häufiger partielle Funktionen erweitern, dazu definieren wir die folgende Schreibweise:

Definition 1.4 Seien A, B Mengen, $a \in A$, $b \in B$ und $f : A \rightharpoonup B$ eine partielle Funktion. Dann bezeichnet $f[{}^{b}/_{a}]$ die partielle Funktion $g : A \rightharpoonup B$ mit den folgenden Eigenschaften.

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

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 1.5 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 $\bot$ oder $\bot_D$. 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 1.6 (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) $\big(\mathcal{P}(M), \subseteq\big)$ *ist eine dcpo mit kleinstem Element* $\emptyset$.

(b) $\big(\mathcal{P}(M), \supseteq\big)$ *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 1.7 (Monotone und stetige Abbildungen) Seien $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$ dcpos.

(a) Eine (totale) Abbildung $f : D \to E$ 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 $f : D \to E$ 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 $f : D \to E$ 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: Trivial. □

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 $\bot$ besitzt, und $f : D \to D$ eine stetige Abbildung. Dann gilt:*

(a) Die Menge $\{f^n(\bot) \mid n \in \mathbb{N}\}$ ist gerichtet.

(b) $\mu f = \bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ ist kleinster Fixpunkt von f.

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

Beweis:

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

(b) Wir zeigen zunächst, dass μf ein Fixpunkt von f ist, also:

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

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 $\mu f = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$ die kleinste obere Schranke aller $f^n(\bot)$ ist, genügt es zu zeigen, dass d eine obere Schranke von $\{f^n(\bot) \mid n \in \mathbb{N}\}$ ist. Dazu zeigen wir durch vollständige Induktion über n, dass $f^n(\bot) \sqsubseteq d$ für alle $n \in \mathbb{N}$:

- $n = 0$

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

- $n \rightsquigarrow n + 1$

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

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

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

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

Einführend soll kurz die einfache funktionale Programmiersprache $\mathcal{L}_f$ vorgestellt werden. 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]) dargestellten übereinstimmt.

Dieser Abschnitt führt nur oberflächlich die Syntax und Semantik der Programmiersprache $\mathcal{L}_f$ ein, welche als Grundlage für die weiteren in dieser Arbeit behandelten Programmiersprachen dient. Es wird vorausgesetzt, dass der Leser über rudimentäre Kenntnisse der ML-Sprachfamilie verfügt. Gute Einführungen in (Standard) ML finden sich unter anderem in [Pau96], [Sta92] und [Ull98].

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

Die Programmiersprache $\mathcal{L}_f$ orientiert sich, wie auch alle weiteren in diesem Dokument betrachteten Sprachen, syntaktisch stark an der Programmiersprache O'Caml[1],

[1] `http://caml.inria.fr/ocaml/` (Stand: 19.09.2007)

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 funktionale Sprache Caml Light[2], den Vorgänger von O'Caml, allerdings verzichten wir auf einige Konzepte wie Pattern Matching und Exceptions, da diese für die Betrachtungen in diesem Dokument nicht relevant sind.

Definition 1.8 Vorgegeben sei

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

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

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

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

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

Definition 1.9 (Abstrakte Syntax von $\mathcal{L}_f$) Die Menge *Op* aller Operatoren *op* ist definiert durch die kontextfreie Grammatik

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

die Menge *Const* aller Konstanten c durch

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

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

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

[2] `http://caml.inria.fr/caml-light/` (Stand: 19.09.2007)

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.

Trotz dieser Vereinfachung ist die Programmiersprache $\mathcal{L}_f$ hinreichend mächtig. Wir werden im weiteren Verlauf dieses Abschnitts sehen, dass man die Funktionen *div* und *mod* in der Sprache $\mathcal{L}_f$ formulieren kann. Darauf aufbauend könnte man dann Operatoren für die Ganzzahldivision und die Divisionsrestbildung als syntaktischen Zucker in die Programmiersprache aufnehmen.

An dieser Stelle soll kurz der Unterschied zwischen konkreter und abstrakter Syntax verdeutlich werden: Die konkrete Syntax beschreibt Zeichenketten, die durch den Programmierer erstellt werden, wobei eine derartige Zeichenkette häufig als *Quelltext* (engl.: *source code*) bezeichnet wird. Die abstrakte Syntax hingegen beschreibt die interne Repräsentation eines Programmes in einer Maschine, basierend auf derer dann die Interpretation des Programms erfolgt. Die Übersetzung von konkreter Syntax in abstrakte Syntax geschieht gemäß [ASU99, S.6f] in zwei Phasen: Der *lexikalischen Analyse*, während der die Zeichenfolge in Symbole zerlegt wird, gefolgt von der *syntaktischen Analyse*, während der die Symbole zu grammatikalischen Sätzen (in der abstrakten Syntax) zugesammengefasst werden.

Die Betrachtungen in diesem Dokument beschränken sich ausschließlich auf die jeweilige abstrakte Syntax, also die interne Repräsentation von Programmen der jeweiligen Sprache. Stellenweise werden wir jedoch ergänzende Syntax einführen, die die Kodierung von Programmen in der abstrakten Syntax erleichtert.

Frei vorkommende Variablen und Substitution

Ein wesentlicher Aspekt der Syntax von Programmiersprachen ist der *Gültigkeitsbereich* von Variablen. Ein Vorkommen einer Variable x heißt *gebunden*, wenn es im Bereich eines Bindungsmechanismus für x vorkommt. In der einfachen Programmiersprache $\mathcal{L}_f$ existieren drei Arten von Bindungsmechanismen:

- λ-Abstraktionen der Form $\lambda x.e$ mit e als Gültigkeitsbereich von x,
- rekursive Ausdrücke der Form $\mathbf{rec}\, x.e$ wiederum mit e als Gültigkeitsbereich, und
- **let**-Ausdrücke der Form $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$ mit e_2 als Gültigkeitsbereich von x.

Ein ungebundenes Vorkommen einer Variablen x wird als *frei* bezeichnet. Die nachfolgende Definition formalisiert diese intuitive Festlegung:

Definition 1.10 (Frei vorkommende Variablen) Die Menge $\mathit{free}\,(e)$ aller im Ausdruck $e \in \mathit{Exp}$ *frei vorkommenden Variablen* ist wie folgt induktiv über die Struktur von e definiert:

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

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

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

Anhand dieser Definition sind Bindungsmechanismen immer dadurch auszumachen, dass jeweils aus der Menge der im Gültigkeitsbereich frei vorkommenden Variablen die durch die Bindung spezifizierten Variablen entfernt werden.

Definition 1.11 (Substitution) Für $e, e' \in \mathit{Exp}$ und $x \in \mathit{Var}$ ist der Ausdruck $e'[{}^{e}/_{x}]$, der aus e' durch *Substitution* von e für x entsteht, wie folgt induktiv über die Struktur von e definiert:

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

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

$$(\lambda x.x\, y)[{}^{+\,x\,1}/_{y}]$$

nicht definiert, denn es gilt $x \in \mathit{free}\,(+\,x\,1)$. Wir treffen deshalb in Übereinstimmung mit [Pie02, S. 71] die folgende Vereinbarung:

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

$$(\lambda x'.x'\, y)[{}^{+\,x\,1}/_{y}]$$

und die Substitution ist definiert, da $x' \notin \{y\} \cup \mathit{free}\,(+\,x\,1)$. Dieses Austauschen von Namen gebundener Variablen in Ausdrücken oder Formeln wird allgemein als *gebundene Umbenennung* bezeichnet. Im Bereich der formalen Semantik verwendet man für diese Umformungsregel den Begriff *α-Konversion* (vgl. [Pau95, S.7]).

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.

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

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

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

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

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

Damit kann für die Substitution ein Algorithmus formuliert werden, der die notwendigen gebundenen Umbenennungen 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

Zur einfacheren Notation von Programmen definieren wir, wie bereits angedeutet, noch zusätzliche Hilfssyntax, die vor jeglicher Bearbeitung zunächst in die zuvor definierte Kernsyntax übersetzt wird:

$e_1\ op\ e_2$	für	$op\ e_1\ e_2$	Infixnotation
$-e$	für	$0 - e$	unäres Minus
$\mathbf{let}\ x\ x_1 \ldots x_n = e_1\ \mathbf{in}\ e_2$	für	$\mathbf{let}\ x = \lambda x_1. \ldots \lambda x_n.e_1\ \mathbf{in}\ e_2$	Funktionsdef.
$\mathbf{let\,rec}\ x\ x_1 \ldots x_n = e_1\ \mathbf{in}\ e_2$	für	$\mathbf{let}\ x = \mathbf{rec}\ x.\lambda x_1. \ldots \lambda x_n.e_1\ \mathbf{in}\ e_2$	rek. Funktionsdef.

Weiterhin geben wir, wie bereits angedeutet, Implementationen der Funktionen *div* und *mod*, die als Basis für die Einführung der fehlenden Divisionsoperatoren als syntaktischer Zucker dienen könnten.

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

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

[3] `http://www.informatik.uni-siegen.de/theo/tpml/` (Stand: 19.09.2007)

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 gar nicht erweiterbar, so dass für alle in diesem Dokument beschriebenen Sprachen jeweils neue Semantiken angegeben werden müssten. Aus diesem Grund verwenden wir ausschließlich operationelle Semantik zur Beschreibung der Programmiersprachen[4].

Für die Beschreibung der Umformung von Ausdrücken in präziser mathematischer Form bieten sich zwei Möglichkeiten an (vgl. [Sie06]), 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. Hierzu muss zunächst festlegt werden, wie die Operatoren der Sprache zu interpretieren sind.

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

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

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

Auf eine exakte Definition der Interpretationen der verschiedenen Operatoren wird hier verzichtet. Stattdessen nehmen wir an, dass diese Funktionen entsprechend ihrer intuitiven Bedeutung definiert sind. Zum Beispiel entspricht also die Interpretation des Stern-Operators der Multiplikation auf den ganzen Zahlen

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

[4]Neben operationeller und denotationeller Semantik existiert mit der *axiomatischen Semantik* noch ein dritter Ansatz (vgl. [Pie02, S.33]), der jedoch heutzutage nahezu bedeutungslos und für unsere Betrachtungen ebenfalls ungeeignet ist.

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

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

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

Definition 1.13 Die Menge $\mathit{Val} \subseteq \mathit{Exp}$ 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.

In der Terminologie 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 $e \to_e e'$ mit $e, e' \in \mathit{Exp}$.

Genau genommen handelt es sich bei $\to_e$ um eine (Auswertungs-)Relation auf *Exp*, diese Sichtweise ist jedoch unüblich. Stattdessen spricht man hier von small steps als *Reduktionen*, und betrachtet $e \to_e e'$ tatsächlich eher als Formel im höheren Sinn, denn als Relation.

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

Definition 1.15 (Gültige small steps für $\mathcal{L}_f$) Ein small step $e \to_e e'$ mit $e, e' \in \mathit{Exp}$ heißt gültig für $\mathcal{L}_f$, wenn er sich mit den folgenden Regeln herleiten lässt:

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

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

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

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

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

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

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

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

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

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

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 $* 21\, 2 \to 42$.
- Die Regel (Beta-V), die als *β-value*-Regel bezeichnet wird, beschreibt die Parameterübergabe: Die Anwendung einer Funktion $\lambda x.e$ auf einen Wert v bewirkt, dass jedes Vorkommen des formalen Parameters x im Rumpf e durch den aktuellen Parameter v ersetzt wird. Zu beachten ist, dass (Beta-V) nur anwendbar ist, wenn der aktuelle Parameter ein Wert, also schon vollständig ausgewertet ist[5].
- Die Regel (App-Left) erlaubt es, den linken Teil einer Applikation weiter auszuwerten, zum Beispiel solange, bis eine λ-Abstraktion erreicht ist.
- Die Regel (App-Right) erlaubt es, das Argument einer Applikation auszuwerten[6].
- (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 ersten Teilausdruck 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.

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

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

- (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.

Bevor wir nun, basierend auf der in diesem Abschnitt vorgestellten Programmiersprache $\mathcal{L}_f$, im anschließenden Kapitel eine objekt-orientierte Erweiterung vorstellen, wollen wir noch kurz anhand eines einfachen Beispiels demonstrieren, wie small steps mit dem Regelwerk aus Definition 1.15 hergeleitet werden können.

Beispiel 1.1 Betrachten wir beispielhaft eine Auswertung des Ausdrucks

$$(\lambda x.x + 1)\,(6 * 7)$$

mit den soeben definierten small step Regeln. Intuitiv ist klar, dass das Ergebnis der Auswertung 43 ist. Bevor wir allerdings das intuitive Ergebnis mit Hilfe der small step Semantik verifizieren können, müssen wir zunächst den Ausdruck in Kernsyntax übersetzen.

$$\begin{array}{lll} & (\lambda x. + x\,1)\,(* \,6\,7) & \\ \longrightarrow & (\lambda x. + x\,1)\,42 & \text{mit (App-Right) und (Op)} \\ \longrightarrow & +\,42\,1 & \text{mit (Beta-V)} \\ \longrightarrow & 43 & \text{mit (Op)} \end{array}$$

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. Da sich die Vorlesung „Theorie der Programmierung“ so weit wie möglich an der Programmiersprache O'Caml[1] orientiert, 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 $\mathcal{L}_o$

Um nun die Syntax der Programmiersprache $\mathcal{L}_f$ um Objekte erweitern zu können, muss zunächst geklärt werden, was überhaupt unter einem Objekt zu verstehen ist. Wegner gibt dazu in [Weg90] die folgende Definition:

> *„Objekte sind Mengen von Operationen, die einen Zustand teilen.“*

Mit anderen Worten besteht ein Objekt also aus beliebig vielen Operationen und einem eigenen Zustand. Im Kontext von objekt-orientierten Programmiersprachen bezeichnet man diese Operationen üblicherweise als *Methoden* und den Zustand als *Attribute*.

Entsprechend muss die Syntax vorsehen, dass Objekte sowohl Methoden als auch Attribute enthalten können. Da ein Objekt mehrere Methoden und mehrere Attribute enthalten kann, müssen diese entsprechend identifizierbar sein. Dazu definieren wir zunächst neue Namen:

Definition 2.1

(a) Vorgegeben seien

- eine Menge *Method* von *Methodennamen* m,
- eine 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:

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

[1] `http://caml.inria.fr/ocaml/` (Stand: 19.09.2007)

Für die Menge *Method* gelten ähnlich wie für die Menge *Var* keinerlei Einschränkungen. Eine Implementierung wird üblicherweise *Method* = *Var* wählen. Die Menge der Attributnamen *Attribute* hingegen muss disjunkt zur Menge der Variablen *Var* sein, da es sich bei Attributen nicht um „gewöhnliche Variablen“ handelt, sondern mit ihnen der Zustand eines Objekts beschrieben werden soll[2].

Zusätzlich zu den Attribut- und Methodennamen benötigen wir noch einen speziellen Bezeichner *self*, der das umgebende Objekt identifiziert. Dieser entspricht dem speziellen Bezeichner **this** in Java oder C++. Während es in O'Caml möglich ist, den Bezeichner für ein Objekt frei zu wählen, beschränken wir uns in der abstrakten Syntax hier auf einen Bezeichner, um die Definition der Semantik und des Typsystems zu erleichtern[3].

Damit können wir nun die kontextfreie Grammatik der Programmiersprache $\mathcal{L}_f$ durch objekt-orientierte Konzepte erweitern. Objekte sollen – wie bereits erwähnt – aus Attributen und Methoden bestehen. Hierzu führen wir zusätzlich zu den Ausdrücken die syntaktische Form der Reihe ein. Jedes Objekt besteht aus einer Reihe. Dabei enthält eine Reihe beliebig viele Attribut- und Methodendeklarationen in beliebiger Reihenfolge.

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

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

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

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

Dabei legen wir fest, dass die Namen aller in einer Reihe deklarierten Attribute paarweise verschieden sein müssen.

Methodenaufrufe $e_1 \# m$ verhalten sich hierbei analog zu Java oder C++: Zunächst wird der Ausdruck e ausgewertet, und sollte sich dann ein Objekt ergeben, so wird diesem die Nachricht m gesendet (also die Methode mit Namen m ausgeführt).

Objekte **object** (*self*) r **end** bestehen wie bereits erläutert aus einer Liste von Attributen und Methoden, der Reihe r. Innerhalb der Methoden in r wird dann über den Namen *self* das Objekt selbst referenziert. Man kann also dem Objekt über *self* andere Nachrichten schicken, d.h. Objekte sind implizit rekursive Datenstrukturen.

Für Duplikationen gibt es kein äquivalentes Konzept in Java oder C++. Eine Duplikation $\{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\}$ kann nur innerhalb von Methoden verwendet werden. Damit wird eine Kopie des zugehörigen Objekts erstellt, wobei die Attribute $a_1, \ldots, a_n$

[2] Das allein ist noch kein Grund, die Mengen disjunkt zu wählen, wir werden aber später sehen, dass dieser technische Trick die Beschreibung der Semantik und des Typsystems erheblich vereinfacht.

[3] In der konkreten Syntax könnten natürlich analog zu O'Caml unterschiedliche *self*-Bezeichner zugelassen werden, die dann entsprechend vorübersetzt werden.

des kopierten Objekts mit den neuen Werten $e_1, \ldots, e_n$ belegt werden. Betrachten wir dazu das Beispiel eines Punktobjekts:

object (*self*)
  val $x = 1$;
  val $y = 5$;
  method $move = \lambda dx.\lambda dy.\{\langle x = x + dx; y = y + dy\rangle\}$;
  ϵ
end

Dieses Objekt besitzt genau zwei Attribute x und y, die die Koordinaten des Punkts darstellen. Die Methode *move* liefert ein neues Punktobjekt, welches basierend auf den Koordinaten des Punktes um dx und dy verschoben ist. In Java oder C++ würde eine derartige Funktionalität über einen speziellen Konstruktor realisiert. Da $\mathcal{L}_o$ jedoch rein auf Objekten, ohne Klassen, arbeitet, wird hier das aus O'Caml bekannte Konzept der Duplikationen verwendet.

Die Festlegung, dass die Namen aller Attribute in einer Reihe verschieden sein müssen, 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 Umbenennungen die Namen der Attributdeklarationen in einer Reihe disjunkt gemacht.

Für Methoden ist explizit zugelassen, dass mehrere Methodendeklarationen mit gleichem Methodennamen in einer Reihe vorkommen dürfen. Die zugrunde liegende Idee ist, dass es bei der in Kapitel 5 vorgestellten Vererbung möglich sein soll, geerbte, also zuvor deklarierte Methoden, zu überschreiben.

Zum Umgang mit Reihen benötigen wir einige einfache Funktionen, die beispielsweise die Namen aller in einer Reihe enthaltenen Attribute oder Methoden auflisten. Diese Funktionen sind nachfolgend induktiv über die Struktur von Reihen definiert.

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

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

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

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

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

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

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

(d) Seien $r_1, r_2 \in \mathit{Row}$ mit $\mathit{dom}_a(r_1) \cap \mathit{dom}_a(r_2) = \emptyset$. Die Reihe $r_1 \oplus r_2$, die durch Konkatenation der Reihen r_1 und r_2 ensteht, ist wie folgt induktiv über die Struktur von r_1 definiert:

$$\begin{array}{rcl} \epsilon \oplus r_2 & = & r_2 \\ (\textbf{val}\ a = e;\ r_1') \oplus r_2 & = & \textbf{val}\ a = e;\ (r_1' \oplus r_2) \\ (\textbf{method}\ m = e;\ r_1') \oplus r_2 & = & \textbf{method}\ m = e;\ (r_1' \oplus r_2) \end{array}$$

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

Konvention 2.1 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

Für die Programmiersprache $\mathcal{L}_o$ führen wir ebenfalls syntaktischen Zucker ein, der eine einfachere Notation von Sprachkonzepten ermöglicht. Dazu führen wir zunächst, wie zuvor für **let**-Ausdrücke in $\mathcal{L}_f$, eine vereinfachte Schreibweise für Methoden mit Paramtern ein (basierend auf der in [Rém02] beschriebenen Syntax von O'Caml):

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

Darüber hinaus vereinbaren wir, dass das abschließende ϵ in einer Reihe weggelassen werden kann.

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

Wir wollen die operationelle Semantik der Programmiersprache $\mathcal{L}_o$ basierend auf der operationellen Semantik der Programmiersprache $\mathcal{L}_f$ (vgl. Abschnitt 1.2.2) definieren. Dazu müssen wir zunächst die Menge der Werte *Val* erweitern. Dazu benötigen wir neben *Val* noch eine Menge $\mathit{RVal} \subseteq \mathit{Row}$ mit sogenannten *Reihenwerten*.

Definition 2.4 (Werte und Reihenwerte)

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

$$\begin{array}{rcl} \omega & ::= & \epsilon \\ & | & \textbf{val}\ a = v;\ \omega \\ & | & \textbf{method}\ m = e;\ \omega \end{array}$$

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

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

zur kontextfreien Grammatik von $\mathcal{L}_f$ (vgl. Definition 1.13) hinzunimmt.

Intuitiv ist ein Objekt genau dann ein Wert, wenn alle enthaltenen Attributausdrücke bereits Werte sind, d.h. hinter jedem Attribut des Objekts ein Wert steht. Dies entspricht der Semantik von Objekten in Java, wobei in Java Objekte immer nur als Wert auftreten können, d.h. die Auswertung von Attributdeklarationen erfolgt bereits vor der Instantiierung der zugehörigen Klasse (vgl. Kapitel 5).

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

Da mit $\mathcal{L}_o$ neue syntaktische Konstrukte, und insbesondere neue Bindungsmechanismen, eingeführt worden sind, müssen wir entsprechend definieren, was wir im Kontext der Programmiersprache $\mathcal{L}_o$ unter freien Vorkommen von Namen verstehen wollen. Wichtig ist hierbei zu beachten, dass in $\mathcal{L}_o$ neben den Variablen noch weitere Arten von Namen existieren. Entsprechend verallgemeinern wir das Prinzip der frei vorkommenden Namen auf Bezeichner $id \in Id$.

Definition 2.5 (Frei vorkommende Namen)

(a) Die Menge $\mathit{free}\,(e)$ aller im Ausdruck $e \in \mathit{Exp}$ *frei vorkommenden Namen* erhält man durch folgende Verallgemeinerung von Definition 1.10:

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

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

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

Es mag zunächst fehlerhaft erscheinen, dass Methodennamen in der obigen Definition nicht berücksichtigt worden sind. Der Grund hierfür ist jedoch, dass durch Gültigkeitsbereiche immer *statische Bindungen* ausgedrückt werden, also Bindungen, die bereits vor der Ausführung des Programms feststehen.

Für das Versenden von Nachrichten an Objekte, d.h. das Aufrufen von Methoden, wird jedoch eine *dynamische Bindung* benutzt, der sogenannte *dynamic dispatch*. Das bedeutet, es wird erst zur Laufzeit bestimmt, welche Methode in welchem Objekt durch das Versenden einer Nachricht ausgeführt werden soll. Entsprechend kann diese Bindung

nicht durch eine auf rein syntaktischen Informationen basierende Definition ausgedrückt werden.

Darüber hinaus ist zu beachten, dass *self* frei in Duplikationen vorkommt, obwohl *self* syntaktisch nicht repräsentiert ist. Hierbei handelt es sich um einen technischen Trick, welcher zum Beweis der Typsicherheit von $\mathcal{L}_o$ in Abschnitt 2.3.3 benötigt wird. Die Intuition hierbei ist, dass Duplikationen innerhalb abgeschlossener Ausdrücke stets im Gültigkeitsbereich von Objekten stehen müssen, denn *self* kann nur durch Objekte gebunden werden.

Man hätte an dieser Stelle auch statt der aus O'Caml bekannten Notation eine andere syntaktische Repräsentation für Duplikationen wählen können, nämlich

$$\mathit{self}\ \{\langle a_1 = e_1; \ldots;\ a_n = e_n\rangle\},$$

durch die diese Bindung explizit ausgedrückt wird.

Definition 2.6 Die Menge $\mathit{Exp}^\star \subseteq \mathit{Exp}$ aller Ausdrücke, die keine freien Vorkommen von Attribut- und Objektnamen enthalten, ist wie folgt definiert:

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

Nur Ausdrücke aus dieser Menge $\mathit{Exp}^\star$ können substituiert werden, denn nur für solche Ausdrücke kann die im Folgenden definierte Substitution zu einer totalen Funktion erweitert werden. Für Attribut- und Objektnamen gilt die α-Konversion nicht, also kann bei der Substitution eines Ausdrucks mit freien Attribut- oder Objektnamen, im Falle eines Konflikts, dieser nicht durch gebundene Umbenennung korrigiert werden. Der Begriff Konflikt bezeichnet in diesem Zusammenhang das Entstehen einer neuen Bindung für einen Bezeichner.

Die Tatsache, dass α-Konversion ausschließlich auf Variablen gilt, mag im ersten Moment wie eine unnötige Einschränkung erscheinen. Tatsächlich wäre es möglich, die Programmiersprache $\mathcal{L}_o$ entsprechend abzuändern, dass α-Konversion auch auf Attributnamen gilt, allerdings wäre es dann nicht mehr möglich basierend auf dieser Programmiersprache in Kapitel 5 Vererbung einzuführen.

Betrachtet man andere objekt-orientierte Programmiersprachen mit Vererbung, wie zum Beispiel Java oder O'Caml, so stellt man fest, dass auch dort die α-Konversion niemals für Attribute gilt. Sei beispielsweise eine Java-Klasse A wie folgt definiert[4]:

```
public class A {
  protected int a = 1;
  public int getA() { return a; }
}
```

Für sich allein genommen gilt für diese Klasse die α-Konversion auch auf Attributen, d.h. das Attribut a könnte in einen beliebigen Name umbenannt werden. Allerdings sind als **protected** markierte Attribute nicht nur innerhalb von Klassen sichtbar, sondern auch in erbenden Klassen. Sei beispielsweise B die wie folgt definierte Klasse:

[4]Wie wir in Kapitel 5 sehen werden, entsprechen die Attribute in unserer Programmiersprache den Attributen mit Sichtbarkeitsbereich **protected** in Java.

```
public class B extends A {
  protected int b = 1;
  public int calc() { return a + b; }
}
```

Würde man nun das Attribut a innerhalb der Klasse A umbenennen, so würde die Klasse B sich nicht mehr übersetzen lassen. Analog ließe sich ein Beispiel in O'Caml konstruieren. Allgemein resultiert das Problem aus der Tatsache, dass Attribute keinen abgeschlossenen Gültigkeitsbereich besitzen, sondern einen offenen, erweiterbaren Gültigkeitsbereich. Folglich ist es leicht einzusehen, dass die α-Konversion für Attributnamen nicht gilt.

Definition 2.7 (Reiheneinsetzung) Für $r \in \mathit{Row}$, $e \in \mathit{Exp}^\star$ und $a \in \mathit{dom}_a(r)$ 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:

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

Bei der Reiheneinsetzung wird einfach der Ausdruck in einer bestimmten Attributdeklaration durch einen neuen Ausdruck ersetzt. Der neue Ausdruck darf aber aus Gründen der Konfliktvermeidung keine freien Attribut- und Objektnamen besitzen. Entscheidend ist an dieser Stelle, dass die Namen der Attribute in einer Reihe disjunkt sein müssen.

Damit können wir nun die Substitution für die Programmiersprache $\mathcal{L}_o$ nachfolgend definieren. Um die weiteren Betrachtungen zu vereinfachen, integrieren wir für den Fall der Substitution eines Objekts für *self* in eine Duplikation bereits Aspekte der späteren Semantik.

Die Idee hierbei ist wie folgt: Da *self* ausschließlich durch Objekte gebunden werden kann, steht *self* stets für das umgebende Objekt. Das Substitutieren für *self* entspricht also – wie nachfolgend durch die small step Semantik dargestellt – dem Auffalten eines Objekts, d.h. nach der Substitution ist kein umgebendes Objekt mehr vorhanden. Die Duplikation macht aber nur innerhalb eines Objektes Sinn, also wird entsprechend bei der Substitution die Duplikation zu einem vollständigen Objekt erweitert, mit den Informationen des substituierten Objekts.

Definition 2.8 (Substitution)

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

Definition 1.11:

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

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

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

Es ist wieder zu beachten, dass gemäß Konvention 1.1 und Konvention 2.1 die Substitution total ist, da der substituierte Ausdruck keine freien Vorkommen von Attribut- oder Objektnamen enthält. Für die übrigen Ausdrücke $e \in \mathit{Exp} \setminus \mathit{Exp}^\star$ ist die Substitution undefiniert.

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 sowie den Bezeichner, für den substituiert wird.

Lemma 2.1 (Eindeutigkeit der Substitution)

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

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

Beweis: Folgt wegen Konvention 1.1 und Konvention 2.1 unmittelbar aus Definition 2.8 (Substitution). □

Es ist leicht zu sehen, dass das Substituieren eines Ausdrucks für einen Bezeichner innerhalb eines Ausdrucks oder einer Reihe, in der dieser Bezeichner nicht frei vorkommt, wieder zum gleichen Ausdruck bzw. zur gleichen Reihe führt. Hierbei ist zu beachten, dass gebundene Umbenennung nach den zuvor getroffenen Konventionen keinen Einfluss auf die Gleichheit hat. Entsprechend gilt das nachfolgende Lemma:

Lemma 2.2 (Frei vorkommende Namen und Substitution)

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

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

Beweis: Leicht durch simultane Induktion zu beweisen. □

2.2.2 Small step Semantik

Die small step Semantik der Programmiersprache $\mathcal{L}_o$ entsteht durch Erweiterung der small step Semantik von $\mathcal{L}_f$. Während aber bei $\mathcal{L}_f$ die small step Relation ausschließlich auf *Exp* definiert worden ist, müssen für $\mathcal{L}_o$ auch small steps auf Reihen betrachtet werden, entsprechend ist die Definition eines small steps zu erweitern:

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

Das Regelwerk zur Herleitung gültiger small steps umfasst folglich sowohl Regeln für Ausdrücke, als auch Regeln für Reihen. Als Grundlage dienen die small step Regeln der Sprache $\mathcal{L}_f$. Neu hinzu kommen small step Regeln für Objekte und Methodenaufrufe, sowie small step Regeln für Attribut- und Methodendeklarationen.

Insbesondere werden keine small step Regeln für Duplikationen benötigt, da die notwendige Semantik für Duplikationen bereits durch die Substitution (vgl. Definition 2.8) behandelt wird. Weiterhin gilt zu beachten, dass neben den small step Regeln für die Ausdrücke der konkreten Kernsyntax auch small steps für die Hilfsausdrücke $\omega\#m$ existieren müssen.

Definition 2.10 (Gültige small steps für $\mathcal{L}_o$) Ein small step der Form $e \to_e e'$ mit $e, e' \in \mathit{Exp}$ oder $r \to_r r'$ mit $r, r' \in \mathit{Row}$ heißt *gültig* für $\mathcal{L}_o$, wenn er sich mit den small step Regeln von $\mathcal{L}_f$ (vgl. Definition 1.15), sowie den folgenden small step Regeln für Objekte

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

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

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

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

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

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

und den small step Regeln für Reihen

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

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

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

herleiten lässt.

Die Regel (Object-Eval) wertet den Rumpf eines Objekts, also die Reihe, aus, bis ein Reihenwert erreicht ist. Anschließend ist nach Definition 2.4 (Werte und Reihenwerte) das Objekt selbst ein Wert. Die Auswertung der Reihe erfolgt mittels der Regeln (Attr-Left), (Attr-Right) und (Method-Right). Hierbei werden die rechten Seiten von Attributdeklarationen ausgewertet, und zwar in der Reihe von links nach rechts.

Die Regel (Send-Eval) wertet den Ausdruck e, dem die Nachricht m gesendet werden soll, aus. Führt diese Auswertung zu einem Objektwert **object** (*self*) ω **end**, so wird mit Regel (Send-Unfold) das Objekt aufgefaltet (wobei dann die enthaltenen Duplikationen in vollwertige Objekte umgewandelt werden) und es entsteht ein Hilfsausdruck der Form $\omega\#m$.

Für diese Hilfsausdrücke der Form $\omega\#m$ wird dann mit Hilfe der Regeln (Send-Attr), (Send-Skip) und (Send-Exec) innerhalb des Reihenwerts ω nach der letzten Deklaration der Methode m gesucht, wobei jeweils die zuvor bei der Auswertung des Objekts ermittelten Attributwerte für die frei vorkommenden Attributnamen im verbleibenden Reihenwert eingesetzt werden.

Hierbei gilt es zu beachten, dass die Regeln (Beta-V), (Unfold), (Let-Exec), (Send-Unfold) und (Send-Attr) nur angewandt werden dürfen, wenn der zu substituierende Ausdruck keine frei vorkommenden Attribut- oder Objektnamen enthält. Sollte dies nicht der Fall sein, so bleibt die Auswertung stecken.

Wir schreiben $e \to e'$ und $r \to r'$, da stets aus dem Zusammenhang hervorgeht, welche der beiden Relationen gemeint ist. Weiter schreiben wir $\xrightarrow{+}$ für den transitiven und $\xrightarrow{*}$ für den reflexiven transitiven Abschluss, 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 $e \not\to$, wenn kein e' existiert mit $e \to e'$, und $r \not\to$, wenn kein r' existiert, so dass $r \to r'$ gilt. Damit sind wir nun in der Lage, die erste einfache Eigenschaft der small step Semantik zu formulieren und zu beweisen.

Lemma 2.3 (Werte und small steps)

(a) $v \not\to$ für alle $v \in Val$.

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

Beweis: Eine einfache simultane Induktion über die Struktur von v und ω: Für Konstanten, Namen, Abstraktionen und die leere Reihe ist die Behauptung unmittelbar klar, da diese weder in den Axiomen noch in den Konklusionen der Regeln links von $\to$ stehen. Für die übrigen syntaktischen Formen von Werten folgt die Behauptung unmittelbar mit Induktionsvoraussetzung. □

Die Auswertung eines Ausdrucks beschreibt man als eine Aneinanderreihung gültiger small steps, die sich mit den small step Regeln herleiten lassen, wie nachfolgend definiert:

Definition 2.11 (Berechnung)

(a) Eine *Berechnungsfolge* ist eine endliche oder unendliche Folge von small steps $e_1 \to e_2 \to \ldots$.

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

Im Falle einer unendlichen Berechnung für einen Ausdruck sagt man auch, dass die Berechnung *divergiert*. Im Falle einer endlichen Berechnung, an deren Ende ein Wert steht, sagt man, dass die Berechnung *terminiert*. Endet eine endliche Berechnung mit einem Ausdruck, der kein Wert ist, so sagt man, dass die Berechnung *stecken bleibt*.

Damit kommen wir zum wichtigsten Satz über die small step Semantik, dem Eindeutigkeitssatz. Dieser besagt, dass auf jeden Ausdruck und jede Reihe höchstens eine small step Regel anwendbar ist. Das bedeutet insbesondere, dass für jeden Ausdruck und jede Reihe höchstens eine Berechnung existiert.

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

Beweis: Die Beweisführung erfolgt durch simultane Induktion über die Struktur von e und r, wobei wir jeweils nach der syntaktischen Form von e und r unterscheiden: Für Konstanten, Bezeichner, λ-Abstraktionen und die leere Reihe existiert nach Lemma 2.3 (Werte und small steps) kein small step. Wir betrachten exemplarisch den Fall der Applikation:

1.) $e = e_1\, e_2$

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

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

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

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

Die restlichen Fälle verlaufen analog und werden hier nicht angegeben. □

Die bereits angedeutete Eindeutigkeit von Berechnungen lässt sich nun einfach als Korollar des vorangegangenen Satzes formulieren:

Korollar 2.1 (Eindeutigkeit der Berechnung)

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

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

2.2.3 Big step Semantik

Wie bereits in Abschnitt 1.2.2 angedeutet wurde, existiert neben dem bisher vorgestellten *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 ausschließlich der rekursive Ansatz benutzt, da dieser der intuitiven Interpretation von Programmen entspricht.

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

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

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

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

(Val) $$v \Downarrow_e v$$

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

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

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

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

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

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

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

sowie den big step Regeln für Objekte

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

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

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

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

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

und den big step Regeln für Reihen

$$\text{(Omega)} \quad \omega \Downarrow_r \omega$$

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

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

herleiten lässt.

Analog zur small step Semantik schreiben wir $\Downarrow$, 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 und big step Regeln offensichtlich, so dass sich vermuten lässt, dass die beiden Semantiken das gleiche Laufzeitverhalten im Bezug auf das Ergebnis einer Berechnung aufweisen. Entsprechend gilt der folgende Satz:

Satz 2.2 (Äquivalenzsatz)

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

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

Der Begriff der Äquivalenz besagt in diesem Zusammenhang lediglich, dass eine terminierende Berechnung eines Programms mit den unterschiedlichen Semantiken zum gleichen Ergebnis führt. Über divergierende oder steckenbleibende Berechnungen lässt sich keinerlei Aussage treffen[5].

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.

[5]Grund hierfür ist, dass es mit den in Definition 2.13 angegebenen big step Regeln nicht möglich ist, zwischen Divergenz und Steckenbleiben zu unterscheiden.

„⇒“ Simultane Induktion über die Länge der Herleitungen der big steps $e \Downarrow v$ und $r \Downarrow \omega$ mit Fallunterscheidung nach der zuletzt angewandten big step Regel:

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

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

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

4.) 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 $\mathbf{object}\,(self)\ \omega\ \mathbf{end} \to \omega[^{\mathbf{object}\,(self)\ \omega\ \mathbf{end}}/_{self}]\#m$, 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

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

Die restlichen Fälle sind ebenso einfach zu beweisen.

„⇐“ Simultane Induktion über die Länge der Berechnungen $e \xrightarrow{n} v$ und $r \xrightarrow{n} \omega$ mit Fallunterscheidung nach der syntaktischen Form von e und r:

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

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

1.) Wenn $e_1, e_2 \in \mathit{Val}$, dann kommen für den ersten small step der Herleitung nur (Op) und (Beta-V) in Frage.

Für (Op) folgt die Behauptung trivialerweise mit big step Regel (Op).

Für (Beta-V) muss e_1 eine λ-Abstraktion sein, also $e_1 = \lambda x.e_1'$, und die Berechnungsfolge die Form

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

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

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

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

Die Berechnungsfolge $e_1\, e_2 \xrightarrow{*} v_1\, e_2$ kann, sofern nicht leer, ausschließlich mit small step Regel (App-Left) aus $e_1 \xrightarrow{*} v_1$ folgen. Entsprechend kann $v_1\, e_2 \xrightarrow{*} v_1\, v_2$, sofern nicht leer, nur mit small step Regel (App-Right) aus $e_2 \xrightarrow{*} v_2$ folgen.

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

Insgesamt folgt also $e_1\, e_2 \Downarrow v$ mit big step Regel (App).

3.) Für $e = \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2$ 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 $v_0 = \mathit{true}$, oder mit small step Regel (Cond-False), für $v_0 = \mathit{false}$, beginnen muss. Für $e_0 \xrightarrow{*} v_0$ existiert nach Induktionsvoraussetzung ein big step $e_0 \Downarrow v_0$.

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

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

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

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

mit big step Regel (Cond-True).

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

Die weiteren Fälle verlaufen ähnlich. □

Mit Hilfe des Äquivalenzsatzes lassen sich dann bereits bewiesene Eigenschaften der small step Semantik auf die big step Semantik übertragen[6]:

Korollar 2.2

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

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

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

[6]Natürlich nur, solange diese sich auf terminierende Berechnungen beziehen.

In Version 2.0 des Lernwerkzeugs TPML[7] findet sich eine Implementation eines Interpreters basierend auf der in diesem Abschnitt entwickelten big step Semantik.

Im weiteren Verlauf dieses Dokuments werden wir allerdings ausschließlich die für Beweise einfacher zu handhabende small step Semantik betrachten. Dieser Abschnitt sollte lediglich veranschaulichen, wie eine aus theoretischen Überlegungen entstandene Programmiersprache in eine reale Implementierung übertragen werden kann.

2.3 Ein einfaches Typsystem

Bisher haben wir nur die ungetypten Programmiersprachen $\mathcal{L}_f$ und $\mathcal{L}_o$ kennengelernt. Diese zeichnen sich durch eine einfache (kontextfreie) Syntax und eine hohe Flexibilität beim Programmieren aus. Diese hohe Flexibilität ungetypter Programmiersprachen führt jedoch auch zu erheblichen Nachteilen, da grobe Programmierfehler erst zur Laufzeit erkannt werden und sich darin äußern, dass die Auswertung stecken bleibt. Beispiele für derartige Fehler sind:

- $(\textbf{if}\, 1 > 0\, \textbf{then}\, \mathit{true}\, \textbf{else}\, 1) + 1$
- $(\textbf{object}\, (\mathit{self})\ \textbf{val}\, a = 1;\ \textbf{val}\, b = 2;\ \textbf{method}\, m = a + b\ \textbf{end})\#n$

Im ersten Beispiel würde nach Auswertung des bedingten Ausdrucks ein neuer Ausdruck $\mathit{true} + 1$ entstehen, für den aber keine small step Regel existiert. Im zweiten Beispiel wird versucht, einem Objekt, welches nur eine Methode namens m besitzt, die Nachricht n zu schicken. In solch kleinen Programmen sind derartig triviale Programmierfehler noch leicht zu entdecken, anders sieht es aber bei großen, mehreren hunderttausend Zeilen langen Programmen aus. In diesen Projekten muss ein Großteil der Entwicklungszeit auf das Testen und die Fehlerbeseitigung verwendet werden.

Um solcherlei (grobe) Programmierfehler möglichst frühzeitig erkennen zu können, verwendet man heute in fast allen gängigen Programmiersprachen *statische Typsysteme*. Dabei findet im Anschluss an die syntaktische Analyse der Programme eine Typüberprüfung statt. Ziel dieser Überprüfung ist es, Programme, in denen grobe Programmierfehler vorhanden sind, zu erkennen und abzulehnen. Ein solcher „grober Programmierfehler“ zeichnet sich im Kontext der Programmiersprache $\mathcal{L}_o$ dadurch aus, dass der Interpreter (möglicherweise) während der Auswertung des Programms stecken bleibt.

Ziel ist es also, ein statisches Typsystem anzugeben, welches alle Programme identifiziert, deren Berechnung stecken bleiben würde, und diese ausschließt. Eine Programmiersprache, welche diese Eigenschaft besitzt, bezeichnet man als *typsicher*, genauer *statisch typsicher*, nach [Pie02, S.6ff].

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

Die in diesem Abschnitt betrachtete Programmiersprache $\mathcal{L}_o^t$ ist

(a) *einfach getypt* und

(b) *explizit getypt*.

[7] `http://www.informatik.uni-siegen.de/theo/tpml/` (Stand: 19.09.2007)

Der erste Punkt besagt, dass das Typsystem dem des *einfach getypten λ-Kalküls* entspricht, also keine komplizierten Mechanismen wie rekursive Typen, Subtyping oder Polymorphie enthält. Wir werden in Kapitel 3 und 4 Erweiterungen der Programmiersprache $\mathcal{L}_o^t$ vorstellen, die Subtyping und rekursive Typen unterstützen.

Der zweite Punkt bedeutet, dass der Programmierer an bestimmten Stellen im Quelltext, zum Beispiel bei Parametern von Funktionen, Typen angeben muss. Dies entspricht der gängigen Vorgehensweise in traditionellen imperativen Programmiersprachen wie C, C++, Pascal und Java. Für explizite Typüberprüfung existiert ein einfacher und effizienter Algorithmus.

Für statisch getypte funktionale Programmiersprachen sind *explizite Typsysteme* heutzutage eher unüblich. Stattdessen benutzt man dort sogenannte *implizite Typsysteme*, in denen während der Typüberprüfung die fehlenden Typangaben automatisch durch den Typechecker rekonstruiert werden. Das Verfahren zur Rekonstruktion von Typangaben wird als *Typinferenz* oder *type reconstruction* bezeichnet.

In dieser Arbeit werden ausschließlich explizit getypte Programmiersprachen betrachtet. Erläuterungen zu impliziten Typsystemen finden sich aber in [Pie02, S.317ff]. Eine Darstellungen der Problematik von Typinferenz in Typsystemen mit Subtyping und rekursiven Typen findet sich in [PWO97] sowie [JP99].

Zunächst wollen wir festlegen, was genau unter einem Typ zu verstehen ist. Da die Syntax der Programmiersprache aus Ausdrücken und Reihen besteht, definieren wir entsprechend zwei Mengen von Typen, eine Menge als Typen für Ausdrücke und eine weitere für Reihen[8].

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

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

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

$$\begin{array}{rcll} \phi & ::= & \emptyset & \text{leerer Reihentyp} \\ & \mid & m : \tau;\ \phi_1 & \text{Methodentyp} \end{array}$$

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

Da wir ein explizites Typsystem definieren wollen, muss die kontextfreie Grammatik von Ausdrücken erweitert werden, indem die Produktionen für Abstraktionen, Rekursionen und Objekte durch die „getypten Produktionen“

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

ersetzt werden.

[8] Diese strikte Unterscheidung ist nicht zwingend notwendig, wir werden sie aber beibehalten in der Hoffnung, dadurch ein besseres Verständnis zu vermitteln.

Konvention 2.2 Die Reihenfolge, in der die Methodentypen in einem Reihentypen aufgelistet werden, ist irrelevant, d.h. es gilt

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

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

Diese Vereinbarung ist nicht zwingend notwendig, und teilweise sogar unüblich, erleichtert jedoch die nachfolgenden Definitionen erheblich, da es von nun an keine Rolle mehr spielt, wo innerhalb eines Reihentyps der Typ einer konkreten Methode steht.

Darüber hinaus definieren wir auf Reihentypen, ähnlich wie zuvor auf Reihen, die Vereinigung (oder Konkatenation):

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

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

und

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

ist definiert als

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

mit

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

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

Insbesondere führt die Vereinigung zweier inkompatibler Reihentypen, die auf dem Schnitt ihrer Methodentypen nicht übereinstimmen, während der Typüberprüfung zu einem Typfehler, da in diesem Fall kein Vereinigungstyp definiert ist.

Definition 2.16 Seien $\phi_1 = m_1 : \tau_1; \ldots m_n : \tau_n; \emptyset$ und $\phi_2 = m'_1 : \tau'_1; \ldots m'_l : \tau'_l; \emptyset$. Die Reihentypen ϕ_1 und ϕ_2 heissen *disjunkt*, wenn $\{m_1, \ldots, m_n\} \cap \{m'_1, \ldots, m'_l\} = \emptyset$ gilt.

Syntaktischer Zucker

Den zuvor für die Programmiersprachen $\mathcal{L}_f$ und $\mathcal{L}_o$ definierten syntaktischen Zucker versieht man dann für die Programmiersprache $\mathcal{L}_o^t$ entsprechend mit Typen:

$$\begin{array}{ll} & \textbf{let}\, x\, (x_1 : \tau_1) \ldots (x_n : \tau_n) = e_1\, \textbf{in}\, e_2 \\ \text{für} & \textbf{let}\, x = \lambda x_1 : \tau_1. \ldots \lambda x_n : \tau_n . e_1\, \textbf{in}\, e_2 \end{array}$$

$$\begin{array}{ll} & \textbf{let rec}\, x\, (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau = e_1\, \textbf{in}\, e_2 \\ \text{für} & \textbf{let}\, x = \textbf{rec}\, x : \tau_1 \to \ldots \to \tau_n \to \tau . \lambda x_1 : \tau_1. \ldots \lambda x_n : \tau_n . e_1\, \textbf{in}\, e_2 \end{array}$$

$$\begin{array}{ll} & \textbf{method}\ m\ (x_1 : \tau_1) \ldots (x_n : \tau_n) = e;\ r \\ \text{für} & \textbf{method}\ m = \lambda x_1 : \tau_1. \ldots \lambda x_n : \tau_n . e;\ r \end{array}$$

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

Bisher haben wir nur die kontextfreie Grammatik der (statisch getypten) Programmiersprache $\mathcal{L}_o^t$ definiert. Jetzt müssen wir die Kontextregeln festlegen, unter denen ein Programm akzeptiert werden soll. Beispiele für derartige Kontextbedingungen sind:

- Jeder Bezeichner muss deklariert werden, bevor er verwendet werden kann.
- Nachrichten können nur an Objekte gesendet werden, die über eine gleichnamige Methode verfügen.

Die erste Bedingung ließe sich schon mit den bisherigen Mitteln sehr einfach formulieren: Jedes Programm muss abgeschlossen sein. Die zweite Bedingung ist deutlich aufwändiger zu prüfen, da das Senden von Nachrichten an beliebige Ausdrücke möglich ist, und ggfs. erst zur Laufzeit bekannt ist, an welches Objekt die Nachricht wirklich gesendet werden soll.

Zunächst betrachten wir einen einfachen Kalkül mit dem wir sogenannte *Typurteile für Konstanten* herleiten können. Damit wird jeder Konstanten in einem Programm ein eindeutiger Typ zugeordnet.

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

(Bool)	$b :: \mathbf{bool}$	
(Int)	$n :: \mathbf{int}$	
(Unit)	$() :: \mathbf{unit}$	
(Aop)	$op :: \mathbf{int} \to \mathbf{int} \to \mathbf{int}$	falls $op \in \{+, -, *\}$
(Rop)	$op :: \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$	falls $op \in \{<, >, \leq, \geq, =\}$

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

Als nächstes wollen wir Ausdrücken einen Typ zuordnen. Der Typ eines Ausdrucks soll sich dabei aus den Typen der unmittelbaren Teilausdrücke (und Teilreihen) herleiten lassen, d.h. wir müssen auch nicht abgeschlossene Ausdrücke betrachten. Dazu müssen jeweils die Typen der frei vorkommenden Namen bekannt sein.

Jeder Name erhält seinen Typ aus der Umgebung, also dem formalen Parameter, an den er gebunden ist. Wir benötigen daher einen Mechanismus zur Buchführung über Typen bereits bekannter Namen, eine sogenannte *Typumgebung*.

Definition 2.18 (Typumgebungen)

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

(b) Sei Γ eine Typumgebung, dann bezeichnet $\Gamma^\star$ die Einschränkung von Γ auf Variablen, d.h. für die Typumgebung $\Gamma^\star$ gilt:

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

(c) Sei Γ eine Typumgebung, dann bezeichnet Γ^+ die Einschränkung von Γ auf alle Namen ausser Variablen, d.h. für die Typumgebung Γ^+ gilt:

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

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

Zur einfacheren Notation von Typumgebungen verwenden wir die Listenschreibweise $[id_1 : \tau_1, \ldots, id_n : \tau_n]$ mit $n \geq 0$ für die Typumgebung Γ mit den Eigenschaften

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

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

Die Listenschreibweise verdeutlicht, dass eine Typumgebung letztlich nichts anderes ist als eine „Tabelle“, in der die Typen der bereits bekannten Namen eingetragen sind[9]. 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, der für den Namen id in der Tabelle Γ eingetragen wurde, 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)

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

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

Durch $(\_ \rhd_e \_ :: \_)$ und $(\_ \rhd_r \_ :: \_)$ sind also dreistellige Relationen definiert, genauer

$$(\_ \rhd_e \_ :: \_) \subseteq TEnv \times Exp \times Type$$

und

$$(\_ \rhd_r \_ :: \_) \subseteq TEnv \times Row \times RType.$$

Die eigentliche Typrelation ist definiert als Vereinigung dieser beiden Typrelationen, d.h.

$$(\_ \rhd \_ :: \_) = (\_ \rhd_e \_ :: \_) \cup (\_ \rhd_r \_ :: \_).$$

Wie zuvor bei der Definition der small step Semantik ist es üblich, die Typrelationen nur implizit zu betrachten, und stattdessen stets über „Formeln“ zu argumentieren.

Zur Herleitung gültiger Formeln definieren wir induktiv einen Kalkül, der sowohl Typurteile für Ausdrücke, wie auch Typurteile für Reihen umfasst. Die Typrelationen sind dann die kleinsten Relationen, die unter Anwendung dieser Regeln abgeschlossen sind.

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

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

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

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

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

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

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

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

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

sowie den Typregeln für Objekte

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

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

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

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

und den Typregeln für Reihen

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

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

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

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

Die Typregeln für die funktionale Programmiersprache entsprechen den aus der Vorlesung „Theorie der Programmierung“ (vgl. [Sie06]) bekannten Typregeln und sollten weitestgehend selbsterklärend sein. Beispielsweise sichert (Id), dass Namen deklariert werden müssen bevor sie verwendet werden, indem gefordert wird, dass ein innerhalb eines Ausdrucks vorkommender Name zuvor in die Typumgebung eingetragen worden

sein muss.

Ein derartiger Eintrag wird beispielsweise durch (ABSTR) oder (LET) vorgenommen. Die (ABSTR)-Regel trägt hierbei den durch den Programmierer spezifizierten Typ für den formalen Paramter in die Typumgebung ein und prüft, ob sich damit für den Funktionsrumpf ein gültiger Typ herleiten lässt.

Die neuen Typregeln für Objekte und Reihen sind, basierend auf der im vorangegangenen Abschnitt dargestellten Semantik, ebenfalls weitestgehend selbsterklärend. Hervorzuheben sind lediglich die Regeln (OBJECT), (DUPL) und (ATTR), welche kurz erläutert werden sollen:

Bei (OBJECT) wird zunächst der Typ der enthaltenen Reihe bestimmt. Für diese Typüberprüfung werden zunächst sämtliche Objekt- und Attributnamen (von umgebenden Objekten) aus der Typumgebung entfernt, und anschließend der durch den Programmierer vorgegebene Objekttyp für *self* eingetragen. Der damit bestimmte Reihentyp enthält dann die Typen der Methoden und muss exakt mit dem angegebenen Objekttyp übereinstimmen.

Für Attributdeklarationen wird mit (ATTR) zunächst der Typ des Ausdrucks auf der rechten Seiten der Deklaration bestimmt, wobei alle Objekt- und Attributnamen aus der Typumgebung entfernt werden, da die Ausdrücke auf den rechten Seiten vor dem Auffalten des Objekts ausgewertet werden und somit deren Berechnung stecken bleiben könnte, wenn hier ein Attribut- oder Objektname vorkäme. Anschließend wird der so bestimmte Attributtyp in die Typumgebung eingetragen, in der dann ein Reihentyp für die Restreihe bestimmt wird.

Die erste Prämisse der (DUPL)-Regel bestimmt zunächst den Typ von *self*, womit sichergestellt ist, dass Duplikationen nur innerhalb von Objekten vorkommen. Dann wird überprüft, ob für jedes angegebene Attribut ein Typ in der Typumgebung vorhanden ist, der zuvor durch (ATTR) eingetragen worden sein muss, und ob sich dieser Typ auch für den jeweiligen Ausdruck herleiten lässt. Sind diese Bedingungen erfüllt, so lässt sich für die Duplikation der Typ des umgebenden Objekts herleiten.

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

Bei aufmerksamer Betrachtung der Typregeln aus Definition 2.20 ist intuitiv sofort ersichtlich, dass für jede syntaktische Form eines Ausdrucks oder einer Reihe immer nur genau eine Typregel in Frage kommt. Folglich ist das Typsystem der Programmiersprache $\mathcal{L}_o^t$ deterministisch, d.h. es gilt der Satz:

Satz 2.3 (Typeindeutigkeit) *Sei Γ eine Typumgebung, dann gilt:*

(a) Für jeden Ausdruck $e \in Exp$ existiert höchstens ein Typ $\tau \in Type$ mit $\Gamma \rhd e :: \tau$.

(b) Für jede Reihe $r \in Row$ existiert höchstens ein Reihentyp $\phi \in RType$ mit $\Gamma \rhd r :: \phi$.

Beweis: Einfache simultane Induktion über die Struktur von e und r. □

Definition 2.21 (Wohlgetyptheit)

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

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

Abgeschlossene, wohlgetypte Ausdrücke werden auch als *Programme* bezeichnet.

Korollar 2.3 *Sei* Γ *eine Typumgebung, dann gilt:*

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

(b) Ist $r \in \mathit{Row}$ *wohlgetypt in* Γ*, dann existiert genau ein* $\phi \in \mathit{RType}$ *mit* $\Gamma \rhd e :: \phi$.

Basierend auf dieser Erkenntnis lässt sich nun ein Algorithmus formulieren, welcher überprüft, ob ein Ausdruck in einer gegebenen Typumgebung wohlgetypt ist, und falls ja, den eindeutigen Typ für diesen Ausdruck in der Typumgebung liefert.

Algorithmus 2.1 (Typalgorithmus für $\mathcal{L}_o^t$) Als Eingabe erhält der Algorithmus eine Typumgebung $\Gamma \in \mathit{TEnv}$, sowie einen Ausdruck $e \in \mathit{Exp}$ oder eine Reihe $r \in \mathit{Row}$. Als Ausgabe liefert der Algorithmus entsprechend entweder einen Typ $\tau \in \mathit{Type}$ oder einen Reihentyp $\phi \in \mathit{RType}$, oder – falls keine Typherleitung existiert – einen Typfehler.

1. Wenn $e \in \mathit{Const}$, dann liefere den eindeutigen Typ der Konstanten (also z.B. $\tau = \mathbf{int}$ für $e \in \mathit{Int}$).

2. Wenn $e = \mathit{id} \in \mathit{Id}$, dann prüfe ob $\mathit{id} \in \mathit{dom}\,(\Gamma)$, und falls ja, liefere $\tau = \Gamma(\mathit{id})$, sonst „Typfehler“.

3. Wenn $e = e_1\, e_2$, dann bestimme rekursiv den Typ von e_1 und e_2, jeweils in Γ, und falls einer der rekursiven Aufrufe in einem Typfehler resultiert, brich ab mit „Typfehler“. Falls der Typ $\tau_2 \to \tau$ der Typ von e_1 und τ_2 der Typ von e_2 sein sollte, liefere τ als Ergebnis, sonst brich ab mit „Typfehler“.

4. Wenn $e = \lambda x : \tau_2.e_1$, dann bestimme rekursiv den Typ von e_1 in der Typumgebung $\Gamma[{}^{\tau_2}/_x]$, und falls dies zum Ergebnis τ_2 führt, liefere insgesamt $\tau_2 \to \tau_1$ als Typ. Sonst antworte mit „Typfehler“.

Die übrigen Fälle des Algorithmus überlassen wir der Fantasie des Lesers. Die prinzipielle Umsetzung der Typregeln aus Definition 2.20 (Gültige Typurteile für $\mathcal{L}_o^t$) sollte anhand der angegebenen Fälle klar sein.

2.3.3 Typsicherheit

Nach diesen einleitenden Betrachtungen wollen wir zeigen, dass die Programmiersprache $\mathcal{L}_o^t$ typsicher ist. Typsicherheit bedeutet in diesem Zusammenhang, dass die Berechnung eines wohlgetypten abgeschlossenen Ausdrucks nicht steckenbleibt. Das Typsystem soll also möglichst alle Ausdrücke identifizieren, deren Berechnung stecken bleiben würde.

Wichtig zu bemerken ist hierbei, dass durch ein Typsystem, welches ausschließlich auf syntaktischen Informationen arbeitet[10], keine strikte Trennung zwischen Programmen, deren Berechnung stecken bleibt, und solchen, die mit einem Wert terminieren

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

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 damit abfinden, dass ein Typsystem für eine Turing-vollständige Programmiersprache immer auch Programme ablehnen wird, deren Berechnung nicht stecken bleiben würde. Deshalb ist man bemüht, Typsysteme immer weiter zu verbessern, so dass immer weniger korrekte Programme abgelehnt werden. Wichtig für unsere Betrachtung ist jedoch nur, dass das Typsystem definitiv alle Programme identifiziert, die stecken bleiben würden.

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

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

Die small step Regeln aus $\mathcal{L}_o$ werden übernommen, wobei die Regeln (Beta-V), (Unfold), (Object-Eval) und (Send-Unfold) an die neue Syntax angepasst werden müssen:

$$\begin{array}{ll} (\text{Beta-V}) & (\lambda x : \tau.e)\, v \to_e e[{}^{v}/_{x}] \\ (\text{Unfold}) & \textbf{rec}\, x : \tau.e \to_e e[{}^{\textbf{rec}\, x:\tau.e}/_{x}] \\ (\text{Object-Eval}) & \dfrac{r \to_r r'}{\textbf{object}\,(\mathit{self} : \tau)\ r\ \textbf{end} \to_e \textbf{object}\,(\mathit{self} : \tau)\ r'\ \textbf{end}} \\ (\text{Send-Unfold}) & \textbf{object}\,(\mathit{self} : \tau)\ \omega\ \textbf{end}\#m \to_e \omega[{}^{\textbf{object}\,(\mathit{self}:\tau)\ \omega\ \textbf{end}}/_{\mathit{self}}]\#m \end{array}$$

Es gilt zu beachten, dass der Typ in diesen Ausdrücken keinerlei Einfluss auf den small step hat, d.h. Typen spielen zur Laufzeit keine Rolle. Sie werden lediglich zur Compilezeit während der statischen Typüberprüfung benötigt. D.h. wir müssen nicht zwischen den operationellen Semantiken der Programmiersprachen $\mathcal{L}_o$ und $\mathcal{L}_o^t$ unterscheiden.

Zum Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_o^t$ gehen wir wie in [Pie02] beschrieben vor, indem wir zeigen, dass jeder small step typerhaltend ist – das Preservation-Theorem – und dass wohlgetypte, abgeschlossene Ausdrücke entweder bereits Werte sind oder ein small step existiert – das Progress-Theorem.

Wir beginnen mit dem Preservation-Theorem. Dazu benötigen wir zunächst einige technische Lemmata.

Definition 2.22 Seien $\Gamma_1, \Gamma_2 \in \mathit{TEnv}$ und $A \subseteq \mathit{Id}$. Γ_1 und Γ_2 *stimmen überein auf* A, wenn gilt:

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

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

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

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

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

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

Beweis: Trivial. □

Zwischen den freien Bezeichnern eines Ausdrucks und den Einträgen in der Typumgebung eines Typurteils besteht offensichtlich ein elementarer Zusammenhang: In der Typumgebung müssen zumindest Einträge für alle frei vorkommenden Bezeichner des Ausdrucks enthalten sein.

Lemma 2.5 (Typumgebungen und frei vorkommende Namen)

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

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

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

1.) $\Gamma \rhd \mathbf{object}(self : \tau)\ r\ \mathbf{end} :: \tau$ mit Typregel (OBJECT) kann ausschließlich aus $\Gamma^\star[^\tau/_{self}] \rhd r :: \phi$ und $\tau = \langle\phi\rangle$ folgen. Nach Induktionsvoraussetzung gilt $free(r) \subseteq dom(\Gamma^\star[^\tau/_{self}])$ und wegen $dom(\Gamma^\star[^\tau/_{self}]) \subseteq dom(\Gamma) \cup \{self\}$ folgt $free(r) \setminus \{self\} \subseteq dom(\Gamma)$. Vermöge Definition 2.5 (Frei vorkommende Namen) gilt folglich $free(\mathbf{object}(self : \tau)\ r\ \mathbf{end}) \subseteq dom(\Gamma)$.

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

Die restlichen Fälle verlaufen analog. □

Insbesondere gilt, dass Ausdrücke, welche in einer Typumgebung $\Gamma^\star$ wohlgetypt sind, keine freien Vorkommen von Attribut- oder Objektnamen enthalten können, was nachfolgend als Korollar formuliert ist:

Korollar 2.4 $\forall \Gamma \in TEnv, e \in Exp, \tau \in Type : \Gamma^\star \rhd e :: \tau \Rightarrow e \in Exp^\star$

Offensichtlich hängt der Typ eines Ausdrucks oder einer Reihe nur von den Typen der frei vorkommenden Namen in der Typumgebung ab. Die übrigen Einträge in einer Typumgebung spielen für die Herleitung des Typurteils keine Rolle. Diese Aussage formalisieren wir im nächsten Lemma:

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

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

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

D.h. Typumgebungen, die auf den frei vorkommenden Namen eines Ausdrucks oder einer Reihe übereinstimmen, sind bzgl. der Typurteile für diesen Ausdruck oder diese Reihe austauschbar.

Beweis: Leicht durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma_1 \triangleright e :: \tau$ und $\Gamma_1 \triangleright r :: \phi$ zu beweisen, wobei jeweils nach der zuletzt angewandten Typregel unterschieden wird. Für (Id) gilt die Behauptung trivialerweise wegen Definition 2.22. Die Fälle im Induktionsschritt folgen jeweils unmittelbar mit Induktionsvoraussetzung. □

Korollar 2.5

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

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

Beweis: Folgt mit Lemma 2.5 unmittelbar aus dem Koinzidenzlemma. □

Weiterhin benötigen wir zwei Substitutionslemmata, die einen Zusammenhang zwischen dem Eintragen in Typumgebungen und syntaktischer Substitution auf Ausdrücken oder Reihen herstellen. Es sind zwei Substitutionslemmata notwendig, da – wie in Abschnitt 2.2.1 erläutert – die Substitution für *self* bereits Aspekte der Semantik enthält.

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

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

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

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

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

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

Für $\mathit{id} = \mathit{id}'$ gilt nach Definition 2.18 (Typumgebungen) $\tau = \tau'$. Nach Definition der Substitution gilt weiter $\mathit{id}'[{}^{e}/_{\mathit{id}}] = e$ und aus $\mathit{free}(e) \subseteq \mathit{dom}(\Gamma^\star) \subseteq \mathit{dom}(\Gamma)$ folgt dann mit Lemma 2.6 (Koinzidenzlemma) $\Gamma \triangleright \mathit{id}'[{}^{e}/_{\mathit{id}}] :: \tau'$.

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

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

Dann gilt nach Voraussetzung $\Gamma[{}^{\tau}/_{\mathit{id}}] \triangleright e_1 :: \tau'' \to \tau'$ und $\Gamma[{}^{\tau}/_{\mathit{id}}] \triangleright e_2 :: \tau''$. Mit Induktionsvoraussetzung folgt

$$\Gamma \triangleright e_1[{}^{e}/_{\mathit{id}}] :: \tau'' \to \tau'$$

und

$$\Gamma \triangleright e_2[^{e}/_{id}] :: \tau'',$$

mit Typregel (App) also $\Gamma \triangleright e_1[^{e}/_{id}]\, e_2[^{e}/_{id}] :: \tau'$ und wegen Definition 2.8 folgt schließlich

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

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

Das Typurteil kann nur aus der Prämisse $\Gamma[^{\tau}/_{id}][^{\tau'}/_{x}] \triangleright e' :: \tau''$ folgen. Durch gebundene Umbenennung lässt sich dann die Voraussetzung $x \notin \{id\} \cup \mathit{free}\,(e)$ für die Substitution herstellen. Wegen $x \neq id$ folgt dann $\Gamma[^{\tau'}/_{x}][^{\tau}/_{id}] \triangleright e' :: \tau''$ und nach Induktionsvoraussetzung gilt somit

$$\Gamma[^{\tau'}/_{x}] \triangleright e'[^{e}/_{id}] :: \tau''.$$

Mit Typregel (Abstr) folgt daraus $\Gamma \triangleright \lambda x : \tau'.e'[^{e}/_{id}] :: \tau' \to \tau''$ und wegen $id \notin \{id\} \cup \mathit{free}\,(e)$ mit Definition 2.8 schließlich

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

Die übrigen Fälle verlaufen ähnlich zu den angegebenen. □

Die *self*-Substitution ist ein Spezialfall der allgemeinen Substitution und muss separat betrachtet werden. Für den Fall der Substitution von *self* in Duplikationen wird die Reiheneinsetzung benötigt (vgl. Definition 2.8). Entsprechend müssen wir zunächst zeigen, dass die Reiheneinsetzung typerhaltend ist.

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

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

Beweis: Wegen $\Gamma^{\star} \triangleright e :: \tau$ ist nach Lemma 2.5 (Typumgebungen und frei vorkommende Namen) $e \in \mathit{Exp}^{\star}$, und wegen $\Gamma^{\star} \triangleright r(a) :: \tau$ ist $a \in \mathit{dom}_a\,(r)$. Also ist die Reiheneinsetzung $r\langle^{e}/_{a}\rangle$ gemäss den Voraussetzungen stets definiert. Damit können wir dann den Beweis durch Induktion über die Struktur von r führen. Wir beschränken uns an dieser Stelle darauf, zu zeigen, dass Reiheneinsetzung für Attributdeklarationen typerhaltend ist:

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

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

Im Fall von $a \neq a'$ ist $(\mathbf{val}\ a' = e';\ r')\langle^{e}/_{a}\rangle = (\mathbf{val}\ a' = e';\ r'\langle^{e}/_{a}\rangle)$ und somit $(\mathbf{val}\ a' = e';\ r')(a) = r'(a)$. Wegen $(\Gamma[^{\tau'}/_{a'}])^{\star} = \Gamma^{\star}$ 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 $\Gamma \triangleright (\mathbf{val}\ a' = e';\ r')\langle^{e}/_{a}\rangle :: \phi$.

Für Methodendeklarationen ist die Behauptung einfach zu zeigen, und die leere Reihe muss überhaupt nicht betrachtet werden, denn es gilt $a \notin dom_a(\epsilon) = \emptyset$. □

Damit können wir nun das Substitutionslemma für *self* formulieren, als letzten Baustein zum Beweis des Preservation-Theorems. Im Gegensatz zum allgemeinen Substitutionslemma (Lemma 2.7) benötigen wir in diesem Fall mehr Prämissen, um zu zeigen, dass die bei der *self*-Substitution unter Duplikationen erzeugten Objekte typgleich zu den Duplikationen sind.

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

(a) Wenn für $e \in Exp$ *und* $\tau' \in Type$

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

gilt, dann gilt auch $\Gamma \triangleright e[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau'$.

(b) Wenn für $r' \in Row$ *und* $\phi \in RType$

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

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

Die ersten beiden Prämissen entsprechen jeweils den Voraussetzungen, die auch für das allgemeine Substitutionslemma benötigt werden, wobei im zweiten Fall die Reihe r' Suffix der Reihe des zu substituierenden Objekts sein muss. Die dritte Prämisse bestimmt, dass alle in der Typumgebung vorhandenen Attributnamen in der Reihe des Objekts vorkommen und die in der Typumgebung eingetragenen Typen sich auch für die Ausdrücke der Attribute herleiten lassen müssen.

Beweis: Wegen Lemma 2.5 gilt wieder, dass die Substitution stets definiert ist. Den Beweis führen wir wie üblich 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 angewandten Typregel. Dazu betrachten wir exemplarisch die folgenden Fälle:

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

Nach Voraussetzung gilt $id \in dom(\Gamma[^{\tau}/_{self}])$ 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 $\Gamma^{\star} \triangleright id[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau'$ 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 $self \notin free(id)$. Es folgt unmittelbar $\Gamma \triangleright id[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau'$.

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

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

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

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

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

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

Die restlichen Fälle verlaufen ähnlich. □

Nun können wir endlich daran gehen, das *Preservation-Theorem* zu formulieren, um so die erste Voraussetzung für die Typsicherheit der Programmiersprache $\mathcal{L}_o^t$ zu sichern: Die Wohlgetyptheit und der Typ eines Ausdrucks oder einer Reihe bleibt über small steps hinweg erhalten. Oder anders ausgedrückt: Wenn sich für einen Ausdruck e in einer Typumgebung Γ der Typ τ herleiten lässt, und für diesen Ausdruck ein small step $e \to e'$ existiert, dann lässt sich auch für e' in der Typumgebung Γ der Typ τ herleiten.

Satz 2.4 (Typerhaltung, „Preservation“)

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

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

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

Beweis: Die Behauptung beweisen wir durch simultane Induktion über die Länge der Herleitung der small steps $e \to e'$ und $r \to r'$, wobei wir lediglich die folgenden Fälle betrachten:

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

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

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

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

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

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

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

4.) $\mathbf{val}\ a = e;\ r \to \mathbf{val}\ a = e';\ r$ mit (Attr-Eval) aus $e \to e'$.

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

5.) $(\mathbf{val}\ a = v;\ \omega)\#m \to \omega[{}^{v}/_{a}]\#m$ mit (Send-Attr).

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

6.) $(\mathbf{method}\ m = e;\ \omega)\#m \to e$ mit (Send-Exec).

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

Die übrigen Fälle verlaufen analog. □

Korollar 2.6 $\forall e, e' \in Exp : \forall \tau \in Type : [\,] \triangleright e :: \tau \wedge e \to e' \ \Rightarrow\ [\,] \triangleright e' :: \tau$

Zum Beweis der Typsicherheit bleibt dann abschließend noch zu zeigen, dass abgeschlossene, wohlgetypte Ausdrücke entweder bereits Werte sind oder sich weiter auswerten lassen, das sogenannte *Progress-Theorem*. Hierzu formulieren wir zunächst ein *Canonical Forms Lemma*, welches, basierend auf dem Typ eines abgeschlossenen, wohlgetypten Wertes, Rückschluss auf die Form des Wertes ermöglicht.

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 $[\,] \triangleright v :: \tau \to \tau'$ 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 2.5 (Existenz des Übergangsschritts, „Progress“)

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

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

Wie bereits angedeutet, besagt dieser Satz, dass ein abgeschlossener, wohlgetypter Ausdruck entweder bereits ein Wert ist, oder für diesen Ausdruck ein small step existiert. Wegen Lemma 2.3 (Werte und small steps) ist auch klar, dass sich diese beiden Alternativen gegenseitig ausschließen.

Entsprechendes gilt für Reihen, wobei hier allerdings keine Abgeschlossenheit für die Reihe gefordert wird, sondern lediglich keine freien Variablen. Im Beweis wird im Fall von Objekten klar, warum der Satz sonst nicht zu beweisen wäre.

Beweis: Wir führen den Beweis durch simultane Induktion über die Struktur von e und r, mit 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 $[\,] \triangleright e_1 :: \tau' \to \tau$ und $[\,] \triangleright e_2 :: \tau'$ folgen.

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

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

Es bleibt der Fall $e_1, e_2 \in Val$. Wegen $[\,] \triangleright e_1 :: \tau' \to \tau$ 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 offensichtlich, dass op nur vom Typ $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$ oder $\mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ sein kann. Mit dem Canonical Forms Lemma (Lemma 2.10) folgt aus $[\,] \triangleright v_1 :: \mathbf{int}$ und $[\,] \triangleright e_2 :: \mathbf{int}$, dass $v_1, e_2 \in \mathit{Int}$ gilt, und somit existiert ein small step $op\, v_1\, e_2 \to op^I(v_1, e_2)$ mit Regel (Op).

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

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

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

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

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

Nach Induktionsvoraussetzung gilt also entweder $r \in \mathit{RVal}$ oder es existiert ein $r' \in \mathit{Row}$ mit $r \to r'$. Im ersten Fall ist dann das Objekt selbst schon ein Wert, im zweiten Fall existiert ein small step mit Regel (Object-Eval).

Die restlichen Fälle verlaufen analog. □

Wie zuvor bereits erwähnt, folgt dann aus der Typerhaltung und der Existenz des Übergangsschrittes die Typsicherheit der Programmiersprache $\mathcal{L}_o^t$. Entsprechend gilt der folgende Satz:

Satz 2.6 (Typsicherheit, „Safety“) *Wenn $[\,] \triangleright e :: \tau$, dann bleibt die Berechnung für e nicht stecken.*

Beweis: Wir führen den Beweis indirekt. Sei dazu angenommen, die Berechnung für e bleibe stecken. Dann ist die Berechnung von der Form $e = e_0 \to \ldots \to e_n$ mit $e_n \not\to$ und $e_n \notin \mathit{Val}$ für ein $n \geq 0$. Wegen $[\,] \triangleright e :: \tau$ folgt mit Satz 2.4 (Preservation) induktiv $[\,] \triangleright e_i :: \tau$ für alle $i = 1, \ldots, n$. Also gilt insbesondere $[\,] \triangleright e_n :: \tau$, aber $e_n \notin \mathit{Val}$ und $e_n \not\to$. Dies steht im Widerspruch zu Satz 2.5 (Progress). □

Korollar 2.7 *Sei $e \in \mathit{Exp}$ und $\tau \in \mathit{Type}$. Wenn $[\,] \triangleright e :: \tau$, dann gilt genau eine der folgenden Aussagen.*

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

(b) Die Berechnung von e divergiert.

Da wir abgeschlossene, wohlgetypte Ausdrücke auch als Programme bezeichnen, könnten wir die Aussage des vorangegangenen Satzes auch durch den Slogan

Die Berechnung für ein Programm bleibt nicht stecken

ausdrücken (engl.: *well typed programs don't go wrong*).

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 letztere Bezeichnung eigentlich falsch ist, da es sich beim Subtyping lediglich um eine mögliche Form der Polymorphie handelt[1].

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

3.1 Motivation

Das Typsystem der Programmiersprache $\mathcal{L}_o^t$ ist erwiesenermaßen typsicher, d.h. es identifiziert alle Programme, deren Berechnung stecken bleiben würde. Andererseits lehnt es darüber hinaus aber auch noch viele Programme ab, deren Berechnung nicht stecken bleiben würde. Betrachten wir beispielsweise die folgende Funktion:

$$send_m = \lambda o : \langle m : \mathbf{int} \rangle . o\#m + 1$$

Offensichtlich sendet diese Funktion dem Objekt, welches für den formalen Parameter eingesetzt wird, die Nachricht m und addiert eins auf das Ergebnis des Methodenaufrufs. Rein semantisch könnte diese Funktion auf jedes Objekt angewendet werden, welches eine Methode m besitzt, die einen ganzzahligen Wert als Ergebnis liefert.

Aus Sicht des Typsystems jedoch, kann die Funktion $send_m$ ausschließlich auf Objekte angewendet werden, die exakt eine Methode names m vom Typ **int** besitzen. Beispielsweise lässt sich für den Ausdruck

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

im Typsystem der Sprache $\mathcal{L}_o^t$ kein Typ herleiten. Das Typsystem schränkt demzufolge die Programmiersprache dadurch unnötig ein, dass die Funktion $send_m$ nur auf Objekte eines bestimmten Typs anwendbar ist. Tatsächlich kann $send_m$ jedoch auf Objekte unterschiedlicher Typen angewandt werden, ohne dass die Berechnung stecken bleibt. Dies sind genau diejenigen Objekte, welche eine Methode m vom Typ **int** besitzen.

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

Das Typsystem sollte also vorsehen, dass für die Funktion $send_m$ neben dem offensichtlichen Typ $\langle m : \mathbf{int} \rangle \to \mathbf{int}$ auch der Typ $\langle m : \mathbf{int}; n : \mathbf{bool} \rangle \to \mathbf{int}$ hergeleitet werden kann. Alternativ sollte man für das Objekt neben dem Typ $\langle m : \mathbf{int}; n : \mathbf{bool} \rangle$ auch den Typ $\langle m : \mathbf{int} \rangle$ herleiten können. Ein Typsystem, welches vorsieht, dass für den gleichen Ausdruck unterschiedliche Typen herleitbar sind, bezeichnet man als *polymorph*.

Die grundsätzliche Idee an dieser Stelle ist wie folgt: Die Funktion $send_m$ ist auf jedes Objekt anwendbar, welches sich „mindestens so gut“ verhält, wie ein Objekt, welches eine Methode m besitzt, die einen ganzzahligen Wert liefert (oder divergiert). Diese Art der Polymorphie wird als *Subtyp-Polymorphie* bezeichnet.

3.2 Die Subtyprelation

In diesem Abschnitt definieren wir eine sogenannte *Subtyprelation*, welche genau die Paare (τ, τ') enthält, für die gilt, dass sich ein Ausdruck vom Typ τ in jedem Kontext „mindestens so gut“ verhält, wie ein Ausdruck vom Typ τ'.

Um zu entscheiden, ob ein Objekt anstelle eines Objekts anderen Typs verwendet werden kann, wird in den gängigen Programmiersprachen wie Java oder C++ die sich aus dem *Subclassing* ergebende Vererbungshierarchie benutzt (vgl. Kapitel 5). Nach [AC96, S.17f] beschränkt sich die Definition der Subtyprelation dann auf die einfache Regel:

> Ist c' eine Subklasse von c, und o' eine Instanz von c', dann ist o' ebenfalls eine Instanz von c.

Für reines Subtyping auf Objekttypen, ohne Vererbung, ist diese Regel jedoch ungeeignet, insbesondere, da diese Regel für *structural typing* nicht anwendbar ist. Stattdessen verwenden wir in diesem Dokument das sogenannte *structural subtyping*, bei dem Subtyping dynamisch auf der Struktur der Typen entschieden wird, statt durch Typnamen in einer statischen (Vererbungs-)Hierarchie (vgl. [AC96, S.25ff]).

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

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

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

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

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

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

Eine Formel der Gestalt $\tau_1 \leq \tau_2$ liest man als „τ_1 ist Subtyp von τ_2“ oder „τ_1 ist kleiner als τ_2“. „Kleiner“ bedeutet in diesem Zusammenhang, dass τ_1 mehr Typinformationen enthält als τ_2. Intuitiv heißt das, dass an jeder Stelle, an der ein Ausdruck vom Typ τ_2 erwartet wird, auch ein Ausdruck vom Typ τ_1 eingesetzt werden kann.

Diese Tatsache ist insbesondere bei Objekttypen sofort ersichtlich, denn ein kleinerer Objekttyp enthält nach Regel (S-Obj-Width), der sogenannten *Breitenregel*, mindestens alle Methoden des größeren Typs, sowie beliebig viele weitere. Die *Tiefenregel* (S-Obj-Depth) besagt darüber hinaus, dass auch die Typen der einzelnen Methoden kleiner sein können.

Zu beachten ist weiterhin, dass vermöge Regel (S-Arrow) die Subtyprelation für Funktionstypen kontravariant auf den Parametertypen und kovariant in den Ergebnistypen ist. Dies entspricht der intuitiven Sichtweise, die im vorangegangenen Abschnitt dargestellt wurde. Weiterhin zu beachten ist, dass im Gegensatz zu Java oder C++ zwischen primitiven Typen kein Subtyping definiert ist[2].

Betrachtet man diese Definition der Subtyping-Regeln, so ist unmittelbar ersichtlich, dass die Subtyprelation selbst reflexiv und transitiv ist. Offensichtlich gilt darüber hinaus, dass die Subtyprelation auch antisymmetrisch ist, und es sich dementsprechend um eine Halbordnung auf der Menge der Typen handelt.

Lemma 3.1 (Ordnungseigenschaften der Subtyprelation)

(a) $\forall \tau_1, \tau_2 \in \mathit{Type} : \tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_1 \Rightarrow \tau_1 = \tau_2$

(b) $(\mathit{Type}, \leq)$ *ist eine partielle Ordnung.*

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

Beweis: Trivial. □

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

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

Beweis: Trivial. □

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. Des Weiteren ist der leere Objekttyp maximal. Wir werden dieses Lemma benutzen, um nachfolgend eine Verallgemeinerung, das bereits angesprochene Subtyping-Lemma, zu beweisen.

Das Subtyping-Lemma beschreibt eine alternative Charakterisierung der Subtyprelation und erlaubt einen vereinfachten Umgang mit der Subtyprelation, so dass in den nachfolgenden Beweisen die Regel (S-Trans), die ja an keine bestimmte syntaktische Form eines Typs gebunden ist, nicht ständig berücksichtigt werden muss.

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

[2] In Java gilt beispielsweise $\mathbf{long} \leq \mathbf{int} \leq \mathbf{short} \leq \mathbf{byte}$.

(a) $\tau = \tau'$

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

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

Das heißt, zwei Typen stehen genau dann in Subtyprelation zueinander, wenn entweder die beiden Typen gleich sind, oder beides Funktionstypen sind, wobei die Typen der Parameter kontravariant und die Ergebnistypen kovariant in Relation stehen, oder es sind Objekttypen, von denen der kleinere mindestens soviele Methoden enthält wie der größere und die Typen der gemeinsamen Methoden kovariant in Relation stehen.

Beweis:

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

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

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

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

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

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

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

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

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

„⇐“ Für diese Richtung genügt es zu zeigen, dass sich für die im Lemma angegebenen Fälle die Aussage $\tau \leq \tau'$ mit den Regeln fürs Subtyping aus den Bedingungen herleiten lässt, was aus ziemlich offensichtlichen Gründen der Fall ist. □

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

Nachdem im vorangegangenen Abschnitt die Subtyprelation vorgestellt worden ist und einige wichtige Eigenschaften bewiesen wurden, wenden wir uns in diesem Abschnitt der Frage zu, wie Subtyping 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* – sofern notwendig – automatisch zu einem größeren Typ übergegangen. Die Syntax und Semantik der Programmiersprache bleiben dabei unangetastet.

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

Das implizite Subtyping ist sowohl in der Theorie als auch in den gängigen objekt-orientierten Programmiersprache wie Java oder C++ am weitesten verbreitet. Allerdings kommt in Java oder C++ neben dem impliziten Subtyping auch explizites Subtyping in Form von Cast-Operatoren zum Einsatz, die jedoch in der Regel nicht (statisch) typsicher sind. Die einzige heutzutage im praktischen Einsatz befindliche objekt-orientierte Programmiersprache, welche ausschließlich *explizites Subtyping* benutzt, ist O'Caml.

In diesem Abschnitt werden wir die Programmiersprache $\mathcal{L}_o^t$ aus dem vorangegangen Kapitel um Subsumption zur Sprache $\mathcal{L}_o^{sub}$ erweitern und den Aspekt der Typsicherheit dieser Programmiersprache betrachten[3]. 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 der Form $\Gamma \triangleright e :: \tau$ oder $\Gamma \triangleright r :: \phi$ heißt *gültig* für $\mathcal{L}_o^{sub}$, wenn es sich mit den Typregeln von $\mathcal{L}_o^t$ (aus Definition 2.20) sowie der Subsumption-Regel

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

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.

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

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

[3] In Abschnitt 3.5 betrachten wir kurz eine Erweiterung der Programmiersprache $\mathcal{L}_o^t$ um explizites Subtyping.

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

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.

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

Vergleicht man die (Subsume)-Regel mit den bisherigen Typregeln, so wird auch sofort die Ursache dieser Uneindeutigkeit offensichtlich. Bisher waren Typregeln eineindeutig mit der syntaktischen Form eines Ausdrucks verknüpft, zum Beispiel war (App) die einzige auf Applikationen anwendbare Regel und ebenso war (App) ausschließlich auf Applikationen anwendbar.

Im Typsystem der Sprache $\mathcal{L}_o^{sub}$ kann nun aber unabhängig von der syntaktischen Form des Ausdrucks immer entweder die aus der Sprache $\mathcal{L}_o^t$ geerbte Typregel oder die (Subsume)-Regel angewandt werden, wie an den beiden Typherleitungen für den Ausdruck $\mathbf{object}\,(self : \langle m : \mathbf{int}\rangle)\ \mathbf{method}\ m = 1;\ \mathbf{end}$ gut zu ersehen ist. Wir werden später auf das Thema Typeindeutigkeit in einer Sprache mit Subtyping eingehen. Für den Augenblick halten wir lediglich fest, dass die Typeindeutigkeit in der Sprache $\mathcal{L}_o^{sub}$ nicht gilt.

3.3.1 Typsicherheit

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

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

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

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

Betrachten wir weiterhin das Objekt

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

welches zwei Methoden vom Typ **int** bereitstellt. Mit den Typregeln von $\mathcal{L}_o^t$ lässt sich hierfür offensichtlich der Typ $\langle m : \textbf{int}; n : \textbf{int}\rangle$ herleiten. Gemäß der 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:[4]

$$f_m\, obj$$

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

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

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

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

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

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

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

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

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

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

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

Lemma 3.4 (Typumgebungen und frei vorkommende Namen)

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

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

Beweis: Wie zuvor erfolgt der Beweis durch simultane Induktion über die Länge der Typherleitungen. Der neue Fall der (SUBSUME)-Regel folgt unmittelbar nach Induktionsvoraussetzung. □

Ebenso leicht lässt sich das Koinzidenzlemma der Programmiersprache $\mathcal{L}_o^t$ (Lemma 2.6) für die neue Programmiersprache mit Subsumption verallgemeinern:

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

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

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

Beweis: Hier ist ebenfalls nur der neue Fall für die (SUBSUME)-Regel zu betrachten, in dem die Behauptung unmittelbar mit Induktionsvoraussetzung folgt. Die übrigen Fälle sind wie zuvor bereits durch den Beweis des Lemmas für die Programmiersprache $\mathcal{L}_o^t$ (siehe Lemma 2.6) abgedeckt. □

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

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

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

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

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

(a) Wenn für $e \in \mathit{Exp}$ *und* $\tau' \in \mathit{Type}$

1. $\Gamma[{}^{\tau}/_{\mathit{self}}] \rhd e :: \tau'$,
2. $\Gamma^\star \rhd \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ *und*
3. $\forall a \in \mathit{dom}(\Gamma) \cap \mathit{Attribute}, \tau_a \in \mathit{Type} : \Gamma^\star \rhd r(a) :: \tau_a \Leftrightarrow \Gamma \rhd a :: \tau_a$

gilt, dann gilt auch $\Gamma \rhd e\big[{}^{\mathbf{object}\,(\mathit{self}:\tau)\ r\ \mathbf{end}}/_{\mathit{self}}\big] :: \tau'$.

(b) Wenn für $r' \in \mathit{Row}$ *und* $\phi \in \mathit{RType}$

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

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

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

Zum Beweis der Preservation für die Programmiersprache $\mathcal{L}_o^{sub}$ benötigen wir neben den bisherigen Lemmata noch ein weiteres Lemma, das sogenannte *Inversion-Lemma*. Dieses Lemma erlaubt es uns im Folgenden, aus Typurteilen auf die Prämissen zu schließen, ohne jedesmal die (Subsume)-Regel explizit zu berücksichtigen. Wir beschränken uns dabei auf die Aussagen, die wir im Beweis der Preservation- und Progress-Sätze benutzen werden, statt das Lemma in seiner allgemeinsten Form anzugeben.

Lemma 3.8 (Umkehrung der Typrelation)

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

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

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

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

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

(a) Das Typurteil $\Gamma \triangleright op :: \tau$ kann nur mit einer der Typregeln (Const) oder (Subsume) hergeleitet worden sein. Für (Const) kommen dabei nur (Aop) und (Rop) in Frage, und da primitive Typen nur dann in Subtyprelation stehen können, wenn sie gleich sind, folgt unmittelbar die Behauptung.

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

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

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

(c) Das Typurteil $\Gamma \triangleright e_1\, e_2 :: \tau$ kann ausschließlich mit Typregel (App) oder Typregel (Subsume) hergeleitet worden sein. Im Fall von (App) folgt die Behauptung direkt aus den Prämissen, im Fall von (Subsume) unmittelbar mit Induktionsvoraussetzung.

(d) Das Typurteil $\Gamma \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau'$ kann nur mit (Object) oder (Subsume) hergeleitet worden sein. Die Behauptung folgt dann analog zu den vorangegangenen Fällen. □

Damit können wir nun den Typerhaltungssatz für die Programmiersprache $\mathcal{L}_o^{sub}$ formulieren und beweisen. Die Formulierung des Satzes entspricht exakt der bereits für die Programmiersprache $\mathcal{L}_o^t$ benutzten Formulierung, allerdings bedingt die Hinzunahme der (Subsume)-Regel eine andere Vorgehensweise beim Beweis des Satzes.

Satz 3.1 (Typerhaltung, „Preservation“)

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

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

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

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

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

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

2.) $\Gamma^\star \triangleright e_1\, e_2 :: \tau$ mit Typregel (App) kann nur aus Prämissen der Form $\Gamma^\star \triangleright e_1 :: \tau_2 \to \tau$ und $\Gamma^\star \triangleright e_2 :: \tau_2$ folgen. Der small step $e_1\, e_2 \to e'$ kann nur mit einer der Regeln (App-Left), (App-Right), (Op) oder (Beta-V) hergeleitet worden sein.

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

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

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

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

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

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

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

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

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

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

Aufgrund von $dom\,(\Gamma^\star) \cap Attribute = \emptyset$ folgt somit wegen Lemma 3.7 $\Gamma^\star \triangleright \omega[{}^{\mathbf{object}\,(self:\tau')\ \omega\ \mathbf{end}}/_{self}] :: \phi'$. 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') $\Gamma^\star \triangleright \omega[{}^{\mathbf{object}\,(self:\tau')\ \omega\ \mathbf{end}}/_{self}] \# m :: \tau''$ und wegen $\tau'' \leq \tau$ schließlich $\Gamma^\star \triangleright \omega[{}^{\mathbf{object}\,(self:\tau')\ \omega\ \mathbf{end}}/_{self}] \# m :: \tau$ mit Typregel (SUBSUME).

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

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

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

Die übrigen Fälle verlaufen analog, wobei jeweils auch das Lemma 3.8 (Umkehrung der Typrelation) um die notwendigen Aussagen zu erweitern ist. □

Damit ist nun 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 dann noch die Existenz des Übergangsschritts – das sogenannte *Progress-Theorem* – zu zeigen. Dazu beginnen wir – wie bereits im Beweis des Satzes für $\mathcal{L}_o^t$ – mit einem *Canonical Forms Lemma*, welches Aussagen über die möglichen Formen von abgeschlossenen Werten bestimmter Typen macht. Wir beschränken uns ähnlich wie bei $\mathcal{L}_o^t$ auf die Aussagen des Lemmas, die wir im darauf folgenden Beweis des Progress-Theorems benutzen werden.

Lemma 3.9 (Canonical Forms)

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

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

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

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

Beweis: Die Beweise erfolgen jeweils durch Induktion über die Länge der Herleitung der Typurteile mit Fallunterscheidung nach der zuletzt angewandten Typregel. Konkret sind dabei in jedem Fall die syntaktisch passende Typregel und die (Subsume)-Regel zu betrachten. □

Satz 3.2 (Existenz des Übergangsschritts, „Progress“)

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

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

Beweis: Wie bereits für die Programmiersprache $\mathcal{L}_o^t$ erfolgt der Beweis des Satzes 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.) $[\,] \triangleright e :: \tau$ mit (Subsume) kann ausschließlich aus $[\,] \triangleright e :: \tau'$ mit $\tau' \leq \tau$ folgen. Die Aussage folgt dann mit Induktionsvoraussetzung aus $[\,] \triangleright e :: \tau'$.

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

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

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

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

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

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

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

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

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

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

Die restlichen Fälle verlaufen ähnlich. □

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

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

3.4 Minimal Typing

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

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

Definition 3.3 (Subtyping-Regeln fürs Minimal Typing) Die Subtyprelation $\leq_m$ ist die kleinste Relation auf $\mathit{Type} \times \mathit{Type}$, die unter Anwendung der folgenden Regeln abgeschlossen ist:

(SM-REFL) $\quad \beta \leq_m \beta$ für alle $\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

(SM-ARROW) $\quad \dfrac{\tau_1' \leq_m \tau_1 \quad \tau_2 \leq_m \tau_2'}{\tau_1 \to \tau_2 \leq_m \tau_1' \to \tau_2'}$

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

Intutiv ist klar, dass gemäß Lemma 3.3 die Relationen $\leq$ und $\leq_m$ identisch sind. Der wesentliche Unterschied besteht in der Regel (SM-REFL), die nun auf primitive Typen eingeschränkt worden ist, und in der Regel (SM-OBJECT), die eine Kombination der vorherigen Regeln (S-TRANS), (S-OBJ-DEPTH) und (S-OBJ-WIDTH) darstellt.

Um den Beweis der Gleichheit der beiden Relationen $\leq$ und $\leq_m$ zu erleichtern, zeigen wir zunächst, dass die Relation $\leq_m$ transitiv ist:

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

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

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

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

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

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

Beweis:

„⇒" 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 (Subtyping-Lemma).

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

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

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

Wenn $\tau_1 = \tau_1' \to \tau_1''$, dann gilt $\tau_1' \leq \tau_1'$ und $\tau_1'' \leq \tau_1''$ wegen (S-REFL), mit Induktionsvoraussetzung folgt daraus $\tau_1' \leq_m \tau_1'$ und $\tau_1'' \leq_m \tau_1''$, und mit Regel (SM-ARROW) schließlich $\tau_1' \to \tau_1'' \leq_m \tau_1' \to \tau_1''$.

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

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

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

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

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

Aufgrund dieses Ergebnisses müssen wir nicht länger zwischen $\leq$ und $\leq_m$ unterscheiden, sondern können wahlweise die Regeln aus Definition 3.1 oder Definition 3.3 benutzen, um Aussagen der Gestalt $\tau_1 \leq \tau_2$ herzuleiten.

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

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

1. Wenn $\tau, \tau' \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$ und $\tau = \tau'$, dann Antwort „ja“.
2. Wenn $\tau = \tau_1 \to \tau_2$ und $\tau' = \tau_1' \to \tau_2'$, dann überprüfe rekursiv, ob $\tau_1' \leq \tau_1$ und $\tau_2 \leq \tau_2$ gelten.
3. Wenn $\tau = \langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle$ und $\tau' = \langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle$ ist, wobei $\{m_1', \ldots, m_l'\} \subseteq \{m_1, \ldots, m_k\}$, dann überprüfe rekursiv, ob $\tau_i \leq \tau_j$ für alle i, j mit $m_i = m_j'$ gilt.
4. In allen anderen Fällen ist die Antwort „nein“.

Offensichtlich ist der Algorithmus korrekt. 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 $\mathcal{L}_o^m$

Um ein deterministisches Typsystem zu erhalten, müssen nun noch die Typregeln angepasst werden. Dabei ist darauf zu achten, dass für jede syntaktische Form eines Ausdrucks wieder nur genau eine Typregel anwendbar ist.

Im Fall der (Cond)-Regel stellt sich hier dann die Frage, wie der gemeinsame Obertyp für die Teilausdrücke e_1 und e_2 in deterministischer Weise zu finden ist. Die einfachste Lösung besteht darin, stets den kleinsten gemeinsamen Obertyp zu wählen, das sogenannte *Supremum*.

Definition 3.4 (Suprema und Infima)

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

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

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

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

Insbesondere ist das Supremum zweier inkompatibler Typen undefiniert. Beispielsweise existiert kein Supremum für **int** und $\langle m : \mathbf{bool} \rangle$. Da die Typen der Parameter für Funktionstypen kontravariant in Typrelation zueinander stehen, benötigt man zusätzlich zum Begriff des Supremums auch noch den Begriff des Infimums, also den größten gemeinsamen Untertyp.

Lemma 3.11 *Seien $\tau_1, \tau_2 \in \mathit{Type}$. Dann gilt:*

(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 (Suprema und Infima). □

Offensichtlich existiert für zwei kompatible Typen stets ein Supremum, welches wiederum Subtyp eines bekannten Typs ist. Das folgende Lemma formalisiert diesen Sachverhalt:

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

Beweis: Trivial. □

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

Dahinter steckt folgende Überlegung: Typurteile für diese speziellen Ausdrücke werden lediglich zum Beweis der Typsicherheit mit Hilfe des Kalküls benötigt. 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 der Form $\Gamma \rhd_m e :: \tau$ oder $\Gamma \rhd_m r :: \phi$ heißt gültig für $\mathcal{L}_o^m$, wenn es sich mit den Typregeln für die funktionale Kernsprache

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

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

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

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

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

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

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

den folgenden Typregeln für Objekte

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

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

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

sowie den folgenden Typregeln für Reihen

(Empty) $\Gamma \rhd_m \epsilon :: \emptyset$

(Attr) $$\frac{\Gamma^\star \rhd_m e :: \tau \quad \Gamma[\tau/_a] \rhd_m r_1 :: \phi}{\Gamma \rhd_m \mathbf{val}\, a = e;\, r_1 :: \phi}$$

(Method-Subsume) $$\frac{\Gamma \rhd_m self :: \langle m : \tau; \phi' \rangle \quad \Gamma \rhd_m e :: \tau' \quad \Gamma \rhd_m r_1 :: \phi \quad \tau' \leq \tau}{\Gamma \rhd_m \mathbf{method}\, m = e;\, r_1 :: (m : \tau; \emptyset) \oplus \phi}$$

herleiten lässt.

Die Idee hinter diesem Regelwerk ist, dass man Anwendungen von (Subsume) in vielen Fällen ans Ende einer Herleitung verschieben kann. Dann kann man auch auf die Anwendung von (Subsume) verzichten, da diese nur den Typ vergrößern würde.

An einigen Stellen, insbesondere bei (App) und (Cond) jedoch können Anwendungen der Subsumption-Regel nicht verschoben werden. Bei (App) muss der Typ des Arguments zum Typ des Parameters der Funktion passen, entsprechend muss es möglich sein, entweder den Typ der Funktion oder den Typ des Arguments zuvor durch (Subsume) anzupassen. Diese Idee kann aber direkt in die (App)-Regel integriert werden, so dass,

falls notwendig, eine Typanpassung des Arguments erfolgt. Entsprechend bestimmt die Regel (App-Subsume) zunächst die Typen der Teilausdrücke und prüft anschließend, ob der Typ des Arguments kleiner als – also „mindestens so gut“ wie – der Parametertyp der Funktion ist.

Im Fall der (Cond)-Regel existiert das bereits angesprochene Problem, dass für die Teilausdrücke e_1 und e_2 ein gemeinsamer Typ τ gefunden werden muss. Die einfachste Lösung an dieser Stelle, die durch die Regel (Cond-Subsume) realisiert wird, ist, den kleinsten gemeinsamen Obertyp der Typen, die zuvor für die Teilausdrücke bestimmt wurden, zu wählen. Dabei gilt zu beachten, dass (Cond-Subsume) nur anwendbar ist, wenn das Supremum der beiden Typen existiert. In der Implementation eines Typecheckers auf Basis des Minimal Typing bedeutet dies, dass, falls die Typen inkompatibel sind, der Programmierer mit einer entsprechenden Fehlermeldung behelligt werden muss.

Wir wollen nun zeigen, dass der Minimal Typing Kalkül aus Definition 3.5 korrekt ist, also dass die Typrelation der Programmiersprache $\mathcal{L}_o^m$ eine Teilmenge der Typrelation von $\mathcal{L}_o^{sub}$ ist. Anders ausgedrückt bedeutet das: Wenn sich mit den Regeln von $\mathcal{L}_o^m$ für einen Ausdruck e in einer Typumgebung Γ ein Typ τ herleiten lässt, so lässt sich auch mit den Regeln von $\mathcal{L}_o^{sub}$ der gleiche Typ τ herleiten (und entsprechend für Reihen).

Satz 3.5 (Korrektheit des Minimal Typing)

(a) $\forall \Gamma \in \mathit{TEnv}, e \in \mathit{Exp}, \tau \in \mathit{Type} : \Gamma \rhd_m e :: \tau \;\Rightarrow\; \Gamma \rhd e :: \tau$

(b) $\forall \Gamma \in \mathit{TEnv}, r \in \mathit{Row}, \phi \in \mathit{RType} : \Gamma \rhd_m r :: \phi \;\Rightarrow\; \Gamma \rhd r :: \phi$

Intuitiv sollte im Wesentlichen schon klar sein, dass dieser Satz gilt. Wir betrachten daher im folgenden Beweis lediglich die Herleitungen, die mit der Anwendung einer der fünf geänderten Regeln enden.

Beweis: Wie üblich führen wir den Beweis durch simultane Induktion über die Länge der Herleitungen der Typurteile $\Gamma \rhd_m e :: \tau$ und $\Gamma \rhd_m r :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel (aus dem Minimal Typing Kalkül). Wie bereits erwähnt, betrachten wir lediglich die interessanten Fälle.

1.) $\Gamma \rhd_m e_1\, e_2 :: \tau$ mit Typregel (App-Subsume) kann nur aus Prämissen der Form $\Gamma \rhd_m e_1 :: \tau_2' \to \tau$, $\Gamma \rhd_m e_2 :: \tau_2$ und $\tau_2 \leq \tau_2'$ folgen. Nach Induktionsvoraussetzung gilt also $\Gamma \rhd e_1 :: \tau_2' \to \tau$ und $\Gamma \rhd e_2 :: \tau_2$, woraus sich das Typurteil $\Gamma \rhd e_1\, e_2 :: \tau$ wie folgt herleiten lässt:

$$\dfrac{\Gamma \rhd e_1 :: \tau_2' \to \tau \qquad \dfrac{\Gamma \rhd e_2 :: \tau_2 \qquad \tau_2 \leq \tau_2'}{\Gamma \rhd e_2 :: \tau_2'}\,\textsc{Subsume}}{\Gamma \rhd e_1\, e_2 :: \tau}\,\textsc{App}$$

2.) $\Gamma \rhd_m \mathbf{rec}\, x : \tau.e :: \tau$ mit Typregel (Rec-Subsume) bedingt $\Gamma[^{\tau}/_{x}] \rhd_m e :: \tau'$ und $\tau' \leq \tau$. Mit Induktionsvoraussetzung folgt daraus $\Gamma[^{\tau}/_{x}] \rhd e :: \tau'$, und die Behauptung lässt sich folgendermaßen herleiten:

$$\dfrac{\dfrac{\Gamma[^{\tau}/_{x}] \rhd e :: \tau' \qquad \tau' \leq \tau}{\Gamma[^{\tau}/_{x}] \rhd e :: \tau}\ \textsc{Subsume}}{\Gamma \rhd \mathbf{rec}\, x : \tau.e :: \tau}\ \textsc{Rec}$$

3.) Für $\Gamma \rhd_m \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau_1 \vee \tau_2$ gilt nach Voraussetzung $\Gamma \rhd_m e_0 :: \mathbf{bool}$, $\Gamma \rhd_m e_1 :: \tau_1$ und $\Gamma \rhd_m e_2 :: \tau_2$. Sei $\tau = \tau_1 \vee \tau_2$, dann gilt wegen Lemma 3.11 $\tau_1 \leq \tau$ und $\tau_2 \leq \tau$, und wegen Induktionsvoraussetzung gilt $\Gamma \rhd e_0 :: \mathbf{bool}$, $\Gamma \rhd e_1 :: \tau_1$ und $\Gamma \rhd e_2 :: \tau_2$. Damit lässt sich die Behauptung wie folgt herleiten:

$$\dfrac{\Gamma \rhd e_0 :: \mathbf{bool} \qquad \dfrac{\dfrac{}{\Gamma \rhd e_1 :: \tau_1}\ \tau_1 \leq \tau}{\Gamma \rhd e_1 :: \tau}\ \textsc{Subsume} \qquad \dfrac{\dfrac{}{\Gamma \rhd e_2 :: \tau_2}\ \tau_2 \leq \tau}{\Gamma \rhd e_2 :: \tau}\ \textsc{Subsume}}{\Gamma \rhd \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau}\ \textsc{Cond}$$

4.) $\Gamma \rhd_m \{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\} :: \tau$ mit Typregel (Dupl-Subsume) kann ausschließlich aus Prämissen der Form

$$\Gamma \rhd_m \mathit{self} :: \tau$$

sowie

$$\forall i = 1, \ldots, n : \Gamma \rhd_m a_i :: \tau_i \ \wedge\ \Gamma \rhd_m e_i :: \tau_i' \ \wedge\ \tau_i' \leq \tau_i$$

folgen. Nach Induktionsvoraussetzung gilt

$$\Gamma \rhd \mathit{self} :: \tau$$

und

$$\forall i = 1, \ldots, n : \Gamma \rhd a_i :: \tau_i \ \wedge\ \Gamma \rhd e_i :: \tau_i' \ \wedge\ \tau_i' \leq \tau_i,$$

und mit Typregel (Subsume) folgt

$$\forall i = 1, \ldots, n : \Gamma \rhd a_i :: \tau_i \ \wedge\ \Gamma \rhd e_i :: \tau_i.$$

Daraus schließlich folgt $\Gamma \rhd \{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\} :: \tau$ mit Typregel (Dupl).

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

Die übrigen Fälle folgen trivialerweise aus der Übereinstimmung der Regeln. □

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

$$[\,] \triangleright_m e :: \tau$$
$$\overset{\text{Satz 3.5}}{\Longrightarrow} \quad [\,] \triangleright e :: \tau$$
$$\overset{\text{Satz 3.3}}{\Longrightarrow} \quad \text{die Berechnung von } e \text{ bleibt nicht stecken.}$$

□

Vollständigkeit des Minimal Typing

Anhand der Überlegungen aus dem vorangegangenen Abschnitt ist klar, dass der Minimal Typing Kalkül korrekt und insbesondere typsicher ist. Dies sichert uns aber lediglich, dass jeder Ausdruck, der in $\mathcal{L}_o^m$ wohlgetypt ist, auch in $\mathcal{L}_o^{sub}$ wohlgetypt ist (mit dem gleichen Typ). Damit ist jedoch noch nicht gezeigt, dass die Typsysteme äquivalent sind, d.h. dass $\mathcal{L}_o^m$ als Implementation für $\mathcal{L}_o^{sub}$ zulässig ist.

Um die Äquivalenz zu beweisen, muss noch gezeigt werden, dass der Minimal Typing Kalkül vollständig ist. Das heißt, für jeden Ausdruck, für den in $\mathcal{L}_o^{sub}$ ein Typ herleitbar ist, ist auch in $\mathcal{L}_o^m$ ein Typ herleitbar, welcher kleiner ist als der in $\mathcal{L}_o^{sub}$ hergeleitete.

Wie bereits erwähnt, beschränkt sich die Sprache $\mathcal{L}_o^m$ auf Ausdrücke, die der Programmierer im Programm notiert, und ignoriert Ausdrücke, die ausschließlich als Zwischenschritte der Berechnung entstehen. Entsprechend fassen wir diese Ausdrücke in der Menge $Exp_P \subseteq Exp$ zusammen:

Definition 3.6 $Exp_P := \{e \in Exp \mid e \text{ enthält keinen Teilausdruck der Form } \omega\#m\}$

Insbesondere kann also eine Reihe niemals außerhalb eines Objekts erscheinen, da dies sonst nur durch Ausdrücke der Form $\omega\#m$ möglich war. Entsprechend müssen Typumgebungen für Reihen in gültigen Typurteilen stets einen Typ für *self* enthalten. Dazu definieren wir eine Menge $TEnv_\phi \subseteq TEnv$, die alle Typumgebungen enthält, in denen ein Typ für *self* eingetragen ist, der länger ist als $\langle\phi\rangle$.

Definition 3.7 $TEnv_\phi := \{\Gamma \in TEnv \mid \exists \phi', \phi'' \in RType : \Gamma(self) = \langle\phi'\rangle \wedge \phi' = \phi \oplus \phi''\}$

In Worten bedeutet $\phi' = \phi \oplus \phi''$, dass ϕ' eine Verlängerung von ϕ ist, also dass ϕ' mindestens alle Methodennamen von ϕ enthält, und ϕ' und ϕ auf den gemeinsamen Methodentypen übereinstimmen.

Definition 3.8 (Typumgebungen und Subtyping) Seien $\Gamma, \Gamma' \in TEnv$. Dann definieren wir auf $TEnv$ eine Subtyprelation wie folgt komponentenweise:

$$\Gamma' \leq \Gamma \quad :\Leftrightarrow \quad dom\,(\Gamma) = dom\,(\Gamma') \wedge \forall id \in dom\,(\Gamma) : \Gamma(id) \leq \Gamma'(id)$$

Für diese Subtyprelation auf Typumgebungen gelten einige einfache Aussagen, die wir im nachfolgenden Lemma festhalten:

Lemma 3.13 *Seien* $\Gamma_1, \Gamma_2 \in \mathit{TEnv}$. *Dann gilt:*

(a) $\Gamma_1 \leq \Gamma_2 \;\Rightarrow\; \Gamma_1^\star \leq \Gamma_2^\star$

(b) $\forall \tau \in \mathit{Type} : \forall id \in \mathit{Id} : \Gamma_1 \leq \Gamma_2 \;\Rightarrow\; \Gamma_1[^{\tau}/_{id}] \leq \Gamma_2[^{\tau}/_{id}]$

(c) $\forall \tau_1, \tau_2 \in \mathit{Type} : \forall id \in \mathit{Id} : \Gamma_1 \leq \Gamma_2 \wedge \tau_1 \leq \tau_2 \;\Rightarrow\; \Gamma_1[^{\tau_1}/_{id}] \leq \Gamma_2[^{\tau_2}/_{id}]$

Beweis: Trivial. □

Damit können wir dann die Vollständigkeit des Minimal Typing Kalküls zeigen. Wichtig zu beachten ist, dass die Vollständigkeit nicht allgemein für alle Ausdrücke gilt, sondern lediglich für Ausdrücke, die der Programmierer im Quelltext notieren kann.

Satz 3.7 (Vollständigkeit des Minimal Typing)

(a) $\forall \tau \in \mathit{Type} : \forall \Gamma, \Gamma' \in \mathit{TEnv} : \forall e \in \mathit{Exp}_P :$
$\Gamma \rhd e :: \tau \wedge \Gamma' \leq \Gamma \wedge \Gamma' =_{dom\,(\Gamma) \cap \mathit{Attribute}} \Gamma \;\Rightarrow\; \exists \tau' \in \mathit{Type} : \Gamma' \rhd_m e :: \tau' \wedge \tau' \leq \tau$

(b) $\forall \phi \in \mathit{RType} : \forall \Gamma, \Gamma' \in \mathit{TEnv}_\phi : \forall r \in \mathit{Row} :$
$\Gamma \rhd r :: \phi \wedge \Gamma' \leq \Gamma \;\Rightarrow\; \Gamma' \rhd_m r :: \phi$

Die Voraussetzungen scheinen auf den ersten Blick sehr 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 nicht aber dem Programmierer zur Verfügung stehen und folglich nicht durch den Minimal Typing Kalkül abgedeckt werden. Die Einschränkungen sind auch weniger restriktiv als man zunächst annehmen könnte, wie wir im Anschluss an den Beweis sehen werden.

Beweis: Der Beweis der Vollständigkeit erfolgt durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma \rhd e :: \tau$ und $\Gamma \rhd r :: \phi$, und Fallunterscheidung nach der zuletzt angewandten Typregel (der Programmiersprache $\mathcal{L}_o^{sub}$). Wir betrachten dazu exemplarisch die folgenden Fälle:

1.) Für $\Gamma \rhd e :: \tau$ mit Typregel (SUBSUME) existiert nach Voraussetzung ein $\tau' \in \mathit{Type}$ mit $\tau' \leq \tau$ und $\Gamma \rhd e :: \tau'$. Nach Induktionsvoraussetzung existiert ein $\tau'' \leq \tau'$, so dass $\Gamma' \rhd_m e :: \tau''$ gilt. Wegen (S-TRANS) schließlich folgt $\tau'' \leq \tau$.

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

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

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

$$\Gamma' \rhd_m e_1 :: \tau_1$$

sowie ein $\tau_2 \in \mathit{Type}$ mit $\tau_2 \leq \tau$ und

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

Gemäß Lemma 3.12 ist also das Supremum $\tau_1 \vee \tau_2$ definiert und es gilt $\tau_1 \vee \tau_2 \leq \tau$. Also folgt insgesamt

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

mit Typregel (COND-SUBSUME).

3.) $\Gamma \rhd \mathbf{rec}\, x : \tau.e :: \tau$ mit Typregel (REC) bedingt $\Gamma[^{\tau}/_{x}] \rhd e :: \tau$. Wegen Lemma 3.13 gilt $\Gamma'[^{\tau}/_{x}] \leq \Gamma[^{\tau}/_{x}]$, also existiert nach Induktionsvoraussetzung ein $\tau' \in \mathit{Type}$ mit $\tau' \leq \tau$ und

$$\Gamma'[^{\tau}/_{x}] \rhd_m e :: \tau'.$$

Daraus folgt

$$\Gamma' \rhd_m \mathbf{rec}\, x : \tau.e :: \tau$$

mit Typregel (REC-SUBSUME), und trivialerweise gilt $\tau \leq \tau$.

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

$$\Gamma' \rhd_m \mathit{self} :: \tau'.$$

Darüber hinaus existieren $\tau'_1, \ldots, \tau'_n \in \mathit{Type}$ mit $\tau'_i \leq \tau_i$ und

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

für $i = 1, \ldots, n$. Wegen $\Gamma' =_{dom\,(\Gamma) \cap \mathit{Attribute}} \Gamma$ gilt nach wie vor[5]

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

für $i = 1, \ldots, n$ und somit folgt

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

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

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

5.) 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 ein $\tau_m \in \mathit{Type}$ mit $\tau_m \leq \langle m : \tau; \phi \rangle$, so dass

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

gilt. Gemäß dem Subtyping-Lemma (Lemma 3.3) existieren $\tau' \in \mathit{Type}, \phi' \in \mathit{RType}$ 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 $\mathcal{L}_o^m$.

6.) $\Gamma_\phi \triangleright \textbf{method}\ m = e'; r' :: \phi$ 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 $\tau \in \mathit{Type}$ 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 $\Gamma_\phi \in \mathit{TEnv}_{\phi'}$. Weiterhin existiert nach Definition 3.7 ein $\phi'' \in \mathit{RType}$ mit $\Gamma'_\phi(\mathit{self}) = \langle (m : \tau'; \emptyset) \oplus \phi' \oplus \phi'' \rangle$, und wegen Typregel (ID) folgt daraus

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

Insgesamt folgt also

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

mit der Typregel (METHOD-SUBSUME) 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:

Korollar 3.1 *Sei $e \in \mathit{Exp}_P$ ein abgeschlossener Ausdruck. Dann gilt:*

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

Vereinfacht gesprochen existiert zu jedem Typurteil der Programmiersprache $\mathcal{L}_o^{sub}$ ein äquivalentes in der Programmiersprache $\mathcal{L}_o^m$, wobei aber statt eines beliebigen Typs aus der Menge der möglichen Typen stets der kleinstmögliche Typ hergeleitet wird.

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 sein sollten, den folgenden Satz zu beweisen:

Satz 3.8 (Eindeutigkeit des minimalen Typs)

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

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

Statt beliebige Ausdrücke aus *Exp* zuzulassen, könnten wir uns hierbei ebenfalls auf Exp_P beschränken, also auf Ausdrücke, die der Programmierer im Quelltext des Programms notieren kann. Aber diese Einschränkung wäre unnötig, da wir lediglich zeigen, dass höchstens ein Typ existiert, und für Ausdrücke der Form $\omega\#m$ existiert nach Definition kein Typ.

Beweis: Die Behauptung wird durch simultane Induktion über die Struktur von e und r mit 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. Wie 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. □

3.4.3 Typalgorithmus

Basierend auf dem Regelwerk aus Definition 3.5 lässt sich nun für die Programmiersprache $\mathcal{L}_o^m$ ein Typalgorithmus angeben. Wie im vorherigen Abschnitt nachgewiesen, sind die Sprachen $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^m$ äquivalent. Folglich ist der Typalgorithmus gleichermaßen ein Entscheidungsalgorithmus für die Typrelation der Sprache $\mathcal{L}_o^{sub}$.

Algorithmus 3.2 (Typalgorithmus für $\mathcal{L}_o^m$) Als Eingabe erhält der Algorithmus eine Typumgebung $\Gamma \in TEnv$ und einen Ausdruck $e \in Exp$ oder eine Reihe $r \in Row$. Als Ausage liefert der Algorithmus den kleinsten Typ für den Ausdruck oder die Reihe, oder „Typfehler", falls kein solcher Typ existiert.

1. Falls $e = c \in Const$, dann liefere den eindeutigen Typ für die Konstante c.

2. Falls $e = id \in Id$, dann liefere $\Gamma(id)$, falls $id \in \Gamma$, sonst „Typfehler".

3. Falls $e = e_1\, e_2$, dann bestimme rekursiv die Typen für e_1 und e_2 in der Typumgebung Γ. Existieren diese Typen, so prüfe, ob der Typ für e_1 die Form $\tau_2' \to \tau$ hat, und ob $\tau_2 \leq \tau_2'$ gilt (mittels Algorithmus 3.1), wobei τ_2 der Typ für e_2 ist. Sollten alle Bedingungen erfüllt sein, liefere τ, sonst „Typfehler“.

Die prinzipielle Umsetzung der Minimal Typing Regeln sollte damit klar sein.

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 O'Caml vorgestellten Ansatz. Dabei wird die Syntax und Semantik der Programmiersprache um sogenannte *Coercions* erweitert, welche es dem Programmierer erlauben, 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

$$e \quad ::= \quad (e_1 : \tau')$$

zu erweitern, während für ein Typsystem mit Typinferenz (vgl. [Pie02, S.317ff]), wie zum Beispiel das O'Caml Typsystem, die allgemeinere Form

$$e \quad ::= \quad (e_1 : \tau <: \tau')$$

notwendig ist. Wir verwenden im folgenden die allgemeinere Form.

Für die Programmiersprache $\mathcal{L}_o^{sub}$ existiert eine Subsumption-Regel, die an beliebiger Stelle in einer Typherleitung einem Übergang zu einem Supertyp ermöglicht. Gerade diese (Subsume)-Regel war aber der Grund für die Uneindeutigkeit des Typsystems von $\mathcal{L}_o^{sub}$. Statt der (Subsume)-Regel nimmt man deshalb die sogenannte (Coerce)-Regel zum Typsystem hinzu:

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

Hierdurch kann während der Typherleitung ebenfalls zu einem Supertyp übergegangen werden, allerdings immer nur an den Stellen, die der Programmierer explizit dafür vorgesehen hat. Somit erhält man auf recht einfache Weise ein eindeutiges Typsystem mit Subtyping, welches sich im Gegensatz zum Typsystem der Programmiersprache $\mathcal{L}_o^m$ auch um Typinferenz und ML-Polymorphie erweiteren lässt.

Allerdings muss dann auch die Semantik der Programmiersprache erweitert werden, und zwar um eine Regel für die neue Form von Ausdruck. Wie bisher führen wir keinerlei Typüberprüfung zur Laufzeit durch, so dass die small step Regel entsprechend einfach aussieht:

$$(\text{Coerce}) \quad (e : \tau <: \tau') \to e$$

Die im Ausdruck enthaltenen Typinformationen werden verworfen, und es erfolgt eine Auswertung des Ausdrucks mit dem bisherigen small step Regelwerk.

Es ist ziemlich offensichtlich, dass eine solche Programmiersprache typsicher ist, denn für jede Typherleitung im Typsystem dieser Programmiersprache lässt sich eine äquivalente Herleitung im Typsystem der Sprache $\mathcal{L}_o^{sub}$ konstruieren, indem man

(a) alle coercions aus dem Programmtext entfernt, und

(b) an allen Stellen, an denen in der ursprünglichen Herleitung die (Coerce)-Regel angewendet wurde, die (Subsume)-Regel anwendet mit dem bekannten Supertyp.

Das bedeutet, es gilt

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

Coercion-Ausdrücke sind eingeschränkt vergleichbar mit dem `static_cast` Konstrukt in C++ (vgl. [Str00, S. 440f]), insofern als dass ein `static_cast` ausschließlich zur Compilezeit durch den Typechecker überprüft wird. Da jedoch in C++ für beliebige Zeigertypen ein `static_cast` von und nach `void*` möglich ist, kann hier nicht von einem typsicheren Cast die Rede sein.

Seien beispielsweise `A` und `B` Klassen, wobei `B` von `A` erbt, was in C++ impliziert, dass $\texttt{B} \leq \texttt{A}$ gilt, und sei `a` eine Variable von Typ `A*`, die auf eine gültige Instanz der Klasse `A` zeigt. Dann ist die folgende Zeile ein gültiges C++-Statement:

```
B* b = static_cast<B*> (static_cast<void*> (a));
```

Diese sogenannten Upcasts erfordern in C++ eigentlich einen `reinterpret_cast`, der als generell unsicher deklariert ist. Wie man jedoch sieht, lässt sich auch der scheinbar sichere `static_cast` Operator, der zur Compile-Zeit durch den Typechecker geprüft wird, mit einem einfachen Trick aushebeln. Grund dafür ist, dass `void*` im C++-Typsystem für explizite Typkonversionen zugleich größter wie auch kleinster Zeigertyp ist[6].

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.

[6] Dies ist eine der vielen Stellen, an der sich das unliebsame C-Erbe bemerkbar macht.

4 Rekursive Typen

Im vorangegangenen Kapitel haben wir das einfache objekt-orientierte Typsystem aus Kapitel 2 um Subtyping 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 hinzugewonnenen wohlgetypten Ausdrücke zu Unrecht durch das Typsystem von $\mathcal{L}_o^t$ abgelehnt wurden, denn deren Berechnung bleibt gemäß Satz 3.3 nicht stecken.

Allerdings ist das Typsystem der Sprache $\mathcal{L}_o^{sub}$ immer noch zu eingeschränkt, um sinnvoll mit funktionalen Objekten programmieren zu können. Zum Beispiel ist es nicht möglich, den folgenden Ausdruck so mit Typinformationen zu versehen, dass mit den Typregeln von $\mathcal{L}_o^{sub}$ ein Typ herleitbar wäre.

$$
\begin{array}{l}
\textbf{let}\, point = \\
\quad \textbf{object}\,(self) \\
\qquad \textbf{val}\ x = 1; \\
\qquad \textbf{val}\ y = 2; \\
\qquad \textbf{method}\ move = \lambda dx.\lambda dy.\{\langle x = x + dx; y = y + dy\rangle\}; \\
\quad \textbf{end} \\
\textbf{in}\, point\#move\, 2\, 1
\end{array}
$$

Offensichtlich würde die Berechnung dieses Ausdrucks nicht stecken bleiben, sondern stattdessen mit einem Punktobjekt terminieren, mit den Koordinaten $x = 3$ und $y = 3$. Die Methode *move* liefert ein neues Punktobjekt, welches sich lediglich in den Werten der Attribute x und y unterscheidet. Insbesondere besitzt aber dieses neue Punktobjekt ebenfalls eine Methode *move*.

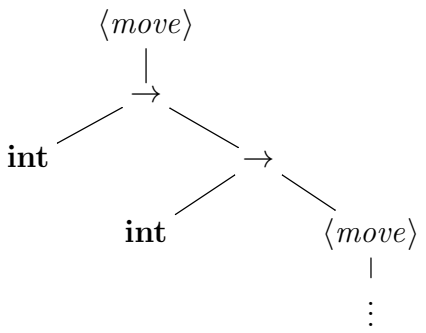
Für das Typsystem bedeutet das, dass ein Typ für *point* „sich selbst enthalten“ müsste, und zwar als Ergebnistyp der Methode *move*. Oder anders ausgedrückt, gesucht ist ein Typ τ_{point} für den gilt:

$$\tau_{point} = \langle move : \textbf{int} \to \textbf{int} \to \tau_{point}\rangle$$

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.

Derartige rekursive Typen lassen sich als unendliche Bäume darstellen, an deren Knoten sich Typkonstruktoren befinden. Beispielsweise entspricht dem Typ τ_{point} der nach-

folgend dargestellte unendliche Baum:



Der Typkonstruktor $\langle move \rangle$ ist einstellig, und auf einen Typ τ angewandt liefert er den Objekttyp $\langle move : \tau \rangle$. Entsprechend existiert für die mit $\langle move \rangle$ markierten Knoten jeweils nur exakt ein Kindknoten. Der Typkonstruktor $\rightarrow$ ist zweistellig und liefert, angewandt auf zwei Typen, einen entsprechenden Funktionstyp.

Es ist offensichtlich, dass sich zu jeder Rekursionsgleichung dieser Art in eindeutiger Weise ein unendlicher Baum konstruieren lässt. Der Baum zu einem derartigen rekursiven Typ enthält immer mindestens einen unendlichen Pfad. Für einfache, nicht rekursive Typen, wie zum Beispiel **int** oder **int** $\rightarrow$ **bool** lässt sich ein entsprechender (endlicher) Baum angeben, der gerade dem Syntaxbaum für diesen Typ entspricht. Analog zu [Pie02, S.267ff] definieren wir (rekursive) Typäquivalenz auf der Baumdarstellung:

Definition 4.1 (Typ-äquivalenz/-gleichheit) Zwei rekursive Typen heißen *äquivalent*, wenn die zugehörigen (unendlichen) Bäume gleich sind.

Damit ist die Semantik für rekursive Typen bereits definiert. Allerdings basiert dieser naive Ansatz auf unendlichen Datenstrukturen, die bekanntlich weder formal noch algorithmisch besonders gut zu handhaben sind.

Um also einen insbesondere algorithmisch handhabbaren Formalismus für ein Typsystem mit rekursiven Typen zu erhalten, werden wir eine endliche Darstellung für rekursive Typen entwickeln, die sogenannten *μ-Typen*, und dann basierend auf dieser endlichen Darstellung eine Definition der Typäquivalenz angeben, die äquivalent zur obigen Definition ist und sich gleichzeitig leicht für einen Typalgorithmus adaptieren lässt.

4.1 Die Sprache $\mathcal{L}_o^{rt}$

Die in 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 4.2 (Syntax der Sprache $\mathcal{L}_o^{rt}$) Vorgegeben sei eine Menge *TName* von *Typnamen* t.

(a) Die Menge $Type^{raw}$ aller syntaktisch herleitbaren Typen τ^{raw} ist durch die kontextfreie Grammatik

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

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

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

definiert.

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

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

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

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

definiert.

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

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

wobei $\mathcal{P}(\tau)$ die Menge aller im Typ τ vorkommenden Typen bezeichnet.

Rekursive Typen sind hierbei analog zu rekursiven Ausdrücken zu lesen, d.h. mit $\mu t.\tau$ bezeichnen wir den Typ, der die Gleichung

$$\mu t.\tau = \tau[{}^{\mu t.\tau}/_t]$$

erfüllt. Allerdings handelt es sich hierbei nicht um die syntaktische Gleichheit der beiden Typen, sondern vielmehr um die übereinstimmende Bedeutung.

Mit $Type^{raw}$ sind die syntaktisch herleitbaren Typen definiert, also insbesondere auch nicht abgeschlossene Typen, die frei vorkommende Typnamen enthalten. Nicht abgeschlossene Typen lassen sich jedoch in der anfangs vorgestellten Baumsemantik nicht sinnvoll repräsentieren. Folglich beschränkt sich die Menge der Typen für die Programmiersprache $\mathcal{L}_o^{rt}$ auf abgeschlossene Typen, die als (unendlicher) Baum repräsentiert werden können. Offensichtlich existiert für einen Typ der Form $\mu t.\mu t_1.\ldots.\mu t_n.t$ keine Baumdarstellung.

Bevor wir uns jedoch genauer mit dem Begriff der Gleichheit von rekursiven Typen beschäftigen, kommen wir zunächst auf das Beispiel mit dem Punktobjekt zurück. In der Sprache $\mathcal{L}_o^{rt}$ lässt sich das Beispiel nun wie folgt mit Typinformationen versehen:

$$
\begin{array}{l}
\textbf{let}\, point = \\
\quad \textbf{object}\,(self : \mu t.\langle move : \textbf{int} \to \textbf{int} \to t;\, \emptyset\rangle) \\
\qquad \textbf{val}\; x = 1; \\
\qquad \textbf{val}\; y = 2; \\
\qquad \textbf{method}\; move = \lambda dx : \textbf{int}.\lambda dy : \textbf{int}.\{\langle x = x + dx; y = y + dy\rangle\}; \\
\quad \textbf{end} \\
\textbf{in}\, point\#move\, 2\, 1
\end{array}
$$

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

$$\mu t.\langle move : \textbf{int} \to \textbf{int} \to t;\, \emptyset\rangle,$$

was dem zu erwartenden Ergebnis entsprechen würde.

Um dieses Ergebnis formal herleiten zu können, übertragen wir nun die Baumsemantik für rekursive Typen auf die endliche μ-Darstellung. Dazu definieren wir zunächst die Menge der Typkonstruktoren, die als Markierungen für die Knoten der (unendlichen) Bäume verwendet werden:

Definition 4.3 Die Menge *TCons* der Typkonstruktoren sei definiert durch

$$TCons \;=\; \{\textbf{bool}, \textbf{int}, \textbf{unit}, \to\} \cup \{\langle m_1; \ldots; m_n\rangle \mid n \in \mathbb{N},\; m_1, \ldots, m_n \in Method\},$$

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

Weiter definieren wir Funktionen, mit denen sich zu einem μ-Typ der entsprechende (unendliche) Typbaum konstruieren lässt:

Definition 4.4

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

$$
\begin{array}{rcl}
root\,(\beta) & = & \beta \quad \text{für } \beta \in \{\textbf{bool}, \textbf{int}, \textbf{unit}\} \\
root\,(\tau_1 \to \tau_2) & = & \to \\
root\,(\langle m_i : {\tau_i}^{i=1\ldots n}\rangle) & = & \langle m_i^{i=1\ldots n}\rangle \\
root\,(\mu t.\tau) & = & root\,(\tau[{}^{\mu t.\tau}/_t])
\end{array}
$$

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

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

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

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

Statt $child(\tau, i)$ schreiben wir $child_i(\tau)$.

Die Funktion *root* liefert zu einem gegebenen μ-Typ den Typkonstruktor, welcher an der Wurzel des zugehörigen Typbaums steht. Die Funktion *arity* berechnet die Anzahl der Unterknoten dieses Wurzelknotens, und die Funktion $child_i$ liefert den Typ für den i-ten Teilbaum. Offensichtlich lässt sich durch rekursive Anwendung dieser Funktionen für jeden μ-Typ die zugehörige Baumdarstellung erzeugen.

Mit Hilfe dieser Funktionen kann nun die anfangs zunächst informal definierte Typgleichheit rekursiver Typen (vgl. Definition 4.1) formalisiert werden. Die grundlegende Idee ist dabei denkbar einfach: Zwei (unendliche) Bäume sind gleich, wenn ihre Wurzeln die gleichen Markierungen aufweisen und die Teilbäume unterhalb der Wurzeln gleich sind.

Definition 4.5 Die Relationen $\sim_n$ seien wie folgt induktiv definiert

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

und die Relation $\sim$ ist durch

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

definiert, also als Durchschnitt der Relationen $\sim_n$.

Die Relation $\sim_{n+1}$ enthält also alle Typen, deren Typkonstruktoren an der Wurzel übereinstimmen und deren Teilbäume bzgl. $\sim_n$ gleich sind. Die Typrelationen $\sim_n$ stellen damit eine schrittweise Verfeinerung dar. Bildet man den Schnitt dieser Relationen, so erhält man genau die zuvor intuitiv definierte Vorstellung von Typgleichheit, in Form der Relation $\sim$. Bleibt noch zu zeigen, dass es sich bei $\sim$ tatsächlich um eine Äquivalenzrelation handelt:

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.

- $n = 0$

  Da $\sim_0 = \mathit{Type}^2$ ist klar, dass $\sim_0$ reflexiv, transitiv und symmetrisch ist.

- $n \rightsquigarrow n+1$

  Nach Induktionsvoraussetzung ist $\sim_n$ reflexiv, transitiv und symmetrisch.

  Reflexivität: Da $\sim_n$ reflexiv ist, folgt trivialerweise, dass auch $\sim_{n+1}$ reflexiv ist.

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

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

Also ist $\sim$ eine Äquivalenzrelation auf *Type*. □

Definition 4.6 Mit *TypeRel* bezeichnen wir die Menge der binären Relationen auf *Type*, also $\mathit{TypeRel} = \mathcal{P}(\mathit{Type} \times \mathit{Type})$.

Die vorangegangene Definition der Relation $\sim$ ist zwar einfach und intuitiv leicht verständlich, allerdings ist es schwierig, anhand dieser Definition Aussagen über $\sim$ zu formulieren oder zu beweisen. Um dieses Problem zu lösen, definieren wir nachfolgend eine Abbildung $\mathcal{E}$ auf Typrelationen, welche einfacher anzuwendende Aussagen für bestimmte Typrelationen enthält und beweisen anschließend, dass diese Aussagen auf $\sim$ zutreffen.

Definition 4.7 Sei $\mathcal{E} : \mathit{TypeRel} \to \mathit{TypeRel}$ definiert durch

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

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. Wir müssen also zunächst beweisen, dass *TypeRel* eine dcpo ist, und die Abbildung $\mathcal{E}$ auf $(\mathit{TypeRel}, \supseteq)$ 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) $(\mathit{TypeRel}, \supseteq)$ *ist eine dcpo mit kleinstem Element* $\mathit{Type} \times \mathit{Type}$.

(b) $\mathcal{E}$ *ist eine stetige Abbildung auf* $(\mathit{TypeRel}, \supseteq)$.

Beweis:

(a) Folgt unmittelbar nach Lemma 1.1.

(b) Zunächst zeigen wir, dass $\mathcal{E}$ eine monotone Abbildung auf $(\mathit{TypeRel}, \supseteq)$ ist. Seien dazu $R_1, R_2 \in \mathit{TypeRel}$ mit $R_1 \supseteq R_2$. Es ist zu 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 $\mathit{root}(\tau)$:

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

Dann gilt $\mathit{root}(\tau) = \mathit{root}(\tau')$, und nach Definition 4.7 gilt $(\tau, \tau') \in \mathcal{E}(R)$ für alle $R \in \mathit{TypeRel}$, also insbesondere $(\tau, \tau') \in \mathcal{E}(R_1)$.

2.) $\mathit{root}(\tau) = \rightarrow$

Dann gilt $\mathit{root}(\tau) = \mathit{root}(\tau')$ und $\big(\mathit{child}_i(\tau), \mathit{child}_i(\tau')\big) \in R_2$ für $i = 1, 2$. Da $R_2 \subseteq R_1$ folgt unmittelbar $\big(\mathit{child}_i(\tau), \mathit{child}_i(\tau')\big) \in R_1$ und somit nach Definition 4.7 auch $(\tau, \tau') \in \mathcal{E}(R_1)$.

3.) $\mathit{root}(\tau) = \langle m_1; \ldots; m_k \rangle$

Folgt völlig analog.

Bleibt noch zu zeigen, dass $\mathcal{E}$ stetig ist. $\mathcal{E}$ ist stetig, wenn

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

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

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

Klar.

2.) $\mathit{root}(\tau) = \rightarrow$

Es gilt wieder $\mathit{root}(\tau) = \mathit{root}(\tau')$ und $\big(\mathit{child}_i(\tau), \mathit{child}_i(\tau')\big) \in R$ für $i = 1, 2$ und alle $R \in \Delta$. Also gilt insbesondere $\big(\mathit{child}_i(\tau), \mathit{child}_i(\tau')\big) \in \bigcap \Delta$ für $i = 1, 2$, und wegen Definition 4.7 folgt somit $(\tau, \tau') \in \mathcal{E}(\bigcap \Delta)$.

3.) $\mathit{root}(\tau) = \langle m_1; \ldots; m_k \rangle$

Folgt wieder analog zum vorangegangenen Fall. □

Damit nun lässt sich das zentrale Lemma über die Typrelation $\sim$ formulieren, welches einen einfacheren Umgang mit der Relation $\sim$ ermöglicht, so dass man nicht länger zeigen muss, dass eine bestimmte Eigenschaft, die für $\sim$ bewiesen werden soll, für alle $\sim_n$ gilt.

Lemma 4.3 (Fixpunkt-Lemma für $\mathcal{L}_o^{rt}$) *Die Relation* $\sim$ *ist bezüglich* $\supseteq$ *der kleinste Fixpunkt der Funktion* $\mathcal{E}$.

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

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

- $n = 0$

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

- $n \rightsquigarrow n + 1$

  Wegen Induktionsvoraussetzung gilt $\sim_n = \mathcal{E}^n(\mathit{Type} \times \mathit{Type})$, 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. $\mathit{root}(\tau) = \mathit{root}(\tau')$ und $(\mathit{child}_i(\tau), \mathit{child}_i(\tau')) \in \sim_n$ für $i = 1, \ldots, \mathit{arity}(\tau)$. Mit Fallunterscheidung nach der Form von $\mathit{root}(\tau)$ 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 $\mathit{root}(\tau)$ zu zeigen, dass auch $(\tau, \tau') \in \sim_{n+1}$ gilt, was aus offensichtlichen Gründen ziemlich trivial ist.

Insgesamt folgt dann $\sim = \bigcap_{n\in\mathbb{N}} \mathcal{E}^n(\mathit{Type} \times \mathit{Type}) = \bigsqcup_{n\in\mathbb{N}} \mathcal{E}^n(\mathit{Type} \times \mathit{Type})$. □

Korollar 4.1 *$\sim$ ist bezüglich $\subseteq$ der größte Fixpunkt von $\mathcal{E}$.*

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) $\tau = \mu t_1. \ldots \mu t_n.\beta$ *und* $\hat{\tau} = \mu\hat{t}_1. \ldots \mu\hat{t}_m.\beta$ *mit* $\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

(b) $\tau = \mu t_1. \ldots \mu t_n.\tau_1 \to \tau_2$ *und* $\hat{\tau} = \mu\hat{t}_1. \ldots \mu\hat{t}_m.\hat{\tau}_1 \to \hat{\tau}_2$ *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) $\tau = \mu t_1. \ldots \mu t_n.\langle m_1 : \tau_1; \ldots; m_k : \tau_k\rangle$ *und* $\hat{\tau} = \mu\hat{t}_1. \ldots \mu\hat{t}_m.\langle m_1 : \hat{\tau}_1; \ldots; m_k : \hat{\tau}_k\rangle$ *mit* $\tau_j[{}^{\tau}/_{t_i}]_{i=1}^n \sim \hat{\tau}_j[{}^{\hat{\tau}}/_{\hat{t}_i}]_{i=1}^m$ *für alle* $j = 1, \ldots, k$

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

Beweis: Folgt unmittelbar aus dem Fixpunkt-Lemma (Lemma 4.3), da die Aussagen des Lemmas mit der Definition von $\mathcal{E}$ übereinstimmen (Definition 4.7). □

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

Definition 4.8 (Gültige Typurteile für $\mathcal{L}_o^{rt}$) Ein Typurteil der Form $\Gamma \rhd_e e :: \tau$ oder $\Gamma \rhd_r r :: \phi$ heißt *gültig* für $\mathcal{L}_o^{rt}$, wenn es sich mit den Typregeln der Programmiersprache $\mathcal{L}_o^t$ aus Definition 2.20 mit Ausnahme der (Object)-Regel, sowie den folgenden neuen Typregeln

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

$$\text{(Equiv)} \quad \frac{\Gamma \rhd_e e :: \tau' \quad \tau' \sim \tau}{\Gamma \rhd_e e :: \tau}$$

herleiten lässt.

Mit Hilfe der neuen (Equiv)-Regel kann also für Ausdrücke in Typherleitungen zu einem äquivalenten Typ übergegangen werden. Das bedeutet letztlich, μ-Typen können bei Bedarf aufgefaltet oder zusammengefaltet werden.

Für Reihen ist keine derartige Regel vorhanden, da schon auf Reihentypen keine Äquivalenzrelation definiert ist. Entsprechend muss jedoch die (Object)-Regel angepasst werden: Die neue Typregel (Object') fordert nicht länger syntaktische Übereinstimmung des angegebenen Typs mit dem aus dem Reihentyp konstruierten Objekttyp. Stattdessen wird hier lediglich Äquivalenz der Typen gefordert.

Durch die (Equiv)-Regel geht, wie zuvor durch (Subsume) bei $\mathcal{L}_o^{sub}$, die Typeindeutigkeit der Programmiersprache verloren, da nun auf jede syntaktische Form eines Ausdrucks jeweils die (Equiv)-Regel und die eigentlich für diesen Ausdruck vorgesehene Regel angewandt werden kann. Für $\mathcal{L}_o^{rt}$ gilt darüber hinaus, dass sich für jeden Ausdruck (abzählbar) unendlich viele syntaktische unterschiedliche Typen herleiten lassen, die alle äquivalent zueinander sind (sofern *TName* mindestens abzählbar unendlich ist).

4.1.1 Typsicherheit

Zum Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_o^{rt}$ formulieren wir zunächst die obligatorischen Lemmata für die neue Programmiersprache.

Lemma 4.5 (Typumgebungen und frei vorkommende Namen)

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

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

Beweis: Entspricht dem Beweis des Lemmas für die Programmiersprache $\mathcal{L}_o^t$ (Lemma 2.5), wobei der neue Fall der (Equiv)-Regel direkt mit Induktionsvoraussetzung folgt. □

Lemma 4.6 (Koinzidenzlemma für $\mathcal{L}_o^{rt}$)

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

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

Beweis: Wiederum wie im Beweis des Lemmas für die Programmiersprache $\mathcal{L}_o^t$ (Lemma 2.6), und der Fall der (Equiv)-Regel folgt ebenfalls direkt mit Induktionsvoraussetzung. □

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

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

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

Beweis: Wie im Beweis des Lemmas für die Programmiersprache $\mathcal{L}_o^t$ (Lemma 2.7) zeigen wir die Behauptung mittels simultaner Induktion über die Länge der Herleitung der Typurteile $\Gamma[{}^{\tau}/_{\mathit{id}}] \triangleright e' :: \tau'$ und $\Gamma[{}^{\tau}/_{\mathit{id}}] \triangleright r :: \phi$, mit Fallunterscheidung nach der letzten angewandten Typregel. Wir betrachten dabei nur die Fälle der (Equiv)- und (Object')-Regeln:

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

Nach Voraussetzung existiert ein $\tau'' \in \mathit{Type}$ mit $\Gamma[{}^{\tau}/_{\mathit{id}}] \triangleright e' :: \tau''$ und $\tau'' \sim \tau'$. Mit Induktionsvoraussetzung folgt dann

$$\Gamma \triangleright e'[{}^{e}/_{\mathit{id}}] :: \tau'',$$

und wegen $\tau'' \sim \tau'$ mit Typregel (Equiv) das erwartete Ergebnis $\Gamma \triangleright e'[{}^{e}/_{\mathit{id}}] :: \tau'$.

2.) $\Gamma[{}^{\tau}/_{\mathit{id}}] \triangleright \mathbf{object}\,(\mathit{self} : \tau')\ r\ \mathbf{end} :: \tau'$ mit Typregel (Object')

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

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

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

und da $\mathit{self} \notin \mathit{Attribute} \cup \mathit{Var}$ folgt weiter

$$\Gamma^\star[{}^{\tau'}/_{\mathit{self}}][{}^{\tau}/_{\mathit{id}}] \triangleright r :: \phi.$$

Darauf lässt sich nun die Induktionsvoraussetzung anwenden, und wir erhalten

$$\Gamma^\star[{}^{\tau'}/_{\mathit{self}}] \triangleright r[{}^{e}/_{\mathit{id}}] :: \phi,$$

woraus sich wegen $\langle \phi \rangle \sim \tau'$ mit Typregel (Object') schließlich

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

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

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

entspricht.

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

Lemma 4.8 (Typurteile und *self*-Substitution) *Sei $\Gamma \in TEnv$, $self \in Self$, $\tau \in Type$ und $r \in Row$. Dann gilt:*

(a) Wenn für $e \in Exp$ und $\tau' \in Type$

1. $\Gamma[^{\tau}/_{self}] \rhd e :: \tau'$,
2. $\Gamma^\star \rhd \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau$ *und*
3. $\forall a \in dom(\Gamma) \cap Attribute, \tau_a \in Type : \Gamma^\star \rhd r(a) :: \tau_a \Leftrightarrow \Gamma \rhd a :: \tau_a$

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

(b) Wenn für $r' \in Row$ und $\phi \in RType$

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

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

Beweis: Der Beweis entspricht im Wesentlichen wieder dem Beweis des Lemmas für die Programmiersprache $\mathcal{L}_o^t$ (Lemma 2.9). Der neue Fall der (Equiv)-Regel folgt leicht mit Induktionsvoraussetzung, der Fall der (Object')-Regel folgt unmittelbar mit Lemma 4.6 (Koinzidenzlemma). □

Das obligatorische *Inversion-Lemma* überspringen wir an dieser Stelle, da die Formulierung exakt der Formulierung des Lemmas für die Programmiersprache $\mathcal{L}_o^{sub}$ entspricht (vgl. Lemma 3.8). Stattdessen wenden wir uns unmittelbar dem Typerhaltungssatz zu:

Satz 4.1 (Typerhaltung, „Preservation“)

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

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

Beweis: Der Beweis erfolgt analog zum Beweis der Typerhaltung für die Programmiersprache $\mathcal{L}_o^{sub}$ (Satz 3.1) durch simultane Induktion über die Länge der Typherleitungen. Der neue Fall der (Equiv)-Regel entspricht dem Fall der (Subsume)-Regel für $\mathcal{L}_o^{sub}$. Die Betrachtungen der $\sim$-Relation entsprechen den Betrachtungen der Subtyprelation im Beweis der Typerhaltung für $\mathcal{L}_o^{sub}$. □

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-Theorems benutzen werden.

Lemma 4.9 (Canonical Forms) *Sei $v \in Val$, $\tau \in Type$, und gelte $[\,] \rhd v :: \tau$.*

(a) Wenn $\tau \sim \mathbf{int}$*, dann gilt* $v \in \mathit{Int}$*.*

(b) Wenn $\tau \sim \tau_1 \to \tau_2$*, dann gilt eine der folgenden Aussagen:*

1.) $v \in \mathit{Op}$

2.) $v = \mathit{op}\, v_1$ *mit* $\mathit{op} \in \mathit{Op}$ *und* $v_1 \in \mathit{Val}$

3.) $v = \lambda x : \tau_1'.e$ *mit* $x \in \mathit{Var}$*,* $\tau_1' \in \mathit{Type}$ *und* $e \in \mathit{Exp}$

(c) Wenn $\tau \sim \langle \phi \rangle$*, dann gilt* $v = \mathbf{object}\,(\mathit{self} : \tau')\ \omega\ \mathbf{end}$*.*

Beweis: Die Beweise der einzelnen Aussagen des Lemmas erfolgen wie üblich durch Induktion über die Länge der Typherleitungen. Im einzelnen folgen die Behauptungen dann leicht mit Induktionsvoraussetzung und Äquivalenz-Lemma (Lemma 4.4). □

Damit sind nun alle Voraussetzungen gegeben, um das Progress-Theorem beweisen zu können. Der Beweis ist im Wesentlichen identisch mit dem Beweis des Progress-Theorems für die Programmiersprache $\mathcal{L}_o^{sub}$ (Satz 3.2), da die wesentlichen Details, in denen sich die Sprachen unterscheiden, bereits in den vorangegangenen Lemmata behandelt wurden.

Satz 4.2 (Existenz des Übergangsschritts, „Progress“)

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

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

Beweis: Simultane Induktion über die Länge der Herleitung der Typurteile $[\,] \triangleright e :: \tau$ und $\Gamma^+ \triangleright r :: \phi$ mit Fallunterscheidung nach der zuletzt angewandten Typregel. Wir betrachten dazu die folgenden Fälle:

1.) $[\,] \triangleright e_1 \# m :: \tau$ mit Typregel (Send).

Das kann nur aus $[\,] \triangleright e_1 :: \langle m : \tau;\ \phi \rangle$ folgen, und nach Induktionsvoraussetzung gilt entweder $e_1 \in \mathit{Val}$ oder es existiert ein $e_1' \in \mathit{Exp}$ mit $e_1 \to e_1'$.

Sei also $e_1 \in \mathit{Val}$, dann gilt nach Lemma 4.9, dass $e_1 = \mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end}$. Also existiert ein small step

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

mit small step Regel (Send-Unfold).

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

$$e_1\#m \to e_1'\#m$$

mit small step Regel (Send-Eval).

2.) $\Gamma^+ \triangleright \textbf{val}\ a = e'; r' :: \phi$ mit Typregel (ATTR).

Das Typurteil kann nur aus Prämissen der Form $(\Gamma^+)^\star \triangleright e' :: \tau'$ und $\Gamma^+ \triangleright r' :: \phi'$ folgen, wobei zu beachten gilt, dass $(\Gamma^+)^\star = [\,]$. Nach Induktionsvoraussetzung gilt dann entweder $e' \in \mathit{Val}$ oder es existiert ein $e'' \in \mathit{Exp}$ mit $e' \to e''$. Ebenso gilt entweder $r' \in \mathit{RVal}$ oder existiert ein $r'' \in \mathit{Row}$ mit $r' \to r''$.

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

Falls $e' \notin \mathit{Val}$, also $e' \to e''$, so existiert ein small step

$$\textbf{val}\ a = e'; r' \to \textbf{val}\ a = e''; r'$$

mit small step Regel (ATTR-LEFT).

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

$$\textbf{val}\ a = e'; r' \to \textbf{val}\ a = e'; r''$$

mit Regel (ATTR-RIGHT).

Die restlichen Fälle verlaufen ähnlich wie im Beweis des Satzes für die Programmiersprachen $\mathcal{L}_o^t$ und $\mathcal{L}_o^{sub}$ (vgl. Satz 2.5 und 3.2). □

Mit diesen Ergebnissen können wir die Typsicherheit der Programmiersprache $\mathcal{L}_o^{rt}$ beweisen, da sich der Beweis des nachfolgenden Satzes analog zum Beweis des Safety-Theorems der Sprache $\mathcal{L}_o^t$ (Satz 2.6) 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. □

4.1.2 Typalgorithmus

Ähnlich wie zuvor für die Programmiersprache $\mathcal{L}_o^{sub}$ lässt sich auch für die Programmiersprache $\mathcal{L}_o^{rt}$ ein äquivalenter Typkalkül formulieren, welcher deterministisch ist. Die Regeln ähneln dabei den Typregeln in Definition 3.5 (Gültige Typurteile für $\mathcal{L}_o^m$), wobei allerdings statt der (SUBSUME)-Regel von $\mathcal{L}_o^{sub}$ die (EQUIV)-Regel in die anderen Typregeln integriert wird.

Zum Beweis, dass für die Typrelation der Programmiersprache $\mathcal{L}_o^{rt}$ ein Entscheidungsalgorithmus existiert, bleibt zu zeigen, dass die Relation $\sim$ entscheidbar ist. Dazu geben wir zunächst, wie zuvor für die Typ- und Subtyprelationen, ein Regelwerk an, mit dem sich Urteile über die Äquivalenz von zwei Typen herleiten lassen.

Definition 4.9 (Gültige Äquivalenzurteile) Eine Formel der Gestalt $A \vdash \tau \sim \tau'$ heißt *Äquivalenzurteil*, mit $A \subseteq \mathit{TypeRel}$ und $\tau, \tau' \in \mathit{Type}$. Ein Äquivalenzurteil heißt gültig, wenn es sich mit den Regeln

$$(\textsc{Asm}) \quad \frac{(\tau, \tau') \in A}{A \vdash \tau \sim \tau'}$$

$$(\textsc{Equ}) \quad \frac{root\,(\tau) = root\,(\tau') \quad A \cup \{(\tau, \tau')\} \vdash child_i\,(\tau) \sim child_i\,(\tau') \text{ f.a. } i = 1 \ldots arity\,(\tau)}{A \vdash \tau \sim \tau'}$$

herleiten lässt, wobei die (Asm)-Regel Vorrang vor der (Equ)-Regel hat.

Eine Formel der Gestalt $A \vdash \tau \sim \tau'$ liest man als „τ ist äquivalent zu τ' unter der Bedingung, dass die in A aufgeführten Äquivalenzen gelten“. Es ist ziemlich offensichtlich, dass

$$\tau \sim \tau' \quad \Leftrightarrow \quad \emptyset \vdash \tau \sim \tau'$$

für alle $\tau, \tau' \in \mathit{Type}$ gilt.

Weiter ist offensichtlich, dass sich gemäß dem Regelwerk ein Algorithmus formulieren lässt, welcher mit Rückwärtsanwendung von Äquivalenzregeln arbeitet. Hier bleibt dann lediglich zu zeigen, dass der Algorithmus in jedem Fall terminiert. Nehmen wir dazu an, die Herleitung für ein Äquivalenzurteil wäre unendlich. Das kann nur dann der Fall sein, wenn die (Equ)-Regel unendlich oft angewendet worden ist. Das bedingt aber, dass es unendlich viele unterschiedliche (τ, τ')-Paare gibt, die in A aufgenommen werden können.

Um also zu zeigen, dass der Algorithmus in jedem Fall terminiert, genügt es zu zeigen, dass in einer Herleitung für $\emptyset \vdash \tau \sim \tau'$ stets nur endlich viele unterschiedliche Typpaare enstehen können. Mit anderen Worten ist also zu zeigen, dass jeder Typ nur endlich viele verschiedene Nachkommen hat.

Definition 4.10 (Nachkommen von Typen) Die Menge $desc_n\,(\tau)$ der n-ten Nachkommen des Typs τ ist wie folgt induktiv definiert

$$\begin{aligned} desc_0\,(\tau) &= \{\tau\} \\ desc_{n+1}\,(\tau) &= \bigcup_{i=1}^{arity\,(\tau)} desc_n\,(child_i\,(\tau)), \end{aligned}$$

und darauf aufbauend ist die Menge aller Nachkommen von τ durch

$$desc\,(\tau) \;=\; \bigcup_{n \in \mathbb{N}} desc_n\,(\tau)$$

definiert.

Es ist ziemlich offensichtlich, dass während der Herleitung von $\emptyset \vdash \tau \sim \tau'$ nur Voraussetzungen aus der Menge $desc\,(\tau) \times desc\,(\tau')$ auftreten können. Entsprechend genügt es zu zeigen, dass die Menge $desc\,(\tau)$ für alle $\tau \in \mathit{Type}$ endlich ist. Zum Beweis der Endlichkeit definieren wir, basierend auf der syntaktischen Struktur von Typen, eine Menge von (strukturellen) Teiltypen. Hierzu benötigen wir zunächst den Begriff der Substitution auf Typen (vgl. [Sie04]):

Definition 4.11 (Substitution)

(a) Eine *Substitution* ist eine totale Abbildung $s : \mathit{TName} \to \mathit{Type}$ für die gilt: Die Menge $\{t \in \mathit{TName} \mid s(t) \neq t\}$ ist endlich.

(b) Sei $\tau \in \mathit{Type}$ und s eine Substitution. Dann bezeichnet $\tau\, s$ den Typ, der aus τ entsteht, indem jeder freie Typname t in τ durch $s(t)$ ersetzt wird.

(c) Die Menge aller Substitutionen bezeichnen wir mit *TSubst*.

Für Substitutionen verwenden wir die übliche Schreibweise. D.h. wenn s eine Substitution ist mit $s(t_i) = \tau_i$ für $i = 1, \ldots, n$ und $s(t) = t$ für alle $t \notin \{t_1, \ldots, t_n\}$, dann schreiben wir $s = [{}^{\tau_1}/_{t_1}, \ldots, {}^{\tau_n}/_{t_n}]$.

Definition 4.12 Sei $\tau \in \mathit{Type}$ und $s \in \mathit{TSubst}$. Die Menge $\mathit{Desc}\,(\tau, s) \subseteq \mathit{Type} \times \mathit{TSubst}$ ist wie folgt induktiv definiert über die Struktur von τ:

$$\begin{array}{rcl}
\mathit{Desc}\,(\beta, s) & = & \{(\beta, s)\} \quad \text{für } \beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\} \cup \mathit{TName} \\
\mathit{Desc}\,(\tau_1^{raw} \to \tau_2^{raw}, s) & = & \mathit{Desc}\,(\tau_1^{raw}, s) \cup \mathit{Desc}\,(\tau_2^{raw}, s) \cup \{(\tau_1^{raw} \to \tau_2^{raw}, s)\} \\
\mathit{Desc}\,(\langle m_1 : \tau_1^{raw}; \ldots; m_n : \tau_n^{raw}\rangle, s) & = & \bigcup_{i=1}^{n} \mathit{Desc}\,(\tau_i^{raw}, s) \cup \{(\langle m_1 : \tau_1^{raw}; \ldots; m_n : \tau_n^{raw}\rangle, s)\} \\
\mathit{Desc}\,(\mu t.\tau^{raw}, s) & = & \mathit{Desc}\,(\tau^{raw}, s \circ [{}^{\mu t.\tau^{raw}}/_t]) \cup \{(\mu t.\tau^{raw}, s)\}
\end{array}$$

Offensichtlich gilt der folgende einfache Zusammenhang zwischen rekursiven Typen und Substitutionen:

Lemma 4.10 *Sei $\tau \in \mathit{Type}^{raw}$ und $t \in \mathit{TName}$. Dann gilt:*

$$\{\tau^{raw}\, s \mid (\tau^{raw}, s) \in \mathit{Desc}\,(\tau[{}^{\mu t.\tau}/_t], [\,])\} \quad \subseteq \quad \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in \mathit{Desc}\,(\tau, [{}^{\mu t.\tau}/_t])\}$$

Beweis: Trivial. □

Da $\mathit{Desc}\,(\tau, s)$ induktiv über die syntaktische Struktur von τ definiert ist, ist die Menge trivialerweise endlich. Wenn wir also zeigen können, dass $\big|\mathit{desc}\,(\tau)\big| \leq k \big|\mathit{Desc}\,(\tau, [\,])\big|$ für alle $\tau \in \mathit{Type}$ und ein beliebiges $k > 1$ gilt, so wäre bewiesen, dass $\mathit{desc}\,(\tau)$ stets endlich ist. Damit wiederum wäre gezeigt, dass der obige Algorithmus in jedem Fall terminiert.

Lemma 4.11 $\forall \tau \in \mathit{Type} :\ \mathit{desc}\,(\tau) \subseteq \{\hat{\tau}^{raw}\, s \mid (\hat{\tau}^{raw}, s) \in \mathit{Desc}\,(\tau, [\,])\}$

Beweis: Da nach Definition $\mathit{desc}\,(\tau) = \bigcup_{n \in \mathbb{N}} \mathit{desc}_n\,(\tau)$ ist, genügt es zu zeigen, dass

$$\mathit{desc}_n\,(\tau) \quad \subseteq \quad \{\hat{\tau}^{raw}\, s \mid (\hat{\tau}^{raw}, s) \in \mathit{Desc}\,(\tau, [\,])\}$$

für alle $n \in \mathbb{N}$ gilt. Diese Aussage beweisen wir durch Induktion über (n, m) mit lexikographischer Ordnung, wobei m die Anzahl der führenden μ's in τ ist:

Für $n = 0$ gilt die Behauptung trivialerweise. Im Induktionsschritt unterscheiden wir nach der syntaktischen Form von τ:

- $\tau = \beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

  Dann gilt $\mathit{arity}\,(\beta) = 0$, also $\mathit{desc}_{n+1}\,(\beta) = \emptyset$.

- $\tau = \tau_1 \to \tau_2$

  Nach Definition von desc_{n+1} gilt

  $$\mathit{desc}_{n+1}\,(\tau_1 \to \tau_2) \quad = \quad \mathit{desc}_n\,(\tau_1) \cup \mathit{desc}_n\,(\tau_2),$$

  worauf sich dann die Induktionsvoraussetzung anwenden lässt, und wir folgern

  $$\begin{array}{l}
  \mathit{desc}_{n+1}\,(\tau_1 \to \tau_2) \\
  \subseteq \{\tau_1^{raw}\, s \mid (\tau_1^{raw}, s) \in \mathit{Desc}\,(\tau_1, [\,])\} \cup \{\tau_2^{raw}\, s \mid (\tau_2^{raw}, s) \in \mathit{Desc}\,(\tau_2, [\,])\} \\
  = \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in \mathit{Desc}\,(\tau_1, [\,]) \cup \mathit{Desc}\,(\tau_2, [\,])\} \\
  \subseteq \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in \mathit{Desc}\,(\tau_1, [\,]) \cup \mathit{Desc}\,(\tau_2, [\,]) \cup \{(\tau_1 \to \tau_2, [\,])\}\} \\
  = \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in \mathit{Desc}\,(\tau_1 \to \tau_2, [\,])\},
  \end{array}$$

  was zu zeigen war.

- $\tau = \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle$

  Folgt analog zum vorangegangenen Fall über Funktionstypen.

- $\tau = \mu t.\hat{\tau}$

  Nach Definition von $desc_{n+1}$ gilt

  $$desc_{n+1}\,(\mu t.\hat{\tau}) = desc_{n+1}\,(\hat{\tau}[{}^{\mu t.\hat{\tau}}/_t]),$$

  und mit Induktionsvoraussetzung folgt daraus

  $$\subseteq \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in Desc\,(\hat{\tau}[{}^{\mu t.\hat{\tau}}/_t], [\,])\}.$$

  Nach Lemma 4.10 folgt weiter

  $$\begin{aligned}
  &\subseteq \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in Desc\,(\hat{\tau}, [{}^{\mu t.\hat{\tau}}/_t])\} \\
  &\subseteq \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in Desc\,(\hat{\tau}, [{}^{\mu t.\hat{\tau}}/_t]) \cup \{(\mu t.\hat{\tau}, [\,])\}\} \\
  &= \{\tau^{raw}\, s \mid (\tau^{raw}, s) \in Desc\,(\mu t.\hat{\tau}, [\,])\},
  \end{aligned}$$

  was zu zeigen war. □

Da $Desc\,(\tau, [\,])$ stets endlich ist, ist somit gezeigt, dass auch $desc\,(\tau)$ stets endlich ist, und somit muss ein Algorithmus, der nach dem in Definition 4.9 angegebenen Regelwerk arbeitet, stets nach endlich vielen Schritten terminieren.

4.2 Die Sprache $\mathcal{L}_o^{srt}$

Abschließend wollen wir betrachten, in wie weit sich die beiden bisher vorgestellten Erweiterungen des Typsystems – Subtyping und rekursive Typen – in einer objekt-orientierten Programmiersprache kombinieren lassen.

Dazu definieren wir zunächst eine Relation $\lesssim$, welche die Eigenschaften der Relationen $\leq$ und $\sim$ in sich vereinigt. Genauergesagt verallgemeinern wir die Subtypregeln (Sm-Refl), (Sm-Arrow) und (Sm-Object) des Minimal Typkalküls aus Definition 3.3 auf die zuvor eingeführte Baumsemantik für rekursive Typen:

Definition 4.13 Die Relationen $\lesssim_n$ seien wie folgt induktiv definiert

$$\begin{aligned}
\lesssim_0 &= Type^2 \\
\lesssim_{n+1} &= \{(\tau, \tau') \in Type^2 \mid root\,(\tau) = root\,(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}\} \\
&\cup \{(\tau, \tau') \in Type^2 \mid root\,(\tau) = root\,(\tau') = \to \\
&\qquad \wedge\, child_1\,(\tau') \lesssim_n child_1\,(\tau) \,\wedge\, child_2\,(\tau) \lesssim_n child_2\,(\tau')\} \\
&\cup \{(\tau, \tau') \in Type^2 \mid root\,(\tau) = \langle m_1; \ldots; m_{k+l} \rangle \,\wedge\, root\,(\tau') = \langle m_1; \ldots; m_k \rangle \\
&\qquad \wedge \forall 1 \leq i \leq k :\, child_i\,(\tau) \lesssim_n child_i\,(\tau')\},
\end{aligned}$$

und die Relation $\lesssim$ ist durch

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

definiert, also als Durchschnitt aller Relationen $\lesssim_n$.

Wie üblich betrachten wir nun zunächst Ordnungseigenschaften der Relation $\lesssim$ bevor wir zur Erweiterung des Typsystems kommen.

Lemma 4.12 *Die Relation $\lesssim$ ist eine Quasiordnung.*

Beweis: Es genügt zu zeigen, dass jede Relation $\lesssim_n$ eine Quasiordnung ist, was ziemlich offensichtlich ist. □

Im Gegensatz zur Subtyprelation in Kapitel 3 (vgl. Definition 3.1) begnügen wir uns in diesem Fall mit einer Quasiordnung, da die Eigenschaft der Antisymmetrie für den Beweis der Typsicherheit nicht zwingend erforderlich ist. Stattdessen gilt aber der folgende Zusammenhang zwischen $\sim$ und $\lesssim$, welcher im weiteren Sinne einer Antisymmetrie entspricht:

Lemma 4.13 $\forall \tau, \tau' \in \mathit{Type} : \ \tau \lesssim \tau' \wedge \tau' \lesssim \tau \ \Leftrightarrow \ \tau \sim \tau'$

Beweis: Es genügt zu zeigen, dass die Aussage

$$\forall \tau, \tau' \in \mathit{Type} : \ \tau \lesssim_n \tau' \wedge \tau' \lesssim_n \tau \ \Leftrightarrow \ \tau \sim_n \tau'$$

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 $\sim_0 = \mathit{Type}^2 = \lesssim_0$.

- $n \rightsquigarrow n + 1$

  Für $\tau \sim_{n+1} \tau'$ muss nach Voraussetzung

  $$\mathit{root}(\tau) = \mathit{root}(\tau')$$

  und

  $$\forall 1 \leq i \leq \mathit{arity}(\tau) : \ \mathit{child}_i(\tau) \sim_n \mathit{child}_i(\tau')$$

  gelten. Nach Induktionsvoraussetzung folgt daraus, dass

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

  gilt. Damit folgt insgesamt die Behauptung mit einer einfachen Fallunterscheidung nach der Form von $\mathit{root}(\tau)$ und Anwenden der Definition von $\lesssim_{n+1}$ (Definition 4.13).

„$\Rightarrow$“ Folgt ebenso leicht durch vollständige Induktion über n. □

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

Korollar 4.2 $\forall \tau, \tau' \in \mathit{Type} : \ \tau \lesssim \tau' \wedge \tau' \lesssim \tau \ \Rightarrow \ \tau \sim \tau'$

Um die Relation $\lesssim$ leichter handhaben zu können, beweisen wir nun, wie zuvor für die Sprache $\mathcal{L}_o^{rt}$, dass $\lesssim$ Fixpunkt einer generierenden Abbildung $\mathcal{S}$ ist, die nachfolgend definiert ist:

Definition 4.14 Sei $\mathcal{S} : \mathit{TypeRel} \to \mathit{TypeRel}$ definiert durch

$$\begin{array}{rcl}
\mathcal{S}(R) & = & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \mathit{root}\,(\tau') \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}\} \\
& \cup & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \mathit{root}\,(\tau') = \to \\
& & \quad \wedge\, (\mathit{child}_1\,(\tau'), \mathit{child}_1\,(\tau)) \in R \wedge (\mathit{child}_2\,(\tau), \mathit{child}_2\,(\tau')) \in R\} \\
& \cup & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \langle m_1;\ldots;m_{k+l}\rangle \wedge \mathit{root}\,(\tau') = \langle m_1;\ldots;m_k\rangle \\
& & \quad \wedge \forall 1 \le i \le k : (\mathit{child}_i\,(\tau), \mathit{child}_i\,(\tau')) \in R\},
\end{array}$$

also eine totale Abbildung auf binären Typrelationen.

Zum Beweis, dass $\lesssim$ Fixpunkt der Abbildung $\mathcal{S}$ ist, müssen zunächst die Voraussetzungen des Fixpunktsatzes erfüllt sein: Für die Programmiersprache $\mathcal{L}_o^{rt}$ hatten wir bereits bewiesen, dass $(\mathit{TypeRel}, \supseteq)$ eine dcpo mit einem kleinsten Element $\mathit{Type} \times \mathit{Type}$ ist (vgl. Lemma 4.2). Für die Anwendung des Fixpunktsatzes von Kleene bleibt also lediglich zu zeigen, dass die Abbildung $\mathcal{S}$ stetig ist.

Lemma 4.14 *$\mathcal{S}$ ist eine stetige Abbildung auf $(\mathit{TypeRel}, \supseteq)$.*

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 $(\mathit{TypeRel}, \supseteq)$ ist. Seien dazu wieder $R_1, R_2 \in \mathit{TypeRel}$ 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 $\mathit{root}\,(\tau)$:

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

Dann ist $\mathit{root}\,(\tau) = \mathit{root}\,(\tau')$, und nach Definition 4.14 gilt $(\tau,\tau') \in \mathcal{S}(R)$ für alle $R \in \mathit{TypeRel}$, also insbesondere $(\tau,\tau') \in \mathcal{S}(R_1)$.

2.) $\mathit{root}\,(\tau) = \to$

Nach Definition 4.14 gilt wieder $\mathit{root}\,(\tau) = \mathit{root}\,(\tau')$, $\big(\mathit{child}_1\,(\tau'), \mathit{child}_1\,(\tau)\big) \in R_2$ sowie $\big(\mathit{child}_2\,(\tau), \mathit{child}_2\,(\tau')\big) \in R_2$. Also insbesondere $\big(\mathit{child}_1\,(\tau'), \mathit{child}_1\,(\tau)\big) \in R_1$ und $\big(\mathit{child}_2\,(\tau), \mathit{child}_2\,(\tau')\big) \in R_1$, denn es gilt $R_2 \subseteq R_1$, und somit folgt die Behauptung unmittelbar nach Definition.

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

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 $\Delta \subseteq \mathit{TypeRel}$ gilt. Sei also wieder $\Delta \subseteq \mathit{TypeRel}$ 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 $\mathit{root}\,(\tau)$ zeigen wir, dass dann auch $(\tau,\tau') \in \mathcal{S}(\bigcap \Delta)$ gilt:

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

Klar.

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

Dann ist $root\,(\tau) = root\,(\tau')$ und für alle $R \in \Delta$ gilt $\big(child_1\,(\tau'), child_1\,(\tau)\big) \in R$ sowie $\big(child_2\,(\tau), child_2\,(\tau')\big) \in R$. Das bedeutet aber insbesondere, dass dann $\big(child_1\,(\tau'), child_1\,(\tau)\big), \big(child_2\,(\tau), child_2\,(\tau')\big) \in \bigcap \Delta$ gilt, also folgt nach Definition 4.14, dass auch $(\tau, \tau') \in \mathcal{S}(\bigcap \Delta)$ gilt.

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

Folgt wieder analog zum vorherigen Fall. □

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

Lemma 4.15 (Fixpunkt-Lemma für $\mathcal{L}_o^{srt}$) *Die Relation $\lesssim$ ist bezüglich $\supseteq$ der kleinste Fixpunkt der Funktion $\mathcal{S}$.*

Beweis: Der Beweis erfolgt analog zum Beweis des Fixpunkt-Lemmas für die Programmiersprache $\mathcal{L}_o^{rt}$ (Lemma 4.3). □

Korollar 4.3 *Die Relation $\lesssim$ ist bezüglich $\subseteq$ der größte Fixpunkt von $\mathcal{S}$.*

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 4.16 (Subtyping-Lemma für $\mathcal{L}_o^{srt}$) *$\tau \lesssim \tau'$ gilt genau dann, wenn eine der folgenden Aussagen zutrifft:*

(a) $\tau \sim \tau' \sim \beta$ *mit* $\beta \in \{\mathbf{bool}, \mathbf{int}, \mathbf{unit}\}$

(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) $\tau \sim \langle m_1 : \tau_1; \ldots; m_{k+l} : \tau_{k+l} \rangle$ *und* $\tau' \sim \langle m_1 : \tau_1'; \ldots; m_k : \tau_k' \rangle$ *mit* $\tau_i \lesssim \tau_i'$ *für alle* $i = 1, \ldots, k$

Beweis: Folgt unmittelbar aus der Tatsache, dass die Subtyprelation $\lesssim$ Fixpunkt der stetigen Abbildung $\mathcal{S}$ ist (Lemma 4.15), denn die Aussagen des Lemmas entsprechen genau der Definition der Abbildung $\mathcal{S}$ (Definition 4.14). □

Bevor wir nun die Typsicherheit der Programmiersprache $\mathcal{L}_o^{srt}$ betrachten, müssen wir natürlich erst 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 4.15 (Gültige Typurteile für $\mathcal{L}_o^{srt}$) Ein Typurteil der Form $\Gamma \rhd_e e :: \tau$ oder $\Gamma \rhd_r r :: \phi$ heißt *gültig* für $\mathcal{L}_o^{srt}$, wenn es sich mit den Typregeln der Programmiersprache $\mathcal{L}_o^{rt}$ aus Definition 4.8 mit Ausnahme der (Equiv)-Regel sowie der neuen Typregel

$$\text{(Subsume')} \quad \frac{\Gamma \rhd_e e :: \tau' \quad \tau' \lesssim \tau}{\Gamma \rhd_e e :: \tau}$$

herleiten lässt.

4.2.1 Typsicherheit

Intuitiv lässt sich bereits vermuten, dass die Programmiersprache $\mathcal{L}_o^{srt}$ ebenfalls typsicher ist, da das Typsystem im Wesentlichen durch Zusammenführung der Typsysteme der Programmiersprachen $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$ entstanden ist. Allerdings ist diese Eigenschaft deshalb noch nicht zwingend trivial ersichtlich, denn es existieren Programme, die zwar in $\mathcal{L}_o^{srt}$ wohlgetypt sind, jedoch weder in $\mathcal{L}_o^{sub}$ noch in $\mathcal{L}_o^{rt}$.

Also beweisen wir in diesem Abschnitt zunächst wieder 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 4.17 (Typumgebungen und frei vorkommende Namen)

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

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

Beweis: Wie zuvor verläuft der Beweis durch Induktion über die Länge der Herleitung der Typurteile. Die bekannten Fälle folgen wie gehabt, der neue Fall der (Subsume')-Regel folgt unmittelbar mit Induktionsvoraussetzung. □

Lemma 4.18 (Koinzidenzlemma für $\mathcal{L}_o^{srt}$)

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

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

Beweis: Ebenfalls wie gehabt, wobei der Fall der (Subsume')-Regel direkt mit Induktionsvoraussetzung folgt. □

Lemma 4.19 (Typurteile und Substitution) *Sei $id \in \mathit{Attribute} \cup \mathit{Var}$, $\Gamma \in \mathit{TEnv}$, $\tau \in \mathit{Type}$ und $e \in \mathit{Exp}$. Dann gilt:*

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

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

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

Lemma 4.20 (Typurteile und *self*-Substitution) *Sei $\Gamma \in \mathit{TEnv}$, $\mathit{self} \in \mathit{Self}$, $\tau \in \mathit{Type}$ und $r \in \mathit{Row}$. Dann gilt:*

(a) Wenn für $e \in \mathit{Exp}$ und $\tau' \in \mathit{Type}$

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

gilt, dann gilt auch $\Gamma \triangleright e[^{\mathbf{object}\,(self:\tau)\ r\ \mathbf{end}}/_{self}] :: \tau'$.

(b) Wenn für $r' \in Row$ *und* $\phi \in RType$

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

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

Beweis: Folgt wie zuvor im Beweis des Lemmas für die Programmiersprachen $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$ (vgl. Lemma 3.7 und 4.8). □

Auch das *Inversion-Lemma* muss für die Programmiersprache $\mathcal{L}_o^{srt}$ angepasst werden, wobei wir uns jeweils wieder auf die interessanten Aussagen beschränken:

Lemma 4.21 (Umkehrung der Typrelation)

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

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

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

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

Beweis: Wie zuvor durch Induktion über die Länge der Typherleitungen. □

Wie bereits im Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_o^{sub}$ 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) $\forall \Gamma \in TEnv : \forall e, e' \in Exp : \forall \tau \in Type : \Gamma^\star \triangleright e :: \tau \wedge e \to e' \Rightarrow \Gamma^\star \triangleright e' :: \tau$

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

Beweis: Der Beweis erfolgt selbstverständlich wieder durch simultane Induktion über die Länge der Typherleitungen von $\Gamma^\star \triangleright e :: \tau$ und $\Gamma \triangleright r :: \phi$, mit Fallunterscheidung nach der zuletzt angewendeten Typregel. Wir betrachten lediglich die folgenden Fälle:

1.) $\Gamma^\star \triangleright \mathbf{object}\,(self : \tau)\ r\ \mathbf{end} :: \tau$ mit Typregel (Object')

Das kann nur aus Prämissen der Form $(\Gamma^\star)^\star[^{\tau}/_{self}] \triangleright r :: \phi$ und $\langle\phi\rangle \sim \tau$ folgen. Der small step $\mathbf{object}\,(self : \tau)\ r\ \mathbf{end} \to e'$ kann andererseits nur mit der small step Regel (Object-Eval) hergeleitet worden sein, wobei nach Voraussetzung ein small step $r \to r'$ existiert und $e' = \mathbf{object}\,(self : \tau)\ r'\ \mathbf{end}$. Nach Induktionsvoraussetzung gilt also

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

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

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

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

1.) Für $e_1\#m \to e'$ mit (SEND-EVAL) existiert dann nach Voraussetzung ein small step $e_1 \to e_1'$ und es gilt $e' = e_1'\#m$. Das bedeutet, die Behauptung folgt direkt mit Induktionsvoraussetzung und Typregel (SEND).

2.) Falls $e_1\#m \to e'$ mit small step Regel (SEND-UNFOLD) hergeleitet worden ist, so muss gelten $e_1 = \mathbf{object}\,(\mathit{self} : \tau')\ \omega\ \mathbf{end}$ und $e' = \omega[{}^{e_1}/_{\mathit{self}}]\#m$. Nach dem Inversion-Lemma (Lemma 4.21) gilt

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

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

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

folgt. Damit sind alle Voraussetzungen für die Anwendung des speziellen Substitutions-Lemmas (Lemma 4.20) erfüllt, und wir erhalten

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

Da $\langle\phi'\rangle \sim \tau'$ und $\tau' \lesssim \langle m : \tau; \phi\rangle$ existieren also nach Lemma 4.16 (Subtyping-Lemma) $\tau'' \in \mathit{Type}$ und $\phi'' \in \mathit{RType}$ mit $\tau'' \lesssim \tau$ und $\langle\phi''\rangle \lesssim \langle\phi'\rangle$, so dass $\phi' = (m : \tau''; \phi'')$. Damit erhalten wir schließlich mit Typregel (SEND')

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

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

Die restlichen Fälle verlaufen ähnlich wie zuvor für die Programmiersprachen $\mathcal{L}_o^t$, $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$ (vgl. Satz 2.4, 3.1 und 4.1). □

Wir haben also gezeigt, dass auch im Typsystem der Programmiersprache $\mathcal{L}_o^{srt}$ small steps typerhaltend sind. Zum Beweis der Typsicherheit bleibt wie zuvor das Progress-Theorem zu zeigen. Dazu passen wir zunächst das *Canonical Forms Lemma* für die Sprache $\mathcal{L}_o^{srt}$ an.

Lemma 4.22 (Canonical Forms) *Sei $v \in \mathit{Val}$, $\tau \in \mathit{Type}$ und gelte $[\,] \triangleright v :: \tau$.*

(a) Wenn $\tau \lesssim \mathbf{int}$, dann gilt $v \in \mathit{Int}$.

(b) Wenn $\tau \lesssim \tau_1 \to \tau_2$, dann gilt eine der folgenden Aussagen:

1.) $v \in \mathit{Op}$

2.) $v = \mathit{op}\, v_1$ mit $\mathit{op} \in \mathit{Op}$ und $v_1 \in \mathit{Val}$

3.) $v = \lambda x : \tau_1'.e$ mit $x \in \mathit{Var}$, $\tau_1' \in \mathit{Type}$ und $e \in \mathit{Exp}$

(c) Wenn $\tau \lesssim \langle\phi\rangle$, dann gilt $v = \mathbf{object}\,(\mathit{self} : \tau')\ \omega\ \mathbf{end}$.

Beweis: Wie zuvor durch Induktion über die Länge der Typherleitungen. □

Damit lässt sich nun auch das Progress-Theorem für die Programmiersprache $\mathcal{L}_o^{srt}$ aufstellen und beweisen. Die Formulierung des Theorems stimmt mit den vorangegangenen Formulierungen überein (vgl. Satz 3.2 und Satz 4.2), und auch der Beweis verläuft sehr ähnlich.

Satz 4.5 (Existenz des Übergangsschritts, „Progress“)

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

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

Beweis: Wir führen den Beweis wieder durch simultane Induktion über die Länge der Herleitung der Typurteile $[\,] \triangleright e :: \tau$ und $\Gamma^+ \triangleright r :: \phi$, mit Fallunterscheidung nach der jeweils zuletzt angewandten Typregel. Exemplarisch betrachten wir die folgenden Fälle:

1.) $[\,] \triangleright e_1\, e_2 :: \tau$ mit Typregel (App)

Das kann nur aus Prämissen der Form $[\,] \triangleright e_1 :: \tau_2 \to \tau$ und $[\,] \triangleright e_2 :: \tau_2$ folgen. Dann zeigen wir je nach Form der Teilausdrücke e_1 und e_2, dass entweder die Applikation bereits ein Wert ist, oder ein small step existiert.

Für $e_1 \notin \mathit{Val}$ lässt sich wie früher leicht zeigen, dass ein small step mit (App-Left) existiert. Ebenso einfach zeigt man, dass für $e_1 \in \mathit{Val}$ und $e_2 \notin \mathit{Val}$ ein small step mit (App-Right) existiert.

Es bleibt also der Fall $e_1, e_2 \in \mathit{Val}$ zu betrachten. Da $\lesssim$ reflexiv ist, lässt sich das Canonical Forms Lemma (4.22) anwenden, demzufolge drei Fälle zu unterscheiden sind:

1.) $e_1 \in \mathit{Op}$

Trivial.

2.) $e_1 = \mathit{op}\, v_1$ mit $\mathit{op} \in \mathit{Op}$ und $v_1 \in \mathit{Val}$

Dann existiert nach Lemma 4.21 ein $\tau_1 \in \mathit{Type}$ mit $[\,] \triangleright \mathit{op} :: \tau_1 \to \tau_2 \to \tau$ und $[\,] \triangleright v_1 :: \tau_1$. Wie früher folgt daraus leicht, dass $v_1, e_2 \in \mathit{Int}$, und somit ein small step $\mathit{op}\, v_1\, e_2 \to \mathit{op}^I(v_1, e_2)$ mit small step Regel (Op) existiert.

3.) $e_1 = \lambda x : \tau_2'.e_1'$ mit $x \in \mathit{Var}$, $\tau_2' \in \mathit{Type}$ und $e_1' \in \mathit{Exp}$

Dann ist trivialerweise ersichtlich, dass ein small step mit (Beta-V) möglich ist.

2.) $[\,] \triangleright e_1 \# m :: \tau$ mit Typregel (Send)

Nach Voraussetzung gilt $[\,] \triangleright e_1 :: \langle m : \tau; \phi \rangle$. Mit Induktionsvoraussetzung folgt, dass entweder $e_1 \in \mathit{Val}$ gilt oder ein small step $e_1 \to e_1'$ existiert.

Sei also $e_1 \in \mathit{Val}$, dann gilt nach Lemma 4.22, dass $e_1 = \mathbf{object}\,(\mathit{self} : \langle \phi \rangle)\ \omega\ \mathbf{end}$. Also existiert ein small step

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

mit small step Regel (Send-Unfold).

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

$$e_1 \# m \to e_1' \# m$$

mit small step Regel (SEND-EVAL).

Die restlichen Fälle folgen ähnlich wie zuvor im Beweis des Satzes für die Programmiersprachen $\mathcal{L}_o^{sub}$ und $\mathcal{L}_o^{rt}$ (vgl. Satz 3.2 und 4.2). □

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. □

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 also zu zeigen, dass auch die Relation $\lesssim$ entscheidbar ist. Dies ist ebenso leicht zu beweisen wie im Fall der Relation $\sim$, indem man zeigt, dass während der Herleitung eines Subtypingurteils nur endliche viele unterschiedliche Typpaare entstehen können.

5 Vererbung

Abschließend wollen wir noch einen weiteren wichtigen Aspekt objekt-orientierter Programmiersprachen betrachten: Vererbung. Darunter versteht man die Wiederverwendung von Programmcode in unterschiedlichen Klassen. Wir klären dazu zunächst, was unter einer Klasse zu verstehen ist. Booch definiert Klassen in [Boo91] wie folgt:

> *„Eine Klasse ist eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten besitzen.“*

Das Konzept der Klassen dient also dazu, Objekte nach ihren jeweiligen Fähigkeiten einzuteilen. Eine Klasse repräsentiert damit eine Abstraktion eines bestimmten Sachverhalts, während ein Objekt eine konkrete Instantiierung dieses Sachverhalts darstellt. Vereinfacht ausgedrückt dient eine Klasse als Konstruktionsskizze zur Erzeugung strukturell identischer Objekte.

Unter (Klassen-)Vererbung versteht man dann das Aufbauen neuer Klassen unter Verwendung bestehender Klassen. Hierzu werden Attribute und Methoden einer sogenannten Basisklasse bei der Definition einer neuen Klasse mitverwendet, so dass Instanzen dieser beiden Klassen auf der Signatur der Basisklasse überstimmen. Das gilt aber nicht für die Semantik der Objekte, da es erbenden Klassen möglich ist, geerbte Attribute und Methoden mit einer neuen Bedeutung zu versehen.

In einer Vererbungsbeziehung bezeichnet man die Basisklasse, also die Klasse von der geerbt wird, auch als *Superklasse* und die erbende Klasse wird als *Subklasse* bezeichnet. Man spricht dann statt von Vererbung auch häufig von *Subclassing*. Jede Superklasse kann beliebig viele Subklassen besitzen, ebenso wie in Programmiersprachen, die *mehrfaches Erben*[1] erlauben, jede Subklasse beliebig viele Superklassen besitzen kann.

Wichtig ist in diesem Zusammenhang anzumerken, dass die Begriffe *Superklasse* und *Subklasse* keinesfalls mit den aus den vorangegangenen Kapiteln bekannten Begriffen *Supertyp* und *Subtyp* verwechselt werden sollten. Bei Subclassing und Subtyping handelt es sich um zwei eigenständige Konzepte. Leider ist es heutzutage, bedingt durch den starken Einfluss von Programmiersprachen wie Java oder C++, in denen Subtyping nur implizit durch Subclassing möglich ist, üblich, die beiden Begriffe beliebig zu vermischen, so dass in vielen Fällen gar nicht mehr ersichtlich ist, was eigentlich gemeint ist.

Dieses Kapitel stellt das Konzept der Vererbung als Spracherweiterung der Programmiersprache $\mathcal{L}_o^{srt}$ vor. Grundsätzlich orientieren wir uns dazu wieder an den in [RV97], [RV98] und [Rém02] für O'Caml vorgestellten Lösungen, wobei wir allerdings die Sprache sehr einfach halten, und uns auf die grundlegenden theoretischen Aspekte konzentrieren. Der Hauptunterschied zu O'Caml ist, dass wir Klassen als *first-class-citizens* der Programmiersprache betrachten, statt als separaten Kalkül[2].

[1] In der deutschen Literatur findet man hier immer noch oft den durch fehlerhafte Übersetzung aus dem Englischen entstandenen Begriff der *Mehrfachvererbung*.

[2] Diese Einschränkung in O'Caml ist notwendig, da Klassen als einfache Ausdrücke in einem Typsystem mit ML-Polymorphie nicht ohne weiteres untergebracht werden können, ohne den Benutzerkomfort

5.1 Syntax der Sprache $\mathcal{L}_c$

Basierend auf diesen Überlegungen erweiteren wir die kontextfreie Grammatik von Ausdrücken um Klassen und einen **new**-Operator zur Instantiierung von Klassen. Zur Realisierung des Erbens von Basisklassen verallgemeinern wir das Konzept der Reihen zu sogenannten *Klassenrümpfen*. Ein Klassenrumpf besteht aus einer beliebigen Anzahl von **inherit**-Anweisungen, gefolgt von einer Reihe.

Definition 5.1 (Abstrakte Syntax von $\mathcal{L}_c$)

(a) Die Menge *Body* aller *Klassenrümpfe* b ist durch die folgende kontextfreie Grammatik definiert:

$$b \;::=\; \textbf{inherit}\ a_1, \ldots, a_k\ \textbf{from}\ e;\ b_1 \mid r$$

Für $a_1, \ldots, a_k$ schreiben wir kurz A und betrachten A gleichzeitig als Menge, also $A = \{a_1, \ldots, a_k\}$. Ist $A = \emptyset$, so schreiben wir kurz (**inherit** e; b_1).

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

$$e \;::=\; \textbf{class}\,(\mathit{self})\ b\ \textbf{end} \mid \textbf{new}\, e_1$$

Eine **inherit**-Anweisung ist dabei wie folgt zu verstehen: Der Ausdruck e wird ausgewertet, bis eine Klasse erreicht ist, die exakt die Attribute $a_1, \ldots, a_n$ enthält. Die in dieser (Basis-)Klasse enthaltene Reihe wird anschließend anstelle der **inherit**-Anweisung in die erbende Klasse eingesetzt. Sämtliche **inherit**-Anweisungen stehen stets am Anfang einer Klasse, also vor der eigentlichen Reihe.

In einem **new**-Ausdruck wird zunächst der Teilausdruck e_1 ausgewertet, bis eine Klasse erreicht ist, und anschließend wird aus der Klasse ein Objekt konstruiert, indem die Reihe dieser Klasse in ein neues Objekt eingesetzt wird. Wichtig ist hierbei, dass eine Klasse fertig ausgewertet ist, wenn sie nur noch eine Reihe enthält, d.h. wenn alle **inherit**-Anweisungen durch die Reihen aus den Basisklassen ersetzt wurden.

Gemäß der Definition ist klar, dass $\mathit{Row} \subseteq \mathit{Body}$ gilt, d.h. wenn etwas für alle Klassenrümpfe $b \in \mathit{Body}$ gezeigt wird, ist es damit auch für alle Reihen $r \in \mathit{Row}$ gezeigt.

Wie zuvor für Reihen fordern wir nun allgemeiner für Klassenrümpfe, dass alle Attribute in einem Klassenrumpf disjunkt sein müssen. Das schließt insbesondere die geerbten Attribute mit ein. Damit ist dann mehrfaches Erben von der gleichen Klasse ausgeschlossen. Diese Einschränkung erleichtert die Betrachtung der Semantik und des Typsystems erheblich.

Will man mehrfaches Erben von der gleichen Klasse oder mehrfaches Erben von Klassen mit gleichen Attributnamen zulassen, könnte man für **inherit**-Anweisungen ein Präfix einführen, so dass Attribute beim Erben automatisch mit dem Präfix versehen werden, und damit innerhalb der Subklasse eindeutig sind. Die Syntax hierfür würde dann wie folgt aussehen

$$\textbf{inherit}\ a_1, \ldots, a_n\ \textbf{from}\ e\ \textbf{as}\ s;\ b,$$

zu reduzieren.

wobei s ein spezieller *Superklassenname* aus einer entsprechenden Menge *Super* wäre. Die Attribute $a_1, \ldots, a_n$ der Basisklasse würden dann beim Erben zu $s.a_1, \ldots, s.a_n$ umbenannt, und wären somit eindeutig innerhalb der Subklasse, sofern der Name s eindeutig ist. Dies entspricht im Wesentlichen der Vorgehensweise bei C++, wobei dort die Identifizierung – statt über einen frei wählbaren Namen – über den Basisklassennamen erfolgt.

5.2 Operationelle Semantik der Sprache $\mathcal{L}_c$

Da Klassen als *first-class-citizens* in die Programmiersprache eingeführt werden, muss für Klassen eine entsprechende Wertsemantik definiert werden. Wie im vorangegangenen Abschnitt bereits angedeutet, müssen zumindest alle **inherit**-Anweisungen ausgewertet sein, bevor eine Klasse als fertig ausgewertet betrachtet werden kann.

Die dann entstehende Reihe darf aber innerhalb einer Klasse nicht ausgewertet werden. Eine Klasse wird damit als Wert betrachtet, wenn sie nur noch aus einer Reihe besteht.

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

$$v \quad ::= \quad \textbf{class}\,(\mathit{self})\ r\ \textbf{end}$$

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

Für eine rein funktionale Basissprache ohne Rekursion und Exceptions wäre auch eine Vorauswertung von Klassen zulässig. In diesem Fall würden die rechten Seiten der Attribute schon vor einer Instantiierung der Klasse ausgewertet, und instantiierte Objekte wären damit unmittelbar Werte.

5.2.1 Frei vorkommende Namen und Substitution

Für die neuen Formen von Ausdrücken ist sofort ersichtlich, wie die Definition der frei vorkommenden Namen zu erweitern ist. Bei **inherit**-Anweisungen hingegen ist dies nicht so offensichtlich. Zunächst ist hier zu beachten, dass alle geerbten Attribute als Bindungsmechanismen für den folgenden Klassenrumpf auftreten. Dies ist auch der Grund, warum die geerbten Attribute in der Subklasse explizit aufgeführt werden müssen[3].

Weiterhin ist *self* frei in **inherit**-Anweisungen. Grund hierfür ist, dass im Typsystem der Objekttyp der umgebenden Klasse benötigt wird (siehe (Inherit) in Definition 5.9), und somit zum Beweis des Koinzidenzlemmas für die neue Programmiersprache *self* frei in **inherit**-Anweisungen sein muss[4].

Definition 5.3 (Frei vorkommende Namen)

(a) Die Menge $\mathit{free}\,(e)$ aller im Ausdruck $e \in \mathit{Exp}$ *frei vorkommenden Namen* erhält man durch folgende Verallgemeinerung von Definition 2.5:

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

[3] In einer getypten Programmiersprache könnten diese Informationen, ähnlich wie in O'Caml, automatisch durch das Typsystem inferiert werden.

[4] Alternativ könnte auch das Koinzidenzlemma umformuliert werden.

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

$$\mathit{free}\,(\textbf{inherit}\ A\ \textbf{from}\ e;\ b) \quad = \quad \{\mathit{self}\} \cup \mathit{free}\,(e) \cup \mathit{free}\,(b) \setminus A$$

Die Menge $\mathit{Exp}^\star$ ist nach wie vor so definiert, dass Ausdrücke aus dieser Menge nur freie Vorkommen von Variablennamen enthalten dürfen, jedoch keine freien Vorkommen von Attribut- und Objektnamen (vgl. Definition 2.6). Die Reiheneinsetzung ist ebenfalls wie zuvor definiert (vgl. Definition 2.7).

Definition 5.4 (Substitution)

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

$$\begin{aligned} \textbf{class}\,(\mathit{self} : \tau)\ b\ \textbf{end}[{}^{e}/_{\mathit{id}}] \quad &= \quad \begin{cases} \textbf{class}\,(\mathit{self})\ b\ \textbf{end} & \text{falls } \mathit{id} = \mathit{self} \\ \textbf{class}\,(\mathit{self})\ b[{}^{e}/_{\mathit{id}}]\ \textbf{end} & \text{sonst} \end{cases} \\ (\textbf{new}\, e')[{}^{e}/_{\mathit{id}}] \quad &= \quad \textbf{new}\, e'[{}^{e}/_{\mathit{id}}] \end{aligned}$$

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

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

5.2.2 Small step Semantik

Die zuvor intuitiv dargestellte Semantik der Programmiersprache $\mathcal{L}_c$ wird nun formalisiert, indem small step Regeln definiert werden, mit denen Berechnungen von Programmen durchgeführt werden können:

Definition 5.5 (Gültige small steps für $\mathcal{L}_c$) Ein small step $e \to_e e'$ mit $e, e' \in \mathit{Exp}$, $b \to_b b'$ mit $b, b' \in \mathit{Body}$ oder $r \to_r r'$ mit $r, r' \in \mathit{Row}$, heißt *gültig* für $\mathcal{L}_c$, wenn er sich mit den small step Regeln von $\mathcal{L}_o$ (vgl. Definition 2.10), sowie den folgenden zusätzlichen small step Regeln für Ausdrücke

$$\text{(Class-Eval)} \quad \frac{b \to_b b'}{\textbf{class}\,(\mathit{self})\ b\ \textbf{end} \to_e \textbf{class}\,(\mathit{self})\ b'\ \textbf{end}}$$

$$\text{(New-Eval)} \quad \frac{e \to_e e'}{\textbf{new}\, e \to_e \textbf{new}\, e'}$$

$$\text{(New-Exec)} \quad \textbf{new}\,(\textbf{class}\,(\mathit{self})\ r\ \textbf{end}) \to_e \textbf{object}\,(\mathit{self})\ r\ \textbf{end}$$

und den neuen small step Regeln für Klassenrümpfe

$$\text{(INHERIT-RIGHT)} \quad \frac{b \to_b b'}{\textbf{inherit}\ A\ \textbf{from}\ e;\ b \to_b \textbf{inherit}\ A\ \textbf{from}\ e;\ b'}$$

$$\text{(INHERIT-LEFT)} \quad \frac{e \to_e e'}{\textbf{inherit}\ A\ \textbf{from}\ e;\ r \to_b \textbf{inherit}\ A\ \textbf{from}\ e';\ r}$$

$$\text{(INHERIT-EXEC)} \quad \textbf{inherit}\ A\ \textbf{from}\ (\textbf{class}\,(\mathit{self})\ r_1\ \textbf{end});\ r_2 \to_b r_1 \oplus r_2 \quad \text{falls } dom_a\,(r_1) = A$$

herleiten lässt.

Die Regel (CLASS-EVAL) wertet eine Klasse solange aus, bis an Stelle des Klassenrumpfs nur noch eine Reihe vorhanden ist. Anschließend ist die Klasse nach Definition 5.2 (Werte und Reihenwerte) ein Wert.

Zur Instantiierung von Klassen wird zunächst mit (NEW-EVAL) der Ausdruck rechts vom **new**-Operator ausgewertet. Ergibt sich für diesen Ausdruck anschließend ein Klassenwert, so erzeugt (NEW-EXEC) daraus ein neues Objekt.

Die Vererbung erfolgt in drei Schritten: Zunächst wird mittels (INHERIT-RIGHT) der restliche Klassenrumpf zu einer Reihe ausgewertet. Anschließend wertet (INHERIT-LEFT) den Ausdruck zu einem Klassenwert aus. Zuletzt setzt (INHERIT-EXEC) die Reihe der ausgewerteten Basisklasse in die Subklasse ein, so dass am Ende in der Subklasse nur noch eine Reihe steht, und diese damit ebenfalls ein Klassenwert ist.

Wichtig ist insbesondere, dass (INHERIT-EXEC) nur dann anwendbar ist, wenn die im **inherit**-Ausdruck angegebenen Attribute exakt mit den Attributen der Basisklasse übereinstimmen. Denn aus $dom_a\,(r_1) = A$ folgt $dom_a\,(r_1) \cap dom_a\,(r_2) = \emptyset$, also ist $r_1 \oplus r_2$ definiert (vgl. Definition 2.3). Aufgabe des Typsystems für $\mathcal{L}_c$ ist also u.a. sicher zu stellen, dass diese Bedingung stets erfüllt ist.

Intuitiv ist sofort ersichtlich, dass auch die erweiterte small step Semantik immer noch eindeutig ist. Der Beweis erfolgt wie zuvor, indem zunächst gezeigt wird, dass für Werte kein small step existiert, und anschließend allgemein, dass für jeden Ausdruck, jede Reihe und jeden Klassenrumpf höchstens ein small step existiert.

Lemma 5.1 (Werte und small steps)

(a) $v \not\to$ *für alle* $v \in \mathit{Val}$.

(b) $\omega \not\to$ *für alle* $\omega \in \mathit{RVal}$.

Beweis: Trivial. □

Satz 5.1 (Eindeutigkeit des Übergangsschritts)

(a) Für jeden Ausdruck $e \in \mathit{Exp}$ *existiert höchstens ein* $e' \in \mathit{Exp}$ *mit* $e \to_e e'$.

(b) Für jede Reihe $r \in \mathit{Row}$ *existiert höchstens ein* $r' \in \mathit{Row}$ *mit* $r \to_r r'$.

(c) Für jeden Klassenrumpf $b \in \mathit{Body}$ *existiert höchstens ein* $b' \in \mathit{Body}$ *mit* $b \to_b b'$.

Beweis: Leicht durch simultane Induktion über die Struktur von e, r und b zu beweisen. □

5.2.3 Big step Semantik

Für die Programmiersprache $\mathcal{L}_c$ lässt sich ähnlich wie zuvor in Abschnitt 2.2.3 für die Programmiersprache $\mathcal{L}_o$ eine äquivalente big step Semantik angeben:

$$\text{(Class)} \quad \frac{b \Downarrow_b r}{\textbf{class}\,(\mathit{self})\ b\ \textbf{end} \Downarrow_e \textbf{class}\,(\mathit{self})\ r\ \textbf{end}}$$

$$\text{(New)} \quad \frac{e \Downarrow_e \textbf{class}\,(\mathit{self})\ r\ \textbf{end}}{\textbf{new}\,e \Downarrow_e \textbf{object}\,(\mathit{self})\ r\ \textbf{end}}$$

$$\text{(Inherit)} \quad \frac{b \Downarrow_b r_2 \quad e \Downarrow_e \textbf{class}\,(\mathit{self})\ r_1\ \textbf{end} \quad \mathit{dom}_a\,(r_1) = A}{\textbf{inherit}\ A\ \textbf{from}\ e;\ b \Downarrow_b r_1 \oplus r_2}$$

Es sei dem Leser überlassen, sich davon zu überzeugen, dass diese Semantik tatsächlich äquivalent zur im vorangegangenen Abschnitt beschriebenen small step Semantik ist.

5.3 Typsystem

Betrachtet man die Syntax und Semantik der Programmiersprache $\mathcal{L}_c$, so wird schnell klar, dass die Hinzunahme von Klassen ein komplexeres Typsystem notwendig macht. Während bei Objekten im Typsystem das Prinzip des *information hiding* durchgesetzt wird, müssen für Klassen im Typsystem detailierte Informationen bereitgehalten werden, insbesondere müssen die Typen der enthaltenen Attribute nach außen sichtbar sein.

Noch wichtiger im Hinblick auf Klassen ist jedoch die Tatsache, dass Klassen auf unterschiedliche Weise verwendet werden können. Betrachten wir beispielsweise eine einfache Klasse:

$$\textbf{let}\,c_1 = \textbf{class}\,(\mathit{self})\ \textbf{method}\ \mathit{myself} = \mathit{self}\ \textbf{end}$$

Von dieser Klasse c_1 könnte nun mit **new** eine Instanz erstellt werden, und für das entstehende Objekt würde man mit den Typregeln der Sprache $\mathcal{L}_o^{rt}$ (ohne Subtyping) den Typ $\mu t.\langle \mathit{myself} : t\rangle$ herleiten. Rückschließend könnte man also der Methode *myself* in der Klasse c_1 genau diesen Typ zuordnen.

Natürlich könnte man von dieser Klasse c_1 auch erben. Betrachten wir dazu also die folgende Klasse:

$$\textbf{let}\,c_2 = \textbf{class}\,(\mathit{self})\ \textbf{inherit}\,c_1;\ \textbf{method}\ \mathit{one} = 1\ \textbf{end}$$

Für jede Instanz dieser Klasse würde man dann in $\mathcal{L}_o^{rt}$ (ohne Subtyping) den Typ $\mu t.\langle \mathit{myself} : t;\ \mathit{one} : \textbf{int}\rangle$ herleiten, welcher offensichtlich nicht mit dem Typ für Instanzen von c_1 übereinstimmt.

Die Methode *myself* ist also nur an einer einzigen Stelle definiert. Basierend auf Instanzen der Klassen c_1 und c_2 haben wir jedoch zwei unterschiedliche Typen für diese Methode hergeleitet. Der Grund hierfür ist der unterschiedliche Typ von *self* in c_1 und c_2. In O'Caml benutzt man aus diesem Grund ML-Polymorphie für die *self*-Typen von Klassen, d.h., *self* kann für die unterschiedlichen Verwendungszwecke dynamisch mit einem passenden Typ versehen werden, welcher das Gesamttypurteil gültig macht.

Wir werden für die Programmiersprache $\mathcal{L}_c^t$ zeigen, dass Subtyping-Polymorphie ausreichend ist, um dieses Problem zu lösen. Dazu belassen wir beim Erben den Typ von

self in der Basisklasse bestehen und fordern lediglich, dass die Subklasse im *self*-Typ kleiner ist als die Basisklasse. Diese Vorgehensweise entspricht im Ergebnis dem in Java oder C++ verwendeten Mechanismus (vgl. [AC96, S.23f]).

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

Das Typsystem der Programmiersprache $\mathcal{L}_o^{srt}$ muss um Klassentypen erweitert werden. Da wir Klassen als *first-class-citizens* der Programmiersprache eingeführt haben, müssen Klassentypen in die Menge *Type* aufgenommen werden.

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:

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

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

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

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 Reihentypen paarweise verschieden sein müssen. Für Objekttypen sind aber nach wie vor nur Reihentypen zugelassen, die ausschließlich Typen für Methoden enthalten.

(c) Des Weiteren 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. Darüber hinaus 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 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.

Ein Klassentyp $\zeta(\tau_1 \,:\, \phi)$ besteht aus zwei Komponenten: Dem Objekttyp τ_1, also dem Typ, der bei einer Instantiierung der Klasse dem entstehenden Objekt zugewiesen wird, und dem Reihentyp ϕ, welcher alle Typinformationen der Klasse enthält. Alternativ könnte man auch den Objekttyp implizit aus den Typinformationen der Klasse generieren, allerdings wäre es dann nicht möglich, sogenannte *abstrakte Basisklassen* oder

Interfaces zu typen[5]. Diese legen eine bestimmte Schnittstelle fest, ohne jedoch konkret alle Methoden der Schnittstelle zu implementieren. Beispielsweise spezifiziert der Typ

$$\zeta(\mu t.\langle \mathit{push} : \mathbf{int} \to t;\ \mathit{pop} : t;\ \mathit{top} : \mathbf{int};\ \emptyset\rangle\ :\ \emptyset)$$

eine gemeinsame Schnittstelle für Stack-Klassen. Eine (abstrakte) Basisklasse dieses Typs würde lediglich einen Teil der Schnittstelle erbender Klassen festlegen, jedoch keine Implementierung vorschreiben. Eine solche Basisklasse wird auch als *abstrakter Datentyp (ADT)* bezeichnet.

Wichtig hierbei ist, zu beachten, dass von derartigen abstrakten Basisklassen lediglich geerbt werden kann. Es dürfen keine Instanzen dieser Klassen konstruiert werden, da sonst die Programmiersprache nicht typsicher wäre.

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

Zur Definition des Typsystems der Programmiersprache $\mathcal{L}_c^t$ müssen wir die Äquivalenz- und Subtyprelationen um die neuen Klassentypen erweitern. Dazu verallgemeinern wir zunächst die Definition der Typkonstruktoren (vgl. Definition 4.3):

Definition 5.7 (Einfache Typen und Typkonstruktoren)

(a) Die Menge $\mathit{SType} \subseteq \mathit{Type}$ aller *einfachen Typen* τ_s ist definiert als Einschränkung der Menge *Type* aller Typen auf die primitiven Typen und die Klassentypen:

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

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

$$\mathit{TCons} \quad = \quad \mathit{SType} \cup \{\to\} \cup \{\langle m_1; \ldots; m_n\rangle\,|\,n \in \mathbb{N}, m_1, \ldots, m_n \in \mathit{Method}\},$$

wobei wie zuvor die Methodennamen in $\langle m_1; \ldots; m_n\rangle$ disjunkt sein müssen.

Die Idee der einfachen Typen $\tau_s \in \mathit{SType}$ ist, dass auf diesen Typen kein Subtyping definiert ist. Für die primitiven Datentypen war das bisher bereits der Fall. Neu hinzu kommt, dass auch auf Klassentypen kein Subtyping möglich ist. Der Grund dafür wird später im Beweis der Typsicherheit ersichtlich. Intuitiv kann man sich vorstellen, dass jegliche in Klassentypen enthaltene Typinformation immer erhalten bleiben muss. Anhand der Ergebnisse aus den zuvor betrachteten Sprachen ist klar, dass Subtyping auf Objekttypen für eine objekt-orientierte Programmiersprache ausreichend ist.

Die Funktionen *root*, *arity* und *child* aus Definition 4.4 sind dann lediglich durch einen Fall für Klassentypen zu erweitern. Da auf Klassentypen kein Subtyping möglich sein soll, sind diese wie primitive Typen zu behandeln. Die Relation $\sim$ ist dann wie zuvor für die Programmiersprache $\mathcal{L}_o^{rt}$ definiert (vgl. Definition 4.5). Die Subtyprelation $\lesssim$ aus Definition 4.13 muss wie nachfolgend dargestellt erweitert werden:

[5] Wir benutzen die aus C++ bekannte Vorgehensweise, zwischen abstrakten Basisklassen und Interfaces nicht zu unterscheiden (vgl. [Str00, S.331ff]).

Definition 5.8 Die Relationen $\lesssim_n$ werden wie zuvor induktiv durch

$$\begin{array}{rcl}
\lesssim_0 & = & \mathit{Type}^2 \\
\lesssim_{n+1} & = & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \mathit{root}\,(\tau') \in \mathit{SType}\} \\
& \cup & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \mathit{root}\,(\tau') = \to \\
& & \quad \wedge\, \mathit{child}_1\,(\tau') \lesssim_n \mathit{child}_1\,(\tau) \wedge \mathit{child}_2\,(\tau) \lesssim_n \mathit{child}_2\,(\tau')\} \\
& \cup & \{(\tau,\tau') \in \mathit{Type}^2 \mid \mathit{root}\,(\tau) = \langle m_1;\ldots;m_{k+l}\rangle \wedge \mathit{root}\,(\tau') = \langle m_1;\ldots;m_k\rangle \\
& & \quad \wedge \forall 1 \leq i \leq k :\ \mathit{child}_i\,(\tau) \lesssim_n \mathit{child}_i\,(\tau')\}
\end{array}$$

definiert, und die Relation $\lesssim$ ist wie zuvor durch

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

als Durchschnitt aller Relationen $\lesssim_n$ definiert.

Lemma 5.2 (Ordnungseigenschaften der Äquivalenz- und Subtyprelation)

(a) Die Relation $\sim$ ist eine Äquivalenzrelation.

(b) Die Relation $\lesssim$ ist eine Quasiordnung.

(c) $\forall \tau,\tau' \in \mathit{Type} :\ \tau \lesssim \tau' \wedge \tau' \lesssim \tau \;\Leftrightarrow\; \tau \sim \tau'$

Beweis: Analog zum Beweis der entsprechenden Lemmata für die Programmiersprache $\mathcal{L}_o^{rt}$ (Lemma 4.1) und die Programmiersprache $\mathcal{L}_o^{srt}$ (Lemma 4.12 und 4.13). □

Damit lässt sich für die neuen Typrelationen ein Äquivalenz- und ein Subtypinglemma formulieren. Die Beweise dieser Lemmata erfolgen wie zuvor, indem eine entsprechende generierende Funktion konstruiert wird, und gezeigt wird, dass die jeweilige Relation Fixpunkt dieser Funktion ist (vgl. Lemma 4.4 und 4.16):

Lemma 5.3 (Äquivalenz-Lemma für $\mathcal{L}_c^t$) *$\tau \sim \hat{\tau}$ gilt genau dann, wenn eine der folgenden Aussagen zutrifft:*

(a) $\tau = \mu t_1.\ldots\mu t_n.\tau_s$ *und* $\hat{\tau} = \mu\hat{t}_1.\ldots\mu\hat{t}_m.\tau_s$ *mit* $\tau_s \in \mathit{SType}$

(b) $\tau = \mu t_1.\ldots\mu t_n.\tau_1 \to \tau_2$ *und* $\hat{\tau} = \mu\hat{t}_1.\ldots\mu\hat{t}_m.\hat{\tau}_1 \to \hat{\tau}_2$ *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) $\tau = \mu t_1.\ldots\mu t_n.\langle m_1 : \tau_1;\ldots;m_{k+l} : \tau_{k+l}\rangle$ *und* $\hat{\tau} = \mu\hat{t}_1.\ldots\mu\hat{t}_m.\langle m_1 : \hat{\tau}_1;\ldots;m_k : \hat{\tau}_k\rangle$ *mit* $\tau_j[{}^{\tau}/_{t_i}]_{i=1}^{n} \sim \hat{\tau}_j[{}^{\hat{\tau}}/_{\hat{t}_i}]_{i=1}^{m}$ *für alle* $j = 1,\ldots,k$

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* $\tau_s \in \mathit{SType}$

(b) $\tau \sim \tau_1 \to \tau_2$ *und* $\tau' \sim \tau_1' \to \tau_2'$ *mit* $\tau_1' \lesssim \tau_1$ *und* $\tau_2 \lesssim \tau_2'$

(c) $\tau \sim \langle m_1 : \tau_1;\ldots;m_{k+l} : \tau_{k+l}\rangle$ *und* $\tau' \sim \langle m_1 : \tau_1';\ldots;m_k : \tau_k'\rangle$ *mit* $\tau_i \lesssim \tau_i'$ *für alle* $i = 1,\ldots,k$

Basierend auf den erweiterten Relationen $\sim$ und $\lesssim$ können wir nun Typregeln für die Programmiersprache $\mathcal{L}_c^t$ aufstellen, um die zuvor intuitiv beschriebene Funktionsweise des Typsystems zu formalisieren:

Definition 5.9 (Gültige Typurteile für $\mathcal{L}_c^t$) Ein Typurteil $\Gamma \rhd_e e :: \tau$ oder $\Gamma \rhd_r b :: \phi$ 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

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

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

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

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

$$\text{(Class)} \quad \frac{\Gamma^\star[\tau/_{self}] \rhd_r b :: \phi \quad \tau \lesssim \langle methods\,(\phi) \rangle}{\Gamma \rhd_e \textbf{class}\,(self : \tau)\ b\ \textbf{end} :: \zeta(\tau\ :\ \phi)}$$

$$\text{(New)} \quad \frac{\Gamma \rhd_e e :: \zeta(\tau\ :\ \phi) \quad \tau \sim \langle methods\,(\phi) \rangle}{\Gamma \rhd_e \textbf{new}\, e :: \tau}$$

$$\text{(Subsume')} \quad \frac{\Gamma \rhd_e e :: \tau' \quad \tau' \lesssim \tau}{\Gamma \rhd_e e :: \tau}$$

und den verallgemeinerten Typregeln für Reihen und Klassenrümpfe

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

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

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

$$\text{(Inherit)} \quad \frac{\Gamma \rhd_e self :: \tau \quad \Gamma^\star \rhd_e e :: \zeta(\tau\ :\ \phi) \quad dom_a\,(\phi) = A \quad \Gamma[\phi(a)/_a]_{a \in A} \rhd_r b :: \phi'}{\Gamma \rhd_r \textbf{inherit}\ A\ \textbf{from}\ e;\ b :: \phi \oplus \phi'}$$

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.

Die Typregeln (Send), (Send'), (Dupl) und (Subsume') stimmen mit den gleichnamigen Typregeln der Programmiersprache $\mathcal{L}_o^{srt}$ überein und sind nur aus Gründen der Übersichtlichkeit hier aufgeführt. Die Typregel (Object") entspricht im Wesentlichen der (Object')-Regel, wobei hier darauf zu achten ist, dass nur die Typen der Methoden aus dem Reihentyp in den Typ des Objekts aufgenommen werden.

Wie bereits angedeutet, soll das Typsystem für $\mathcal{L}_c^t$ *abstrakte Basisklassen* zulassen. Folglich fordert die Typregel (Class) lediglich, dass alle im Klassenrumpf präsenten Methoden auch im Objekttyp aufgelistet sind. Umgekehrt müssen aber nicht alle im Objekttyp spezifizierten Methoden auch tatsächlich im Klassenrumpf vorhanden sein.

Bei der Instantiierung von Klassen muss allerdings dafür Sorge getragen werden, dass exakt die angegebenen Methoden im instantiierten Objekt vorhanden sind. Dies wird durch die zweite Prämisse der (New)-Regel sichergestellt.

Die Typregeln (Empty) und (Method) sind wieder wie zuvor definiert. Die (Attr')-Regel unterscheidet sich von der in den vorangegangenen Sprachen benutzten Typregel (Attr), da nun auch die Typen der Attribute in den Reihentyp aufgenommen werden.

Die tatsächlich interessante Neuerung ist die (Inherit)-Regel für das Subclassing. Durch die ersten beiden Prämissen wird sichergestellt, dass der Objekttyp der erbenden Klassen kleiner ist, als der Objekttyp der Basisklasse, und zwar durch die Formulierung der ersten Prämisse, die es erlaubt, bei der Bestimmung des *self*-Typs (Subsume') anzuwenden. Alternativ könnte die erste Prämisse auch umformuliert werden zu:

$$\Gamma(\mathit{self}) = \tau' \ \wedge \ \tau' \lesssim \tau$$

Die dritte Prämisse stellt sicher, dass die Attribute der Basisklasse, repräsentiert durch den Reihentyp ϕ der Basisklasse, exakt mit den in der **inherit**-Anweisung angegebenen Attributen übereinstimmen. Die letzte Prämisse überprüft dann, ob sich mit den geerbten Attributen für den restlichen Klassenrumpf ein Typ herleiten lässt.

5.3.3 Typsicherheit

Zum Beweis der Typsicherheit der Programmiersprache $\mathcal{L}_c^t$ muss zunächst wieder die Menge *Val* der Werte an die getypte Syntax angepasst werden, in dem die Produktion für Klassenwerte durch die neue Produktion

$$v \ ::= \ \textbf{class}\,(\mathit{self} : \tau)\ r\ \textbf{end}$$

ersetzt wird. Weiterhin müssen die small step Regeln (Class-Eval), (New-Exec) und (Inherit-Exec) an die getypte Syntax angepasst werden:

(Class-Eval)
$$\frac{b \to_b b'}{\textbf{class}\,(\mathit{self} : \tau)\ b\ \textbf{end} \to_e \textbf{class}\,(\mathit{self} : \tau)\ b'\ \textbf{end}}$$

(New-Exec)
$$\textbf{new}\,(\textbf{class}\,(\mathit{self} : \tau)\ r\ \textbf{end}) \to_e \textbf{object}\,(\mathit{self} : \tau)\ r\ \textbf{end}$$

(Inherit-Exec)
$$\textbf{inherit}\ A\ \textbf{from}\ (\textbf{class}\,(\mathit{self} : \tau)\ r_1\ \textbf{end});\ r_2 \to_b r_1 \oplus r_2 \quad \text{falls } \mathit{dom}_a\,(r_1) = A$$

Insbesondere wird bei der Instantiierung von Klassen der in der Klasse angegebene Objekttyp für das entstehende Objekt benutzt. Dies ist zulässig, denn nach Typregel (New) enthält der Objekttyp einer zu instantiierenden Klasse ausschließlich Methoden.

Die Typsicherheit beweisen wir wie üblich, indem wir zeigen, dass small steps typerhaltend sind, und dass ein abgeschlossener, wohlgetypter Ausdruck entweder bereits ein Wert ist, oder für diesen ein small step existiert.

Dazu formulieren wir zunächst die wohlbekannten Lemmata, wobei die Aussagen im einzelnen teilweise auf Klassenrümpfe verallgemeinert werden müssen. Die Beweise der Lemmata überspringen wir für $\mathcal{L}_c^t$, da diese völlig analog zu den bisher betrachteten Programmiersprachen verlaufen.

Lemma 5.5 (Typumgebungen und frei vorkommende Namen)

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

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

Beweis: Simultane Induktion über die Länge der Typherleitungen mit Fallunterscheidung nach der letzten Typregel. Die neuen Fälle folgen dabei unmittelbar nach Induktionsvoraussetzung, wobei zu beachten ist, dass *self* frei ist in **inherit**-Anweisungen. □

Lemma 5.6 (Koinzidenzlemma für $\mathcal{L}_c^t$)

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

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

Beweis: Auch hier erfolgt der Beweis wie üblich durch simultane Induktion über die Länge der Herleitung der Typurteile. Die Fälle für die neuen Typregeln folgen dabei leicht mit Induktionsvoraussetzung und Lemma 2.4. □

Lemma 5.7 (Typurteile und Substitution) *Sei $id \in \mathit{Id} \setminus \mathit{Self}$, $\Gamma \in \mathit{TEnv}$, $\tau \in \mathit{Type}$ und $e \in \mathit{Exp}$. Dann gilt:*

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

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

Beweis: Simultane Induktion über die Länge der Typherleitungen $\Gamma[{}^{\tau}/_{id}] \triangleright e' :: \tau'$ und $\Gamma[{}^{\tau}/_{id}] \triangleright b :: \phi$. Die Fälle für die neuen oder geänderten Typregeln folgen unmittelbar mit Induktionsvoraussetzung, Lemma 2.4 und dem Koinzidenzlemma für die Programmiersprache $\mathcal{L}_c^t$ (Lemma 5.6). □

Die Nutzung des speziellen Substitutionslemmas für *self* beschränkt sich nach wie vor auf das Auffalten von Objekten, so dass sowohl die Formulierung als auch der Beweis des Lemmas im Wesentlichen von den vorangegangenen Programmiersprachen übernommen werden können:

Lemma 5.8 (Typurteile und *self*-Substitution) *Sei $\Gamma \in \mathit{TEnv}$, $\mathit{self} \in \mathit{Self}$, $\tau \in \mathit{Type}$ und $r \in \mathit{Row}$. Dann gilt:*

(a) Wenn für $e \in \mathit{Exp}$ und $\tau' \in \mathit{Type}$

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

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

(b) Wenn für $r' \in \mathit{Row}$ und $\phi \in \mathit{RType}$

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

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

Insbesondere trifft das Lemma keine Aussage über *self*-Substitution in Klassenrümpfen, stattdessen beschränkt es sich ausschließlich auf Ausdrücke und Reihen. Die Idee dahinter ist sehr einfach: Klassenrümpfe 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: Erfolgt analog zum Beweis des Lemmas für die früheren Programmiersprachen. Für (Class) und (Object") ist dabei – wie bereits angedeutet – nichts zu zeigen. Für (New) und (Attr) folgt die Behauptung leicht mit Induktionsvoraussetzung und Lemma 5.5 (Typumgebungen und frei vorkommende Namen). □

Neben den zuvor aufgeführten, bekannten Lemmata benötigen wir zum Beweis der Typerhaltung für die Programmiersprache $\mathcal{L}_c^t$ zwei neue Lemmata. Das erste Lemma sichert dabei, dass der Übergang zu einem kleineren Typ für einen Eintrag in der Typumgebung keinen Einfluss auf den Typ hat, welcher für Ausdrücke und Klassenrümpfe hergeleitet werden kann:

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) $\forall e \in Exp, \hat{\tau} \in Type : \Gamma[^{\tau}/_{id}] \rhd e :: \hat{\tau} \ \Rightarrow\ \Gamma[^{\tau'}/_{id}] \rhd e :: \hat{\tau}$

(b) $\forall b \in Body, \phi \in RType : \Gamma[^{\tau}/_{id}] \rhd b :: \phi \ \Rightarrow\ \Gamma[^{\tau'}/_{id}] \rhd b :: \phi$

Beweis: Wir führen den Beweis wie üblich durch simultane Induktion über die Länge der Herleitung der Typurteile $\Gamma[^{\tau}/_{id}] \rhd e :: \hat{\tau}$ und $\Gamma[^{\tau}/_{id}] \rhd b :: \phi$, wobei wir jeweils nach der zuletzt angewandten Typregel in der Herleitung unterscheiden. Dazu betrachten wir folgende Fälle:

1.) $\Gamma[^{\tau}/_{id}] \rhd id' :: \hat{\tau}$ 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').

Anderenfalls, für $id \neq id'$, gilt $\Gamma[^{\tau}/_{id}](id') = \Gamma[^{\tau'}/_{id}](id') = \hat{\tau}$, also ist nichts zu zeigen.

2.) $\Gamma[^{\tau}/_{id}] \rhd e_1\, e_2 :: \hat{\tau}$ mit Typregel (App)

Das kann nur aus Prämissen der Form $\Gamma[^{\tau}/_{id}] \rhd e_1 :: \hat{\tau}_2 \to \hat{\tau}$ und $\Gamma[^{\tau}/_{id}] \rhd e_2 :: \hat{\tau}_2$ folgen. Nach Induktionsvoraussetzung gilt dann auch

$$\Gamma[^{\tau'}/_{id}] \rhd e_1 :: \hat{\tau}_2 \to \hat{\tau},$$

sowie

$$\Gamma[^{\tau'}/_{id}] \rhd e_2 :: \hat{\tau}_2.$$

Damit folgt unmittelbar

$$\Gamma[^{\tau'}/_{id}] \triangleright e_1\, e_2 :: \hat{\tau}$$

mit Typregel (App).

3.) $\Gamma[^{\tau}/_{id}] \triangleright$ **inherit** A **from** e; $b :: \phi \oplus \phi'$ mit Typregel (Inherit)

Dieses Typurteil kann ausschließlich aus den Voraussetzungen $\Gamma[^{\tau}/_{id}] \triangleright \mathit{self} :: \hat{\tau}$, $(\Gamma[^{\tau}/_{id}])^\star \triangleright e :: \zeta(\hat{\tau} : \phi)$, $\mathit{dom}_a(\phi) = A$ und $\Gamma[^{\tau}/_{id}][^{\phi(a)}/_a]_{a\in A} \triangleright b :: \phi'$ folgen. Für die erste Prämisse folgt unmittelbar mit Induktionsvoraussetzung, dass auch

$$\Gamma[^{\tau'}/_{id}] \triangleright \mathit{self} :: \hat{\tau}$$

gilt.

Falls $id \in \mathit{Var}$, so gilt $(\Gamma[^{\tau}/_{id}])^\star = \Gamma^\star[^{\tau}/_{id}]$, also folgt nach Induktionsvoraussetzung

$$\Gamma^\star[^{\tau'}/_{id}] \triangleright e :: \zeta(\hat{\tau} : \phi),$$

und wegen $\Gamma^\star[^{\tau'}/_{id}] = (\Gamma[^{\tau'}/_{id}])^\star$ folgt somit

$$(\Gamma[^{\tau'}/_{id}])^\star \triangleright e :: \zeta(\hat{\tau} : \phi).$$

Anderenfalls, für $id \in \mathit{Id} \setminus \mathit{Var}$, gilt $(\Gamma[^{\tau}/_{id}])^\star = \Gamma^\star = (\Gamma[^{\tau'}/_{id}])^\star$, 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

$$\Gamma[^{\tau'}/_{id}] \triangleright \textbf{inherit}\ A\ \textbf{from}\ e;\ b :: \phi \oplus \phi'$$

gilt, was zu zeigen war.

Die restlichen Fälle verlaufen ähnlich. □

Speziell zum Beweis der Typerhaltung benötigen wir für $\mathcal{L}_c^t$ noch ein weiteres Lemma, welches im Fall der small step Regel (Inherit-Exec) zum Einsatz kommt, die einen **inherit**-Klassenrumpf mit einer Klasse vereinfacht zu einer Konkatenation zweier Reihen. Entsprechend spezifiziert das Lemma, unter welchen Bedingungen sich für die durch Konkatenation enstandene Reihe ein bestimmter Typ herleiten lässt:

Lemma 5.10 (Reihenkonkatenation) *Seien $r_1, r_2 \in Row$ sowie $\phi_1, \phi_2 \in RType$ mit $dom_a(r_1) \cap dom_a(r_2) = \emptyset$ und $\phi_1 \oplus \phi_2$ definiert, und sei $\Gamma \in TEnv$ 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: Die Behauptung ist leicht durch Induktion über die Struktur von r_1 zu beweisen. □

Weiter müssen wir das *Inversion-Lemma* für die Programmiersprache $\mathcal{L}_c^t$ erweitern. Wie üblich beschränken wir uns dabei auf die für die nachfolgenden Beweise interessanten Aussagen:

Lemma 5.11 (Umkehrung der Typrelation)

(a) Wenn $\Gamma \triangleright id :: \tau$, dann gilt $\Gamma(id) = \tau'$ und $\tau' \lesssim \tau$.

(b) Wenn $\Gamma \triangleright \mathbf{class}\,(self : \tau)\ b\ \mathbf{end} :: \zeta(\tau' : \phi)$, dann gilt $\Gamma^\star[{}^{\tau}/_{self}] \triangleright b :: \phi$ und $\tau = \tau'$ mit $\tau \lesssim \langle methods\,(\phi)\rangle$.

Beweis: Klar. □

Dank des vorangegangenen Lemmas können nun Anwendungen der (Subsume')-Regel in den nachfolgenden Beweisen im Wesentlichen ignoriert werden. Damit kommen wir nun zum Beweis des Preservation-Theorems. Wichtig in diesem Fall ist, dass die small step Relationen $\rightarrow_r$ und $\rightarrow_b$ gesondert betrachtet werden müssen, entsprechend ändert sich die Aussage des Satzes:

Satz 5.2 (Typerhaltung, „Preservation")

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

(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 $\Gamma^\star \triangleright e :: \tau$, $\Gamma \triangleright r :: \phi$ und $\Gamma \triangleright b :: \phi$, mit Fallunterscheidung nach der zuletzt angewendeten Typregel:

1.) $\Gamma^\star \triangleright \mathbf{class}\,(self : \tau)\ b\ \mathbf{end} :: \zeta(\tau : \phi)$ mit Typregel (Class)

Dann gilt nach Voraussetzung $(\Gamma^\star)^\star[{}^{\tau}/_{self}] \triangleright b :: \phi$ mit $\tau \lesssim \langle methods\,(\phi)\rangle$. Dann kann der small step $\mathbf{class}\,(self : \tau)\ b\ \mathbf{end} \rightarrow_e e'$ ausschließlich mit (Class-Eval) hergeleitet worden sein, was wiederum bedingt, dass ein small step $b \rightarrow_b b'$ existiert und $e' = \mathbf{class}\,(self : \tau)\ b'\ \mathbf{end}$ gilt. Nach Induktionsannahme gilt also

$$(\Gamma^\star)^\star[{}^{\tau}/_{self}] \triangleright b' :: \phi,$$

woraus dann unmittelbar mit Typregel (Class) folgt, dass auch

$$\Gamma^\star \triangleright \mathbf{class}\,(self : \tau)\ b'\ \mathbf{end} :: \zeta(\tau : \phi)$$

gilt.

2.) $\Gamma^\star \triangleright \mathbf{new}\, e :: \tau$ mit Typregel (New)

Also müssen die Prämissen $\Gamma^\star \triangleright e :: \zeta(\tau : \phi)$ und $\tau \sim \langle \mathit{methods}\,(\phi)\rangle$ erfüllt sein. Der small step $\mathbf{new}\, e \to_e e'$ 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.) $\mathbf{new}\, e \to_e e'$ mit small step Regel (New-Eval)

Dann existiert ein $e'' \in \mathit{Exp}$, so dass $e' = \mathbf{new}\, e''$ und $e \to_e e'$ gilt. Die Behauptung folgt dann unmittelbar mit Induktionsvoraussetzung und Typregel (New).

2.) $\mathbf{new}\, e \to_e e'$ 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

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

gelten, mit $\tau' = \tau$. Wegen $\tau \sim \langle \mathit{methods}\,(\phi)\rangle$ folgt daraus mit Typregel (Object") schließlich die Behauptung

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

3.) $\Gamma \triangleright \mathbf{val}\ a = e;\ r :: (a : \tau; \emptyset) \oplus \phi$ mit Typregel (Attr')

Dann gilt nach Voraussetzung $\Gamma^\star \triangleright e :: \tau$ und $\Gamma[{}^{\tau}/_{a}] \triangleright r :: \phi$. Der small step $\mathbf{val}\ a = e;\ r \to_r r'$ kann nur mit einer der small step Regeln (Attr-Left) oder (Attr-Right) hergeleitet worden sein:

1.) $\mathbf{val}\ a = e;\ r \to_r r'$ mit small step Regel (Attr-Left)

Dann existiert ein small step $e \to_e e'$ und es muss $r' = (\mathbf{val}\ a = e';\ r)$ gelten. Dann folgt die Behauptung mit Induktionsvoraussetzung und Typregel (Attr').

2.) $\mathbf{val}\ a = e;\ r \to_r r'$ mit small step Regel (Attr-Right)

In diesem Fall existiert ein small step $r \to_r r''$ und es gilt $e \in \mathit{Val}$ und $r' = (\mathbf{val}\ a = e;\ r'')$. Hier folgt die Behauptung ebenfalls leicht mit Induktionsvoraussetzung und Typregel (Attr').

4.) $\Gamma \triangleright \mathbf{inherit}\ A\ \mathbf{from}\ e;\ b :: \phi \oplus \phi'$ mit Typregel (Inherit)

Das kann nur aus Prämissen der Form $\Gamma \triangleright \mathit{self} :: \tau$, $\Gamma^\star \triangleright e :: \zeta(\tau : \phi)$, $\mathit{dom}_a\,(\phi) = A$ und $\Gamma\big[{}^{\phi(a)}/_{a}\big]_{a\in A} \triangleright b :: \phi'$ folgen. Dann kann der small step $\mathbf{inherit}\ A\ \mathbf{from}\ e;\ b \to_b b'$ 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.) $\mathbf{inherit}\ A\ \mathbf{from}\ e;\ b \to_b b'$ mit small step Regel (Inherit-Left)

Dann existiert nach Voraussetzung ein small step $e \to_e e'$, und es gilt $b \in \mathit{Row}$ sowie $b' = \mathbf{inherit}\ A\ \mathbf{from}\ e';\ b$. Nach Induktionsannahme gilt

$$\Gamma^\star \triangleright e' :: \zeta(\tau : \phi)$$

und mit Typregel (INHERIT) folgt somit unmittelbar, dass auch

$$\Gamma \rhd \textbf{inherit}\ A\ \textbf{from}\ e'; b :: \phi \oplus \phi'$$

gilt.

2.) **inherit** A **from** e; $b \to_b b'$ mit small step Regel (INHERIT-RIGHT)

Dann muss gelten $b' = \textbf{inherit}\ A\ \textbf{from}\ e;\ b''$ und die Behauptung folgt ähnlich einfach, wie im vorangegangenen Fall.

3.) **inherit** A **from** e; $b \to_b b'$ mit small step Regel (INHERIT-EXEC)

Das bedingt $e = \textbf{class}\,(\mathit{self} : \hat{\tau})\ r_1\ \textbf{end}$, $b = r_2 \in \mathit{Row}$ und $b' = r_1 \oplus r_2$. Wegen Lemma 5.11 existiert ein $\tau' \in \mathit{Type}$ mit $\Gamma(\mathit{self}) = \tau'$ und $\tau' \lesssim \tau$. Ebenfalls wegen Lemma 5.11 gilt $\hat{\tau} = \tau$ und $(\Gamma^\star)^\star[{}^{\tau}/_{\mathit{self}}] \rhd r_1 :: \phi$. Hieraus lässt sich wegen $(\Gamma^\star)^\star[{}^{\tau}/_{\mathit{self}}] =_{\mathit{free}\,(r_1)} \Gamma[{}^{\tau}/_{\mathit{self}}]$ und Lemma 5.9 (Subtyping und Typumgebungen) herleiten, dass auch

$$\Gamma[{}^{\tau'}/_{\mathit{self}}] \rhd r_1 :: \phi$$

und somit, dass

$$\Gamma \rhd r_1 :: \phi$$

gilt. Darauf nun lässt sich Lemma 5.10 (Reihenkonkatenation) anwenden, und wir erhalten

$$\Gamma \rhd r_1 \oplus r_2 :: \phi \oplus \phi',$$

und damit ist die Behauptung gezeigt.

Die restlichen Fälle folgen wie in den Beweisen dieses Satzes für die vorangegangenen Programmiersprachen (vgl. Satz 2.4, 3.1, 4.1 und 4.4). □

Zum Beweis der Typsicherheit von $\mathcal{L}_c^t$ fehlt noch das Progress-Theorem. Dazu formulieren wir zunächst wieder ein *Canonical Forms Lemma* für $\mathcal{L}_c^t$, wobei wir uns auf die Aussage für die einzig neue Form eines Wertes beschränken:

Lemma 5.12 (Canonical Forms) *Sei $v \in \mathit{Val}$, $\tau \in \mathit{Type}$, und gelte $[\,] \rhd v :: \tau$.*

(a) Wenn $\tau \lesssim \zeta(\tau' : \phi')$, dann gilt $v = \textbf{class}\,(\mathit{self} : \tau')\ r\ \textbf{end}$.

Beweis:

(a) Nach Lemma 5.4 und Lemma 5.3 gilt $\tau = \mu t_1. \ldots \mu t_n.\zeta(\tau' : \phi')$, das heißt für das Typurteil $[\,] \rhd v :: \tau$ 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. □

Beim Progress-Theorem ist darauf zu achten, dass explizit zwischen Klassenrümpfen und Reihen unterschieden werden muss: Für (echte) Klassenrümpfe $b \in \mathit{Body} \setminus \mathit{Row}$ existiert keine Vorstellung von einem Wert, wie das für Ausdrücke und Reihen der Fall ist. Stattdessen werden Klassenrümpfe zu Reihen ausgewertet, entsprechend ist der Satz so formuliert, dass für echte Klassenrümpfe stets ein small step existieren muss.

Satz 5.3 (Existenz des Übergangsschritts, „Progress“)

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

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

(c) $\forall \Gamma \in \mathit{TEnv}, b \in \mathit{Body} \setminus \mathit{Row}, \phi \in \mathit{RType} : \Gamma^+ \triangleright b :: \phi \Rightarrow \exists b' \in \mathit{Body} : b \to_b b'$

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:

1.) $[\,] \triangleright \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} :: \tau$ mit Typregel (Object")

Nach Voraussetzung gilt also $[\mathit{self} : \tau] \triangleright r :: \phi$ mit $\tau \sim \langle \mathit{methods}\,(\phi) \rangle$, woraus nach Induktionsannahme folgt, dass r entweder bereits ein Reihenwert ist, oder ein small step $r \to_r r'$ existiert.

Gilt bereits $r \in \mathit{RVal}$, so gilt damit auch $(\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}) \in \mathit{Val}$.

Falls andererseits ein small step $r \to_r r'$ existiert, so lässt sich für den Gesamtausdruck ein small step $\mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end} \to_e \mathbf{object}\,(\mathit{self} : \tau)\ r'\ \mathbf{end}$ mit Regel (Object-Eval) herleiten.

2.) $[\,] \triangleright \mathbf{class}\,(\mathit{self} : \tau)\ b\ \mathbf{end} :: \zeta(\tau : \phi)$ mit Typregel (Class)

Das bedeutet, es gilt $[\mathit{self} : \tau] \triangleright b :: \phi$ mit $\tau \lesssim \langle \mathit{methods}\,(\phi) \rangle$. Dann müssen wir unterscheiden, ob b ein echter Klassenrumpf ist, oder ob b bereits eine Reihe ist.

Wenn $b \in \mathit{Row}$, so ist nichts zu zeigen, denn dann gilt $(\mathbf{class}\,(\mathit{self} : \tau)\ b\,\mathbf{end}) \in \mathit{Val}$.

Sollte $b \in \mathit{Body} \setminus \mathit{Row}$ gelten, so existiert nach Induktionsvoraussetzung ein small step $b \to_b b'$, also existiert insgesamt mit Regel (Class-Eval) ein small step $\mathbf{class}\,(\mathit{self} : \tau)\ b\ \mathbf{end} \to_e \mathbf{class}\,(\mathit{self} : \tau)\ b'\ \mathbf{end}$.

3.) $[\,] \triangleright \mathbf{new}\,e :: \tau$ mit Typregel (New)

Dann gilt $[\,] \triangleright e :: \zeta(\tau : \phi)$ mit $\tau \sim \langle \mathit{methods}\,(\phi) \rangle$. Nach Induktionsvoraussetzung ist e entweder bereits ein Wert, oder es existiert ein $e' \in \mathit{Type}$ mit $e \to_e e'$.

Falls e bereits ein Wert ist, so existiert nach Lemma 5.12 (Canonical Forms) ein $r \in \mathit{Row}$ mit $e = \mathbf{class}\,(\mathit{self} : \tau)\ r\ \mathbf{end}$. Dann existiert mit Regel (New-Exec) ein small step $\mathbf{new}\,(\mathbf{class}\,(\mathit{self} : \tau)\ r\ \mathbf{end}) \to_e \mathbf{object}\,(\mathit{self} : \tau)\ r\ \mathbf{end}$.

Existiert andererseits für e ein small step $e \to_e e'$, dann lässt sich mit Regel (New-Eval) ein small step $\mathbf{new}\,e \to_e \mathbf{new}\,e'$ herleiten.

4.) $\Gamma^+ \triangleright \mathbf{inherit}\ A\ \mathbf{from}\ e;\ b :: \phi \oplus \phi'$ mit Typregel (Inherit)

Nach Voraussetzung gilt $\Gamma^+ \triangleright \mathit{self} :: \tau$, $(\Gamma^+)^\star \triangleright e :: \zeta(\tau : \phi)$, $\mathit{dom}_a\,(\phi) = A$ 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 $b \in \mathit{Body} \setminus \mathit{Row}$, so existiert bereits nach Induktionsannahme ein $b' \in \mathit{Body}$ mit $b \to_b b'$. Also existiert insgesamt ein small step

$$\mathbf{inherit}\ A\ \mathbf{from}\ e;\ b \to_b \mathbf{inherit}\ A\ \mathbf{from}\ e;\ b'$$

mit Regel (INHERIT-RIGHT).

Anderenfalls ist b bereits eine Reihe. Wegen $(\Gamma^+)^\star = [\,]$ folgt dann nach Induktionsvoraussetzung, dass e entweder bereits ein Wert ist, oder ein small step $e \to_e e'$ existiert. Letzteres bedeutet, dass insgesamt für den Klassenrumpf ein small step **inherit** A **from** e; $b \to_b$ **inherit** A **from** e'; b mit small step Regel (INHERIT-LEFT) existiert.

Damit bleibt lediglich der Fall zu betrachten, dass $e \in \mathit{Val}$ und $b = r_2 \in \mathit{Row}$ gilt. Nach Lemma 5.12 (Canonical Forms) gilt dann wieder $e =$ **class** $(\mathit{self} : \tau)\ r_1$ **end**, und mit Lemma 5.11 folgt $[\mathit{self} : \tau] \triangleright r_1 :: \phi$. Offensichtlich gilt dann wegen $\mathit{dom}_a(\phi) = A$ auch $\mathit{dom}_a(r_1) = A$. Somit existiert insgesamt ein small step **inherit** A **from class** $(\mathit{self} : \tau)\ r_1$ **end**; $r_2 \to_b r_1 \oplus r_2$ mit Regel (INHERIT-EXEC).

Die übrigen Fälle verlaufen ähnlich zum Beweis des Satzes für die zuvor betrachteten Programmiersprachen (vgl. Satz 2.5, 3.2, 4.2 und 4.5). □

Damit ist dann gezeigt, dass auch die Programmiersprache $\mathcal{L}_c^t$ typsicher ist:

Satz 5.4 (Typsicherheit, „Safety“) *Wenn* $[\,] \triangleright e :: \tau$*, dann bleibt die Berechnung für* e *nicht stecken.*

Beweis: Klar. □

5.3.4 Typalgorithmus

Auch für die Programmiersprache $\mathcal{L}_c^t$ könnte man ähnlich wie im Fall der Sprache $\mathcal{L}_o^{sub}$ einen Minimal Typing Kalkül entwickeln, und für die erweiterten Relationen $\sim$ und $\lesssim$ ließe sich ebenfalls recht einfach zeigen, dass sie entscheidbar sind, so dass insgesamt klar ist, dass auch für $\mathcal{L}_c^t$ ein Algorithmus zur Typüberprüfung existiert. Eine detailierte Betrachtung dieses Algorithmus für $\mathcal{L}_c^t$ würde jedoch den Rahmen dieser Arbeit sprengen.

6 Schlussbemerkung

Wie haben also gezeigt, wie sich, basierend auf einer einfach getypten funktionalen Programmiersprache, eine (statisch) typsichere objekt-orientierte Programmiersprache entwickeln lässt. Dabei haben wir die Programmiersprache schrittweise um die wichtigsten objekt-orientieren Konzepte

- Subtyping,
- rekursive Typen
- und Subclassing

erweitert und jeweils betrachtet, wie sich diese Konzepte auf die Semantik und das Typsystem der Progammiersprache auswirken.

Insbesondere haben wir gezeigt, dass die Konzepte Subtyping und Subclassing weitgehend unabhängig voneinander sind, was sich in den gängigen objekt-orientierten Programmiersprachen nicht wiederspiegelt. Allerdings haben wir auch gesehen, dass ein Typsystem für eine Sprache mit Vererbung eine gewisse Form der Polymorphie voraussetzt. An dieser Stelle haben wir auf die bereits eingeführte Subtyping-Polymorphie zurückgegriffen.

Alternativ zur Subtyping-Polymorphie könnte man für das Typsystem einer Programmiersprache, welche Subclassing unterstützt, auch ML-Polymorphie verwenden, wie dies beispielsweise in O'Caml der Fall ist. Der O'Caml-Ansatz, bei dem der Typ für *self* polymorph gemacht wird, und an geeigneter Stelle in der Subklasse instantiiert wird (vgl. [Rém02]), ist, wie man sich leicht überlegen kann, echt mächtiger als der in diesem Dokument betrachtete Subtyping-Ansatz.

Entsprechend ist auf dem Gebiet der objekt-orientierten Typsysteme noch viel Bedarf an Forschung. Denn in den heute üblichen objekt-orientierten Programmiersprachen wird noch immer durchgehend Subtyping-Polymorphie verwendet, was unter anderem dafür sorgt, dass nicht typsichere Sprachelemente wie *Upcasts* notwendig sind, um bestimmte Probleme überhaupt sinnvoll lösen zu können[1].

Des Weiteren erscheint es an der Zeit, zu überlegen, ob Programmiersprachen wie Java oder C++ nicht von *structural typing* profitieren würden. Zwar wäre damit immer noch ein gewisser Overhead zur Compilezeit verbunden, allerdings ist dieser, in Anbetracht der Leistungsmaße moderner Rechner, wohl leicht zu verschmerzen. Gerade im Bereich verteilter Systeme, der zunehmend an Bedeutung gewinnt, könnte ein auf *structural typing* basierendes Typsystem, gekoppelt mit einer einfachen, klassenlosen Objektsemantik (ähnlich wie $\mathcal{L}_o^t$), viele der bekannten Probleme lösen. Insbesondere wäre damit ein Grad an Sprachunabhängigkeit erreichbar, der weit jenseits dessen liegt, was heutzutage erreicht wird.

[1] Das ist natürlich keine pauschale Entschuldigung für schlechtes Softwaredesign.

Kommen wir abschließend noch einmal auf die wesentliche Eigenschaft aller betrachteten (getypten) Programmiersprachen zurück: (statische) Typsicherheit. Obwohl diese Eigenschaft zunächst nicht sehr bedeutend erscheint, ist sie eine sehr starke Charakterisierung der Sprache. Sie sichert dem Programmierer zu, dass ein Programm, welches als statisch wohlgetypt eingestuft wird, zur Laufzeit niemals stecken bleibt.

Nun kann man an dieser Stelle argumentieren, dass auch Java-Programme niemals stecken bleiben, obwohl Java selbst nicht statisch typsicher ist. Diese Feststellung ist jedoch in dieser Form nicht ganz richtig. Zwar lässt sich Java auf dieser Basis nicht direkt mit den in diesem Dokument entwickelten Programmiersprachen vergleichen, da für Java keine small step Semantik existiert, aber man kann sich intuitiv den Unterschied leicht klar machen:

Ein wohlgetyptes Java-Programm kann zur Laufzeit in einen Zustand geraten, in dem ein möglicher small step Interpreter stecken bleiben würde. Betrachten wir dazu das folgende berüchtigte Beispiel:

```
public String f() {
  String[] a = new String[1];
  a[0] = "1";
  g(a);
  return a[0];
}

public void g(Object[] a) {
  a[0] = new Integer(1);
}
```

Dieses Programmstück ist, wie man sich leicht überzeugen kann, wohlgetypt in Java. Denn im Typsystem von Java gilt $T[] \leq T'[]$ für $T \leq T'$. Dies führt allerdings dazu, dass die Methode g versucht, ein Integer Objekt in einem String Array abzulegen, so dass nach Rückkehr in die Methode f an Stelle 0 in a kein String mehr steht und auch keine Instanz einer Subklasse von String.

Um diese Art von Problemen zu lösen, wird in Java Laufzeittypüberprüfung durchgeführt, d.h. es wird nicht nur während der Übersetzung in Bytecode eine statische Typüberprüfung durchgeführt, sondern (fast) jede Aktion wird zur Laufzeit noch einmal auf Gültigkeit überprüft. Sollte dabei ein Problem erkannt werden, so löst die Laufzeitumgebung eine Exception aus, welche die Ausführung an der aktuellen Position abbricht, und vom Programmierer an geeigneter Stelle gefangen werden kann.

Die in diesem Dokument betrachteten Programmiersprachen kommen ohne Exceptions[2] und ohne Laufzeittypüberprüfung aus, denn Probleme, die in Sprachen wie Java diese Korrekturmaßnahmen notwendig machen, werden in einer (statisch) typsicheren Programmiersprache bereits zur Compilezeit erkannt. Für den Programmierer bedeutet dies eine teilweise erhebliche Zeitersparnis, da hiermit eine bedeutende Fehlerquelle ausgeschaltet ist.

[2]Dabei ist zu beachten, dass bestimmte Aspekte des Laufzeitverhaltens, wie Division durch Null, nicht durch die statische Typüberprüfung erkannt werden können, und somit zur korrekten Behandlung Exceptions erfordern würden.

Index

W

Literaturverzeichnis

[AC96] Abadi, Martin ; Cardelli, Luca: *A Theory of Objects*. New York : Springer Verlag, 1996. – ISBN 0–387–94775–2

[ASS01] Abelson, Harold ; Sussman, Gerald J. ; Sussman, Julie: *Struktur und Interpretation von Computerprogrammen*. 4. Auflage. Berlin : Springer-Verlag, 2001. – ISBN 3–540–42342–7

[ASU99] Aho, Alfred V. ; Sethi, Ravi ; Ullmann, Jeffrey D.: *Compilerbau Teil 1*. 2. Auflage. München ; Wien : Oldenbourg Verlag, 1999. – ISBN 3–486–25294–1

[Boo91] Booch, Grady: *Object oriented design with applications*. Redwood City, CA, USA : Benjamin-Cummings Publishing Co., Inc., 1991. – ISBN 0–8053–0091–0

[DP88] Dörfler, Willibald ; Peschek, Werner: *Einführung in die Mathematik für Informatiker*. München : Carl Hanser Verlag, 1988. – ISBN 3–446–15112–5

[JP99] Jim, Trevor ; Palsberg, Jens: *Type inference in systems of recursive types with subtyping*. 1999

[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“

[Pau95] Paulson, Lawrence C.: *Foundations of functional programming*. 1995. – Manuskript

[Pau96] Paulson, Lawrence C.: *ML for the working programmer*. 2. Auflage. New York, NY, USA : Cambridge University Press, 1996. – ISBN 0–521–56543–X

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

[PWO97] Palsberg, Jens ; Wand, Mitchell ; O'Keefe, Patrick: Type Inference with Non-Structural Subtyping. In: *Formal Aspects of Computing* 9 (1997), Nr. 1, S. 49–67

[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

[Sta92] Stanisfer, Ryan: *ML Primer*. Englewood Cliffs, NJ, USA : Prentice Hall, 1992. – ISBN 0–13–561721–9

[Str00] Stroustrup, Bjarne: *Die C++ Programmiersprache*. 4. Auflage. München : Addison-Wesley Verlag, 2000. – ISBN 3–827–31660–X

[Tar55] Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. In: *Pacific J. Math.* 5 (1955), S. 285–309

[Ull98] Ullmann, Jeffrey D.: *Elements of ML programming*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1998. – ISBN 0–137–90387–1

[Wag94] Wagner, Klaus W.: *Einführung in die Theoretische Informatik*. Hamburg : Springer-Verlag, 1994. – ISBN 3–540–58139–1

[Weg90] Wegner, Peter: Concepts and paradigms of object-oriented programming. In: *SIGPLAN OOPS Messenger* 1 (1990), Nr. 1, S. 7–87. – ISSN 1–055–6400

[Weg93] Wegener, Ingo: *Theoretische Informatik*. Wiesbaden : 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 26. September 2007 _____

(Benedikt Meurer)