Ein Kalkül für polymorphe Objekte mit Untertypen

Diplomarbeit von Dirk Thierbach

Betreuer: Prof. Dr. K. Keimel
Fachbereich Mathematik

Technische Hochschule Darmstadt
Juli 1997

Inhaltsverzeichnis

Kapitel 1

Einleitung

Language shapes the way we think,
and determines what we can think about.

B. L. Whorf

Objekt-orientiertes Programmieren ist immer noch eines der derzeit aktuellen Schlagwörter der Informatik. Dennoch sind die mathematischen Grundlagen, die diesem Paradigma zu Grunde liegen, noch nicht besonders gut verstanden. In den letzten Jahren gab es viele teilweise sehr erfolgreiche Versuche, dieses Paradigma mit den mathematischen Methoden, die man für das funktionale Programmieren kennt, zu erfassen [GM94, Bru94, PT94, QKB96].

Diese Arbeit versucht, sich dem Problem von einer anderen Seite zu nähren. Um objekt-orientiertes Programmieren zu verstehen, ist es wichtig, die minimalen Grundbegriffe zu identifizieren, auf denen es beruht. Diese Grundbegriffe werden hier in Form eines Kalküls formalisiert. Der Kalkül liegt irgendwo zwischen imperativem und funktionalem Programmieren, da er einerseits noch mit Kopierregeln bei der Reduktion arbeitet, es andererseits aber nicht möglich ist, Funktionen direkt im Kalkül anzugeben. Es handelt sich also nicht einfach um eine Erweiterung des Lambda-Kalküls, sondern um einen eigenständigen Ansatz.

Fast noch wichtiger als einen passenden Kalkül zu finden ist es, für diesen Kalkül ein angemessenes Typsystem zu formulieren. Typen unterstützen den Programmierer, indem sie ihn Fehler früher erkennen lassen. Daher kann man von dem Typsystem fordern, daß es bestimmte Arten von Fehlern auszuschließen helfen soll. Die andere wichtige Anforderung ist natürlich, daß das Typsystem stark genug sein soll, um alle wichtigen Programme auch erfassen zu können.

Als Indikator dafür, wann das Typsystem umfassend genug ist, wurde hier die Darstellbarkeit der Formalismen aus dem funktionalen Programmieren im getypten Objektkalkül benutzt. Dies ist keine triviale Forderung, da der Kalkül eben auf einem völlig anderen Ansatz als der Lambda-Kalkül beruht.

Um das Verständnis zu erleichtern werden wir den Kalkül in mehreren Schritten einführen. Wir beginnen mit einer einfachen Grundversion. Diese wird dann zunächst um den Begriff der Initialisierung erweitert und schließlich noch polymorph gemacht. Danach zeigen wir, daß das Typsystem jetzt umfassend genug ist, um auch den polymorphen Lambda-Kalkül interpretieren zu können. Schließlich fassen wir die wichtigsten Erkenntnisse noch einmal zusammen.

An dieser Stelle möchte ich mich noch recht herzlich bei Herrn Prof.Dr. Keimel für die geduldige Betreuung während bei der Entstehung dieser Arbeit bedanken.

Kapitel 2

Der Grundkalkül

In diesem Kapitel wird zum besseren Verständnis eine vereinfachte Version des Gesamtkalküls eingeführt, der dann in den folgenden Kapiteln erweitert wird. Für den Kalkül wird ein Typsystem in erklärt und einige wichtige Eigenschaften des Kalküls und des Typsystems bewiesen. Der Grundkalkül selbst wird in zwei Schritten definiert, wobei der zweite Schritt zunächst durch das Typsystem motiviert wird.

2.1 Syntax und Reduktionsregeln

Ein Objekt in unserem Kalkül hat zwei verschiedene Arten von Attributen, nämlich *Felder* und *Methoden*. Die Felder entsprechen Daten, die Methoden Programmen (oder Prozeduren). Beide Arten von Attributen werden innerhalb eines Objektes durch *Namen* gekennzeichnet. Auf die Felder eines Objektes kann wie auf die Felder eines Records (Verbundes) zugegriffen werden, das heißt, ein Feld kann gelesen und überschrieben werden. Eine Methode dagegen kann ausgeführt werden und bewirkt damit eine Zustandsänderung des Objektes. Dazu ist es nötig, innerhalb einer Methode auf das gesamte Objekt zugreifen zu können (*Selbst-Referenz*).

Damit sind schon alle grundlegenden Konzepte erklärt, die zur Modellierung benötigt werden. Wir notieren die Terme unseres Kalküls durch die Buchstaben A, B, C, ... und Namen allgemein durch N, M. Konkrete Namen wie move oder new_position werden dagegen klein geschrieben. Zur Unterscheidung zwischen Methoden und Feldern wird Methoden ein kleines σ vorangestellt, das einerseits an das λ für Funktionen aus dem Lambda-Kalkül erinnern soll und andererseits für *state change* steht. Die Selbst-Referenz wird durch $ notiert; dies ist gleichzeitig die einzige Variable in dem Kalkül. Ein Objekt schließlich besteht aus einer Liste L von Attributdefinitionen in eckigen Klammern. Dabei sind diese Attributslisten nur eine Notationsvereinfachung und kein konzeptionell eigenständiger Bestandteil des Kalküls. Das Lesen eines Feldes wird durch $A.N$, das Überschreiben mit einem neuen Wert B durch $A.N \Leftarrow B$ und das Ausführen einer Methode durch $A.\sigma N$ gekennzeichnet. Schließlich führen wir noch ein *Fehlersymbol* $\Diamond$ ein, damit wir später Aussagen über das Reduktionsverhalten getypter Terme machen können.

Als eine weitere Notationsvereinfachung werden wir das mehrfache Auftreten gleichartiger Ausdrücke durch Indizes kennzeichenen und die zugehörige Bereichsangabe meist weglassen. Also steht N_i=A_i zum Beispiel für die Attributsliste N_1=A_1, N_2=$A_2, \ldots, N_k$=A_k für ein geeignetes k. Für Listen schreiben wir auch einfach L. Damit können wir jetzt die (vorläufige) Syntax des Kalküls angeben:

Definition 2.1 Ein Term A des Kalküls wird nach folgender Grammatik gebildet:

$$\begin{array}{lcl} A & ::= & A.N \mid A.N \Leftarrow A' \mid A.\sigma N \mid [L] \mid \$ \mid \Diamond \\ L & ::= & N_i\text{=}A_i, \sigma M_j\text{=}B_j \end{array}$$

Für die Reduktionsregeln benötigen wir die Substitution $A\,[\$:= B]$, die weiter unten definiert wird und jedes Vorkommen von \$ in A außerhalb einer Methode durch B ersetzt. Außerdem muß die Reduktion das Fehlersymbol $\Diamond$ berücksichtigen.

Definition 2.2 Ein *Redex* ist ein Term bestehend aus einem Objekt oder dem Fehlersymbol, jeweils gefolgt von einer Lese-, Schreib- oder Ausführoperation. Es sei L eine Liste, die keine Definition für N beziehungsweise σN enthält. Dann *kontrahiert* ein Redex auf eine der folgenden Arten:

$$\begin{array}{ll} [N\text{=}A, L].N & \to A \\ [N\text{=}A, L].N \Leftarrow B & \to [N\text{=}B, L] \\ [\sigma N\text{=}A, L].\sigma N & \to A\,[\$:= [\sigma N\text{=}A, L]] \end{array}$$

Ist die linke Seite des Redex von der Form $C \equiv [L]$ oder $C \equiv \Diamond$, dann kontrahiert der Redex zu $\Diamond$:

$$\begin{array}{ll} C.N & \to \Diamond \\ C.N \Leftarrow B & \to \Diamond \\ C.\sigma N & \to \Diamond \end{array}$$

Es tritt also eine Fehler auf, sobald ein Objekt das geforderte Attribut nicht enthält, und dieser Fehler pflanzt sich fort. Nachzutragen ist noch die Definition der Substitution.

Definition 2.3 Die Substitution $A\,[\$:= C]$ ist induktiv definiert wie folgt (für die syntaktische Gleichheit schreiben wir $\equiv$):

$$
\begin{array}{ll}
\Diamond\,[\$:= C] & \equiv \Diamond \\
\$\,[\$:= C] & \equiv C \\
(A.N)\,[\$:= C] & \equiv A\,[\$:= C]\,.N \\
(A.N \Leftarrow B)\,[\$:= C] & \equiv A\,[\$:= C]\,.N \Leftarrow B\,[\$:= C] \\
(A.\sigma N)\,[\$:= C] & \equiv A\,[\$:= C]\,.\sigma N \\
[N_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j]\,[\$:= C] & \equiv [N_i\texttt{=}A_i\,[\$:= C], \sigma M_j\texttt{=}B_j]
\end{array}
$$

Es wird also $ in allen Teiltermen ersetzt, außer innerhalb von Methodendefinitionen. In diesem Sinne kann man sagen, daß Methodendefinitionen die Variable $ *binden*.

Schließlich wollen wir noch die Kontraktion eines Redex in jedem Kontext ermöglichen. Dazu erweitern wir die Notation $\to$ zu einer Relation zwischen beliebigen Termen:

Definition 2.4 Ein Term A *reduziert* in einem Schritt zu einem Term A' — in Symbolen $A \to A'$ — wenn A einen Redex B enthält, dieser zu B' kontrahiert und A' aus A entsteht, indem man B in A durch B' ersetzt.

Wir schreiben $\twoheadrightarrow$ für die reflexive und transitive Hülle von $\to$.

Die Kontextregeln kann man auf die übliche Art und Weise auch rekursiv definieren, vergleiche auch Definition 2.12.

2.2 Beispiele

Das Standard-Beispiel, das für solche Kalküle immer wieder verwendet wird, ist der Punkt als graphisches Objekt:

Beispiel 2.5 Ein zweidimensionaler Punkt habe als Zustand seine x- und y-Koordinate und als Methode eine Prozedure move, die ihn um eine Distanz delta — die wieder ein Punkt ist — verschiebt. Dabei kann man sich Zahlenwerte zunächst durch geeignete Terme dargestellt denken, und außerdem gebe es eine Infix-Operation $+$, die Zahlen verknüpft. Dann sei

$$
\begin{array}{ll}
\mathsf{POINT} \equiv & [\mathsf{x=0}, \mathsf{y=0}, \mathsf{delta}\texttt{=}\Diamond, \\
& \;\sigma\mathsf{move}\texttt{=}\$.\mathsf{x} \Leftarrow (\$.\mathsf{x} + \$.\mathsf{delta.x}).\mathsf{y} \Leftarrow (\$.\mathsf{y} + \$.\mathsf{delta.y})]
\end{array}
$$

Als eine Beispielreduktion betrachten wir

$$
\begin{aligned}
&\quad \mathsf{POINT.delta}\lhd\!\!-[\mathsf{x{=}2, y{=}3, z{=}4}].\sigma\mathsf{move} \\
&\to [\mathsf{x{=}0, y{=}0, delta{=}[x{=}2, y{=}3, z{=}4]}, \\
&\quad \sigma\mathsf{move{=}\$.x}\lhd\!\!-(\mathsf{\$.x} + \mathsf{\$.delta.x}).\mathsf{y}\lhd\!\!-(\mathsf{\$.y} + \mathsf{\$.delta.y}) \\
&\quad].\sigma\mathsf{move} \\
&\twoheadrightarrow [\mathsf{x{=}0, y{=}0, delta{=}[x{=}2, y{=}3, z{=}4]}, \\
&\quad \sigma\mathsf{move{=}\$.x}\lhd\!\!-(\mathsf{\$.x} + \mathsf{\$.delta.x}).\mathsf{y}\lhd\!\!-(\mathsf{\$.y} + \mathsf{\$.delta.y}) \\
&\quad].\mathsf{x}\lhd\!\!-(0 + [\mathsf{x{=}2, y{=}3, z{=}4}].\mathsf{x}).\mathsf{y}\lhd\!\!-(0 + [\mathsf{x{=}2, y{=}3, z{=}4}].\mathsf{y}) \\
&\twoheadrightarrow [\mathsf{x{=}0, y{=}0, delta{=}[x{=}2, y{=}3, z{=}4]}, \\
&\quad \sigma\mathsf{move{=}\$.x}\lhd\!\!-(\mathsf{\$.x} + \mathsf{\$.delta.x}).\mathsf{y}\lhd\!\!-(\mathsf{\$.y} + \mathsf{\$.delta.y}) \\
&\quad].\mathsf{x}\lhd\!\!-(0 + 2).\mathsf{y}\lhd\!\!-(0 + 3) \\
&\twoheadrightarrow [\mathsf{x{=}0, y{=}0, delta{=}[x{=}2, y{=}3, z{=}4]}, \\
&\quad \sigma\mathsf{move{=}\$.x}\lhd\!\!-(\mathsf{\$.x} + \mathsf{\$.delta.x}).\mathsf{y}\lhd\!\!-(\mathsf{\$.y} + \mathsf{\$.delta.y}) \\
&\quad].\mathsf{x}\lhd\!\!-(2).\mathsf{y}\lhd\!\!-(3) \\
&\to [\mathsf{x{=}2, y{=}0, delta{=}[x{=}2, y{=}3, z{=}4]}, \\
&\quad \sigma\mathsf{move{=}\$.x}\lhd\!\!-(\mathsf{\$.x} + \mathsf{\$.delta.x}).\mathsf{y}\lhd\!\!-(\mathsf{\$.y} + \mathsf{\$.delta.y}) \\
&\quad].\mathsf{y}\lhd\!\!-(3) \\
&\to [\mathsf{x{=}2, y{=}3, delta{=}[x{=}2, y{=}3, z{=}4]}, \\
&\quad \sigma\mathsf{move{=}\$.x}\lhd\!\!-(\mathsf{\$.x} + \mathsf{\$.delta.x}).\mathsf{y}\lhd\!\!-(\mathsf{\$.y} + \mathsf{\$.delta.y})]
\end{aligned}
$$

Man beachte, daß der Parameter delta als ein zusätzliches Feld modelliert wurde. Dafür wollen wir eine eigene Notation einführen: Eingabeparameter von, sagen wir, der Methode m werden mit einem hochgestellten Minus m^- gekennzeichnet, Ausgabewerte mit einem hochgestellten Plus m^+. Die Exponenten sind dabei zunächst als Bestandteil des Namens zu betrachen.

Zu erklären ist auch noch, wie der Infix-Operator + modelliert wird. Man kann dies zum Beispiel wie in `Smalltalk` erreichen, indem die Objekte für Zahlen mit einer Methode add zusammen mit einem Eingabeparameter add^+ versehen werden. Etwas weiter unten werden wir einen konkreten Term in dieser Art angeben. Damit wir auch einstellige Prozeduren demonstrieren können, nehmen wir außerdem eine Methode abs hinzu, die den Absolutbetrag einer Zahl berechnet. Damit können wir das Beispiel von eben vollständig im Kalkül notieren:

Beispiel 2.6 Der Punkt aus 2.5, um eine Methode norm für die 1-Norm mit Ausgabeparameter erweitert, sieht vollständig so aus:

$$
\begin{array}{ll}
\mathsf{POINT} \equiv [\mathsf{x{=}0}, \mathsf{y{=}0}, & \\
\quad \mathsf{move}^{-}\mathsf{=}\lozenge, & \\
\quad \sigma\mathsf{move{=}}\ \mathsf{\$.x}\Leftarrow((\mathsf{\$.x}).\mathsf{add}^{-}\Leftarrow(\mathsf{\$.delta.x}).\sigma\mathsf{add}) & \\
\quad\quad \mathsf{.y}\Leftarrow((\mathsf{\$.y}).\mathsf{add}^{-}\Leftarrow(\mathsf{\$.delta.y}).\sigma\mathsf{add}), & \\
\quad \mathsf{norm}^{+}\mathsf{=0}, & \\
\quad \sigma\mathsf{norm{=}\$.norm}^{+}\Leftarrow((\mathsf{\$.x.}\sigma\mathsf{abs.abs}^{+}).\mathsf{add}^{-}\Leftarrow(\mathsf{\$.y.}\sigma\mathsf{abs.abs}^{+}).\sigma\mathsf{add}) & \\
] &
\end{array}
$$

Man sieht, daß die so im Kalkül notierten Funktionen nicht sonderlich gut lesbar sind. Später werden wir Abkürzungen für Funktionsaufrufe mit Parametern einführen, nachdem allgemein untersucht wurde, wie Funktionen in das Gesamtbild hineinpassen.

Als weiteres Beispiel demonstrieren wir, wie sich zwei grundlegende Datentypen in dem Kalkül darstellen lassen:

Beispiel 2.7 Der Datentyp Boolean wird zunächst durch zwei Objekte FALSE und TRUE modelliert, die auf eine Nachricht $\sigma\mathsf{if}$ hin zwischen den zwei Argumenten then und else auswählt:

$$
\begin{array}{l}
\mathsf{TRUE} \equiv [\mathsf{then{=}}\lozenge, \mathsf{else{=}}\lozenge, \mathsf{if}^{+}\mathsf{=}\lozenge, \sigma\mathsf{if{=}\$.if}^{+}\Leftarrow(\mathsf{\$.then})] \\
\mathsf{FALSE} \equiv [\mathsf{then{=}}\lozenge, \mathsf{else{=}}\lozenge, \mathsf{if}^{+}\mathsf{=}\lozenge, \sigma\mathsf{if{=}\$.if}^{+}\Leftarrow(\mathsf{\$.else})]
\end{array}
$$

Beispiel 2.8 Die natürlichen Zahlen kann man aus einem Objekt ZERO konstruieren, das die Nachfolgeoperation succ, die Vorgängerabfrage pred^{+} und den Test iszero^{+} versteht:

$$
\begin{array}{l}
\mathsf{ZERO} \equiv [\mathsf{iszero}^{+}\mathsf{=TRUE}, \\
\quad \sigma\mathsf{succ{=}}(\mathsf{\$.iszero}^{+}\Leftarrow(\mathsf{FALSE}).\mathsf{pred}^{+}\Leftarrow\mathsf{\$}), \\
\quad \mathsf{pred}^{+}\mathsf{=}\lozenge]
\end{array}
$$

Mit diesen drei Attributen lassen sich sehr leicht die Methoden add, sub und mul definieren:

$$
\begin{array}{l}
\mathsf{ZERO} \equiv [\ldots, \\
\quad \mathsf{add}^{-}\mathsf{=}\lozenge, \sigma\mathsf{add{=}}(\mathsf{\$.add}^{-}).\mathsf{iszero}^{+}.\mathsf{then}\Leftarrow(\mathsf{\$}) \\
\quad\quad \mathsf{.else}\Leftarrow(\mathsf{\$.add}^{-}\Leftarrow(\mathsf{\$.add}^{-}.\mathsf{pred}^{+}).\sigma\mathsf{add}.\sigma\mathsf{succ}).\sigma\mathsf{if.if}^{+} \\
\quad \mathsf{sub}^{-}\mathsf{=}\lozenge, \sigma\mathsf{sub{=}}(\mathsf{\$.add}^{-}).\mathsf{iszero}^{+}.\mathsf{then}\Leftarrow(\mathsf{\$}) \\
\quad\quad \mathsf{.else}\Leftarrow(\mathsf{\$.add}^{-}\Leftarrow(\mathsf{\$.add}^{-}.\mathsf{pred}^{+}).\sigma\mathsf{sub}.\sigma\mathsf{pred}).\sigma\mathsf{if.if}^{+} \\
\quad \mathsf{mul}^{-}\mathsf{=}\lozenge, \sigma\mathsf{mul{=}}(\mathsf{\$.add}^{-}).\mathsf{iszero}^{+}.\mathsf{then}\Leftarrow(\mathsf{\$.add}^{-}) \\
\quad\quad \mathsf{.else}\Leftarrow(\mathsf{\$.add}^{-}(\mathsf{\$.mul}^{-}\Leftarrow(\mathsf{\$.mul}^{-}.\mathsf{pred}^{+})).\sigma\mathsf{add}).\sigma\mathsf{if.if}^{+}]
\end{array}
$$

Man beachte, daß für die Rekursion keine besondere Vorkehrung nötig ist.

Nun versuchen wir, auch für den Datentyp Boolean Methoden zu definieren. Dabei ergibt sich zunächst eine Schwierigkeit: Die Methoden in TRUE müssen unter Umständen den Term FALSE benutzen, und umgekehrt. Es tritt also *gegenseitige* Rekursion auf. Dieses Problem kann man lösen, indem man beide Terme auch verfügbar macht. Wir benennen also TRUE und FALSE in TRUE′ und FALSE′ um und definieren neu:

Beispiel 2.9 Der Datentyp Boolean mit den Methoden and, or und not läßt sich modellieren als

$$
\begin{array}{rl}
\mathsf{TRUE} \equiv & [\mathsf{self}{=}\mathsf{TRUE}', \mathsf{other}{=}\mathsf{FALSE}', \mathsf{then}{=}\Diamond, \mathsf{else}{=}\Diamond, \\
& \quad \mathsf{if}^{+}{=}\Diamond, \sigma\mathsf{if}{=}\$.\mathsf{if}^{+}\Leftarrow(\$.\mathsf{self}.\mathsf{then}\Leftarrow(\$.\mathsf{then}).\mathsf{else}\Leftarrow(\$.\mathsf{else}).\sigma\mathsf{if}.\mathsf{if}^{+}) \\
& \quad \mathsf{and}^{-}{=}\Diamond, \mathsf{and}^{+}{=}\Diamond, \sigma\mathsf{and}{=} \\
& \qquad \$.\mathsf{and}^{+}\Leftarrow(\$.\mathsf{self}.\mathsf{then}\Leftarrow(\$.\mathsf{and}^{-}).\mathsf{else}\Leftarrow(\$)) \\
& \quad \mathsf{or}^{-}{=}\Diamond, \mathsf{or}^{+}{=}\Diamond, \sigma\mathsf{or}{=} \\
& \qquad \$.\mathsf{or}^{+}\Leftarrow(\$.\mathsf{self}.\mathsf{then}\Leftarrow(\$).\mathsf{else}\Leftarrow(\$.\mathsf{or}^{-})) \\
& \quad \mathsf{not}^{+}{=}\Diamond, \sigma\mathsf{not}{=}\$.\mathsf{not}^{+}\Leftarrow(\$.\mathsf{self}\Leftarrow(\$.\mathsf{other}).\mathsf{other}\Leftarrow(\$.\mathsf{self})) \\
&]
\end{array}
$$

und FALSE genauso, nur daß self=FALSE′ und other=TRUE′ gesetzt wird.

In [MMS95] wurde das Problem der gegenseitigen Referenz ebenfalls erkannt, aber auf eine etwas andere Weise gelöst.

Verallgemeinert man diese Beispiele, so sieht man, daß man beliebige induktive Datentypen darstellen kann. Dies ist auch nicht weiter verwunderlich, da wir in einem späteren Kapitel aufzeigen werden, wie der komplette Lambda-Kalkül simuliert werden kann.

2.3 Diskussion der Modellierung

Die Notation der Syntax orientiert sich an dem in [AC94, AC96] entwickelten ς-Kalkül. Auf den ersten Blick ist der Unterschied zwischen beiden Kalkülen nicht allzu groß. Der Hauptunterschied ist, daß die Variable für die Selbstreferenz bei Abadi und Cardelli benannt ist, während hier nur eine einzige Variable $ benötigt wird. Tatsächlich wird in [AC96] sogar behauptet, daß verschiedene Namen nötig sind, um zum Beispiel die natürlichen Zahlen darstellen zu können. Wie man aber an den obigen Beispielen sieht, ist das nicht der Fall. Das liegt daran, daß im ς-Kalkül nicht explizit zwischen Feldern und Methoden unterschieden wird. Ist dieser Unterschied in Hinblick auf das Binden von $ gegeben, so sind die verschiedenen Namen des ς-Kalküls nicht mehr nötig.

Dieser Effekt ist auch symptomatisch für den Unterschied im Modellierungsansatz zwischen beiden Kalkülen. Bei Abadi und Cardelli wird immer noch funktional gedacht: Eine Methode ist im Prinzip eine Funktion mit einem ausgezeichneten Parameter. Sie gehen später sogar so weit, daß sie die übliche λ-Abstraktion — nachdem sie sie im ungetypten Kalkül kodiert haben — als primitive Konstruktion ansehen und eigene Typregeln dafür angeben. Leider stimmen diese Regeln nicht mit denen überein, die sich aus der Kodierung ergeben würden. In diesem Sinn ist also der getypte ς-Kalkül eigentlich eine Erweiterung des getypten Lambda-Kalküls. Man merkt auch noch an anderen Stellen, daß Abadi und Cardelli eher funktional modellieren, so zum Beispiel, wenn sie Klassen als eine Funktion definieren, die aus einer Sammlung von Funktionen (*trait*) ein Objekt macht.

Will man aber das Wesentliche des objekt-orientierten Paradigmas erfassen, dann darf man nicht von Funktionen ausgehen. Fundamental für Objekte ist ihr Zustand und die Zustandsänderungen. Und genau diese beiden, und nur diese beiden, werden hier modelliert. Es ist auch wesentlich für Objekte, daß sie geschützt sind: Man kann ihre Methoden nicht verändern und auch in ihre Felder nicht direkt hineingreifen. Genau dies wäre aber möglich, wenn man freie Variablen für die Selbstreferenz zuläßt.

Es ist typisch für imperatives Programmieren, daß Programme und Daten getrennt sind, und für funktionales Programmieren, daß zwischen ihnen nicht unterschieden wird. Das hier gewählt Modell liegt in der Mitte: Sie werden zwar konzeptionell getrennt, sind aber beide in einem Objekt vereint, das dann als Datenwert auch Programme enthalten kann.

Schließlich ist noch anzumerken, daß die oben geschilderten Beispiele von Datentypen eher untypisch sind. Die meisten dieser Beispiele sind induktive Datentypen, die oftmals binäre (also eigentlich keine zustandsändernde) Operationen haben. Auch ist die hier gewählte Modellierung nicht unbedingt die beste Variante. Sie zeigt nur, daß der Kalkül ausdrucksstark genug ist, und illustriert außerdem, wie das zustandsbasierte Programmieren in diesem Kalkül vor sich geht.

2.4 Die Church-Rosser Eigenschaft

Eine wichtige Eigenschaft, die jeder Kalkül zur Modellierung einer Programmiersprache erfüllen sollte, ist die sogenannte *Church-Rosser-Eigenschaft*. Hat der Kalkül diese Eigenschaft, dann kommt es (anschaulich gesprochen) auf die Reduktionsreihenfolge nicht an. Es sind also verschiedene Auswertungsstrategien möglich.

Es ist an dieser Stelle sinnvoll, sich noch nicht auf eine Auswertungsstrategie festzulegen, da sich beim Lambda-Kalkül gezeigt hat, daß bei den darauf basierenden funktionalen Sprachen dann in der Praxis auch verschiedene Strategien verwendet werden, jede mit ihren Vor- und Nachteilen.

Wir zeigen zunächst einige Hilfssätze, bevor wir die Church-Rosser-Eigenschaft formulieren und beweisen. Dabei gehen wir analog zu der in [Tak92] beschriebenen Tait-Methode für den Lambda-Kalkül vor.

Hilfssatz 2.10 Es gilt $D\,[\$:= C\,[\$:= B]\,] \equiv D\,[\$:= C]\,[\$:= B]$.

Beweis: Eine einfache Induktion über den Termaufbau von D. □

Lemma 2.11 (Substitutionslemma) Gilt $A \to A'$, dann folgt

$$A\,[\$:= B] \twoheadrightarrow A'\,[\$:= B] \qquad .$$

Beweis: Wieder eine einfache Induktion über den Termaufbau von A. Reduziert $A \to A'$ nach den Kontextregeln, dann ist der Induktionsschritt offensichtlich. Auch bei der Reduktion durch Kontraktion eines Redex ist — außer beim Ausführen einer Methode — alles klar. Beim Ausführen einer Methode ist A von der Form $A \equiv C.\sigma N$ mit $C \equiv [\sigma N\texttt{=}D, L]$, und somit $A' \equiv D\,[\$:= C]$. Aber dann ist

$$\begin{aligned} A\,[\$:= B] &\equiv [\sigma N\texttt{=}D, L].\sigma N\,[\$:= B] \equiv [\sigma N\texttt{=}D, L\,[\$:= B]\,].\sigma N \\ &\to D\,[\$:= C\,[\$:= B]\,] \equiv D\,[\$:= C]\,[\$:= B] \equiv A'\,[\$:= B] \end{aligned}$$

nach dem vorigen Hilfssatz und mittels Induktion. □

Die Schlüsselidee für den nachfolgenden Beweis ist der Begriff der *parallelen Reduktion*. Intuitiv gesehen reduziert ein Term A parallel zu einem Term A', wenn mehrere der in A vorhandenen Redexe (die auch verschachtelt sein dürfen) gleichzeitig kontrahiert werden können, aber nicht diejenigen Redexe, die bei diesem Prozeß neu entstehen.

Definition 2.12 (Parallele Reduktion) Die Relation $\Rightarrow$ zwischen Termen wird induktiv definiert wir folgt: Gilt $A \Rightarrow A'$, $A_i \Rightarrow A_i'$ und $B \Rightarrow B'$, und ist $L \equiv N_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j$ sowie $L' \equiv N_i\texttt{=}A_i', \sigma M_j\texttt{=}B_j$ (wobei alle N_i und M_j von N verschieden sind), dann gilt auch

(1)	$\Diamond$	$\Rightarrow \Diamond$
(2)	$\$$	$\Rightarrow \$$
(3.1)	$[N\texttt{=}A, L].N$	$\Rightarrow A'$
(3.2)	$[L].N$	$\Rightarrow \Diamond$
(3.3)	$\Diamond.N$	$\Rightarrow \Diamond$
(3.4)	$A.N$	$\Rightarrow A'.N$

$$
\begin{array}{lll}
(4.1) & [N{=}A, L].N{\twoheadleftarrow}B & \Rightarrow [N{=}B', L] \\
(4.2) & [L].N{\twoheadleftarrow}B & \Rightarrow \lozenge \\
(4.3) & \lozenge.N{\twoheadleftarrow}B & \Rightarrow \lozenge \\
(4.4) & A.N{\twoheadleftarrow}B & \Rightarrow A'.N{\twoheadleftarrow}B' \\
(5.1) & [\sigma N{=}A, L].\sigma N] & \Rightarrow A'\,[\$:= [\sigma N{=}A', L']] \\
(5.2) & [L].\sigma N & \Rightarrow \lozenge \\
(5.3) & \lozenge.\sigma N & \Rightarrow \lozenge \\
(5.4) & A.\sigma N & \Rightarrow A'.\sigma N \\
(6) & [L] & \Rightarrow [L']
\end{array}
$$

Diese Definition ist leider etwas aufwendiger, da der Kalkül doch schon recht kompliziert ist und hier alle der in dem Kalkül vorhandenen Ausdrucksmittel auf einmal explizit gemacht werden müssen. In dieser Definition stecken auch die oben erwähnten Kontextregeln für die Reduktion.

Man beachte, daß die Relation $\Rightarrow$ schon reflexiv ist.

Lemma 2.13 Für die paralle Reduktion $\Rightarrow$ gelten folgende Aussagen:

(i) Aus $A \rightarrow A'$ folgt $A \Rightarrow A'$.

(ii) Aus $A \Rightarrow A'$ folgt $A \twoheadrightarrow A'$.

(iii) Gilt $A \Rightarrow A'$, dann auch $A\,[\$:= B] \Rightarrow A'\,[\$:= B]$.

(iv) Gilt $B \Rightarrow B'$, dann auch $A\,[\$:= B] \Rightarrow A\,[\$:= B']$.

Beweis: Alle vier Aussagen zeigt man durch Induktion über den Termaufbau von A. Aussage (i) ist klar, da die Definition von $\Rightarrow$ alle Kontextregeln und alle Kontraktionsregeln von $\rightarrow$ enthält. Für Aussage (ii) muß man die Reduktionen, die parallel ausgeführt werden, hintereinander ausführen. Dazu muß man in den Fällen (3.1), (4.1) und (5.1) die Induktionsannahme und die Kontextregeln für $\rightarrow$ verwenden; alle anderen Fälle sind offensichtlich. Aussage (iii) zeigt man genau wie das Substitutionslemma 2.11, nur gibt es hier mehr triviale Fälle. Für den Methodenaufruf (5.1) kann man wieder Hilfssatz 2.10 verwenden. Aussage (iv) ist offensichtlich: Man muß jeweils nur gemäß der Definition der Substitution und der parallelen Reduktion umformen. $\square$

Eine Konsequenz dieses Lemmas ist, daß $\Rightarrow$ (gesehen als Paarmenge) zwischen $\rightarrow$ und $\twoheadrightarrow$ liegt. Insbesondere ist $\twoheadrightarrow$ die transitive Hülle von $\Rightarrow$.

Um nun die Church-Rosser-Eigenschaft nachzuweisen, genügt es, die sogenannte *diamond-property* (Karo-Eigenschaft) von $\Rightarrow$ zu beweisen:

($\diamond$) Aus $B_1 \Leftarrow A \Rightarrow B_2$ folgt $B_1 \Rightarrow C \Leftarrow B_2$ für ein C

Durch Zerlegen von $\twoheadrightarrow$ in $\Rightarrow$-Schritte und Aneinanderkleben der Karos folgt dann, daß ($\Diamond$) auch für $\twoheadrightarrow$ gilt, was genau die Aussage des Satzes von Church-Rosser ist.

Es ist allerdings einfacher, eine noch stärkere Aussage als ($\Diamond$) zu zeigen:

($*$) Aus $A \Rightarrow B$ folgt $B \Rightarrow A^*$.

Dabei wird A^* durch A bestimmt, ist aber unabhängig von B ist. Genauer gesagt bekommt man A^*, indem man alle in A vorhandenen Redexe gleichzeitig kontrahiert:

(1^*)	$\Diamond^*$	$\equiv \Diamond$
(2^*)	$\$^*$	$\equiv \$$
(3^*.1)	$([N{=}A, L].N)^*$	$\equiv A^*$
(3^*.2)	$([L].N)^*$	$\equiv \Diamond$
(3^*.3)	$(\Diamond.N)^*$	$\equiv \Diamond$
(3^*.4)	$(A.N)^*$	$\equiv A^*.N$ falls keiner der anderen Fälle zutrifft
(4^*.1)	$([N{=}A, L].N{\leftarrow}B)^*$	$\equiv [N{=}B^*, L]$
(4^*.2)	$([L].N{\leftarrow}B)^*$	$\equiv \Diamond$
(4^*.3)	$(\Diamond.N{\leftarrow}B)^*$	$\equiv \Diamond$
(4^*.4)	$(A.N{\leftarrow}B)^*$	$\equiv A^*.N{\leftarrow}B^*$ falls keiner der anderen Fälle zutrifft
(5^*.1)	$([\sigma N{=}A, L].\sigma N])^*$	$\equiv A^*[\$:= [\sigma N{=}A, L]^*]$
(5^*.2)	$([L].\sigma N)^*$	$\equiv \Diamond$
(5^*.3)	$(\Diamond.\sigma N)^*$	$\equiv \Diamond$
(5^*.4)	$(A.\sigma N)^*$	$\equiv A^*.\sigma N$ falls keiner der anderen Fälle zutrifft
(6)	$[N_i{=}A_i, \sigma M_j{=}B_j]^*$	$\equiv [N_i{=}A_i^*, \sigma M_j{=}B_j]$

Satz 2.14 Die Relation $\rightarrow$ erfüllt ($*$) und hat damit die Church-Rosser-Eigenschaft.

Beweis: Die Idee für den Beweis ist die folgende: Reduziert A zu B, indem irgendwo tief innen in A ein Redex kontrahiert wird, dann ist zu zeigen, daß sich alle Kontraktion weiter außen, die bei der Bildung von A^* erforderlich sind, nachholen lassen. Die Aussagen (iii) und (iv) von Lemma 2.13 stellen dabei sicher, daß sich Substitution und Reduktion gegenseitig nicht stören.

Formal läuft der Beweis mittels einer Induktion über den Termaufbau von A, wobei noch nach der Regel unterschieden wird, wegen der $A \Rightarrow B$ gilt:

Fall 1 Gilt $A \equiv \Diamond \Rightarrow B$, dann muß auch $B \equiv \Diamond$ sein, da nur Fall (1) der Reduktion zutreffen kann. Also gilt auch $B \Rightarrow \Diamond \equiv A^*$.

Fall 2 Für $A \equiv \$$ analog.

Fall 3.1 Ist $A \equiv [N{=}C, L].N$ und außerdem $B \equiv C'$ nach (3.1), dann gilt auch $C \Rightarrow C'$. Da C ein Teilterm von A ist, gilt nach Induktionsvoraussetzung $C' \Rightarrow C^*$. Das heißt aber $B \equiv C' \Rightarrow C^* \equiv A^*$.

Fall 3.2 Ist $A \equiv [L].N$ (wobei N in L nicht definiert wird), dann muß $B \equiv \Diamond$ nach (3.2) sein. Das bedeutet $B \equiv A^*$, also auch $B \Rightarrow A^*$.

Fall 3.3 Analog.

Fall 3.4 Direkt durch Induktion.

Fall 4 Genauso wie Fall 3.

Fall 5 Bis auf (5.1) geht wieder alles wie vorher in den Fällen 3 und 4. Der Fall (5.1) ist auch am interessantesten, da hier die eigentliche Arbeit geleistet wird. Zur Abkürzung setzen wir $D{=}[\sigma N{=}C, L]$. Ist nun aber $A \equiv [D.\sigma N]$ und $B \equiv C'\,[\$:= D']$ mit $C \Rightarrow C'$ und $D \Rightarrow D'$, dann gilt nach Induktionsvoraussetzung $C' \Rightarrow C^*$ sowie $D' \Rightarrow D^*$. Aus 2.13 folgt weiter $B \Rightarrow C^*\,[\$:= D'] \Rightarrow C^*\,[\$:= D^*] \equiv A^*$ wie gewünscht.

Fall 6 Wieder direkt durch Induktion. $\square$

2.5 Das Typsystem

Ein Typsystem für unseren Objekt-Kalkül muß im wesentlichen darüber Buch führen, welche Attribute auf welcher Stufe zur Verfügung stehen, genau wie Typsysteme für den Lambda-Kalkül angeben, auf welche Art von Argumenten man einen Term anwenden kann. Das ist aber noch nicht alles: Ein wichtige Eigenschaft des objekt-orientierten Ansatzes ist es, auch mit Objekten umgehen zu können, die erst später zum Beispiel durch Vererbung zustande kommen und somit zum Zeitpunkt der Erstellung eines bestimmten Objektes noch gar nicht bekannt sind. Dies wird formal durch eine Untertypen-Beziehung beschrieben: Ein Typ T *umfaßt* einen Typen S, in Symbolen $T \leq S$, wenn der Typ T alle Attribute (und eventuell sogar mehr) bereitstellt, die auch der Typ S enthält. Um die später entstandenen Objekte dann auch einsetzen zu können, soll es immer möglich sein, den Typ eines Terms *abzuschwächen*, also auf einen weniger umfassenden Typ einzuschränken.

Damit das auch bei Typen höherer Ordnung funktioniert, muß man zwischen *kovarianten* und *kontravarianten* Feldern unterscheiden. Kontravariante Felder (Notation N^-) dienen nur zur Eingabe, kovariante Felder (Notation N^+) dienen nur zur Ausgabe (man erinnere sich an die obigen Beispiele). Die alte Notation N wird beibehalten und besagt, daß es keine Rolle spielt, ob das Feld ko- oder kontravariant ist. Allerdings muß man noch berücksichtigen, daß man *innerhalb* von Methoden Eingabefelder auch lesen und Ausgabefelder schreiben muß. Diese Verwendung müssen wir besonders kennzeichnen. Dazu erweitern wir die Syntax unseres Kalküls:

Definition 2.15 Ein Term A wird gebildet wie in 2.1, wobei zusätzlich

$$A \quad ::= \quad \dots \mid A.{\downarrow}N \mid A.{\downarrow}N \Leftarrow A'$$

Die Reduktionsregeln für $A.{\downarrow}N$ sind genau dieselben wie für $A.N$, und die Regeln für $A.{\downarrow}N \Leftarrow B$ sind dieselben wie für $A.N \Leftarrow B$. Die Substitutionsregeln werden erweitert um

$$\begin{aligned} (A.{\downarrow}N)\,[\$:= C] &\equiv A\,[\$:= C]\,.N \\ (A.{\downarrow}N \Leftarrow B)\,[\$:= C] &\equiv A\,[\$:= C]\,.N \Leftarrow B\,[\$:= C] \end{aligned}$$

Bei der Substitution fällt also die Zusatzinformation, daß es sich im ein eigenes Feld handeln soll, weg.

Nun können wir auch die korrekte Benutzung von Feldern formalisieren, da wir jetzt zwischen dem Zugriff auf eigene Felder und dem Zugriff auf Felder von außen unterscheiden können:

Definition 2.16 Ein Term heißt *varianzkonsistent* oder kurz konsistent, falls die Felder richtig verwendet werden, also keine Teilterme der Art $A.N^-$ oder $A.N^+ \Leftarrow B$ vorkommen.

Man beachte, daß all diese neuen Notationen eigentlich nur Zusatzinformation für die Typisierung liefern und die Reduktion nicht beeinflussen.

Wir wollen jetzt Typen formal einführen. Wir verwenden für sie die Buchstaben T, S und R. Typen können auch Typvariablen enthalten, für die wir kleine griechische Buchstaben benutzen:

Definition 2.17 Ein *Typ* T ist entweder eine Typvariable $\alpha, \beta, \ldots$ oder ein Ausdruck $\{N_i : T_i, \sigma M_j\}$. Damit ist gemeint, daß das zugehörige Objekt die Methoden M_j und die Felder N_i zur Verfügung stellt, wobei die Felder jeweils den Typ T_i haben (wie schon oben lassen wir die Indexbereiche weg).

Wir schreiben $N \in S$ beziehungsweise $\sigma M \in S$, falls das Feld N beziehungsweise die Methode M in S vorkommt. Genauso schreiben wir $S.N = T$, falls das Feld N in S vorkommt und den Typ T hat. Als Abkürzung für eine Liste von Attributtypen verwenden wir den Buchstaben F, ähnlich wie L bei Objekten.

Folgende Aussagen sind also wahr:

$$N \in \{N : T, F\} \qquad \sigma M \in \{\sigma M, F\} \qquad \{N : T, F\}.N = T$$

Typaussagen werden immer relativ zu einem Typkontext Γ gemacht, der Typvariablen mit Typen in Verbindung setzt:

Definition 2.18 Ein *Typkontext* oder kurz Kontext ist eine geordnete Liste

$$\Gamma = \alpha_1 : T_1, \ldots, \alpha_n : T_n$$

Wir schreiben auch $\Box$ für den leeren Kontext.

Nun müssen wir noch sagen, was die freien Typvariablen eines Typs und eines Kontextes sind:

Definition 2.19 Die *freien Variablen* $\mathrm{FV}(T)$ eines Typs T sind alle Typvariablen, die in T vorkommen. Die freien Variablen $\mathrm{FV}(\Gamma)$ eines Kontextes Γ sind alle Variablen, die in dem Kontext deklariert werden. Formal:

$$\mathrm{FV}(\Box) = \{\} \qquad \mathrm{FV}(\alpha : T, \Gamma) = \{\alpha\} \cup \mathrm{FV}(\Gamma)$$

Die Kontexte sind deshalb geordnet, weil später deklarierte Typvariablen als freie Variablen nur die weiter vorne deklarierten enthalten sollen:

Definition 2.20 Ein Kontext Γ heißt *wohlgeformt*, wenn er die folgenden beiden Bedingungen erfüllt: Erstens dürfen keine Variablen doppelt deklariert werden. Zweites soll er entweder leer sein oder von der Form $\Gamma = \Gamma', \alpha : T$, wobei Γ' auch wohlgeformt und die Bedingung $\mathrm{FV}(T) \subseteq \mathrm{FV}(\Gamma')$ erfüllt ist. Ab sofort werden wir nur noch wohlgeformte Kontexte betrachten.

Diese Definition rechtfertigt auch die Begriffsbildung *freie Variablen eines Kontextes*. Darunter könnte man sich zunächst auch die Vereinigung aller freien Variablen der dort deklarierten Typen vorstellen. Bei wohlgeformten Kontexten Γ umfaßt aber $\mathrm{FV}(\Gamma)$ diese Vereinigung.

type var

$$\frac{}{\Gamma, \alpha : T, \Gamma' \vdash \alpha \ \mathsf{type}}$$

type obj

$$\frac{\Gamma \vdash T_i \ \mathsf{type}}{\Gamma \vdash \{N_i : T_i, \sigma M_j\} \ \mathsf{type}}$$

Abbildung 2.1: Der Grundkalkül: Typwohlgeformtheit

sub refl

$$\frac{\Gamma \vdash S \ \mathsf{type}}{\Gamma \vdash S \leq S}$$

sub trans

$$\frac{\Gamma \vdash S \leq R \qquad \Gamma \vdash R \leq T}{\Gamma \vdash S \leq T}$$

sub var

$$\frac{}{\Gamma, \alpha : T, \Gamma' \vdash \alpha \leq T}$$

sub obj

$$\frac{\Gamma \vdash S_i' \leq S_i \qquad \Gamma \vdash T_j \leq T_j'}{\begin{array}{l}\Gamma \vdash \{N_i^- : S_i, N_j^+ : T_j, \sigma M_k\} \\ \quad \leq \{N_i^- : S_i', N_j^+ : T_j', \sigma M_k\}\end{array}}$$

sub ext

$$\frac{\Gamma \vdash \{F'\} \ \mathsf{type}}{\Gamma \vdash \{F, F'\} \leq \{F\}}$$

Abbildung 2.2: Der Grundkalkül: Untertypen

Bemerkung 2.21 Wir wollen später die Aussage $\Gamma \vdash A : T$ so verstehen, daß der Term A im Kontext Γ getypt werden kann und *mindestens* den Typ T umfaßt, aber unter Umständen sogar einen stärkeren Typ $S \geq T$. Genauso ist auch die Deklaration $\alpha : T$ zu verstehen: Die Typvariable α steht für Typen, die *mindestens* den Typ T umfassen.

Man hätte dafür auch die Notation $\alpha \leq T$ wählen können, aber im Hinblick auf spätere Erweiterungen des Kalküls erschien diese Notation passender.

Definition 2.22 Es gibt drei Arten von Regeln, die bei der Typisierung eine Rolle spielen. Wir formalisieren sie als Urteile.

- Ein Typ T heißt *wohlgeformt* im Kontext Γ, wenn in T nur Variablen vorkommen, die in Γ auch erklärt werden. Das schreiben wir $\Gamma \vdash T$ type und formalisieren es durch die beiden Regeln in Abbildung 2.1.

- Die oben erwähnte Untertyprelation wird ausgedrückt durch das Urteil $\Gamma \vdash S \leq T$. Der Typ S umfaßt oder subsumiert also im Kontext Γ den Typ T, wenn dies durch die Regeln in Abbildung 2.2 hergeleitet werden kann. Man beachte, daß die Relation explizit reflexiv und transitiv gemacht wird.

- Einem Term A kann im Kontext Γ der Typ T zugewiesen werden, wenn $\Gamma \vdash A : T$ nach den Regeln in Abbildung 2.3 bewiesen werden kann. Aus

`val subs`

$$\frac{\Gamma \vdash A : S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash A : T}$$

`val self`

$$\frac{}{\Gamma, \alpha : S \vdash \$: \alpha}$$

`val read`

$$\frac{\Gamma \vdash A : S \qquad S.N = T}{\Gamma \vdash A.N : T}$$

`val read'`

$$\frac{\Gamma, \alpha : S \vdash A : \alpha \qquad S.N = T}{\Gamma, \alpha : S \vdash A.{\downarrow}N : T}$$

`val write`

$$\frac{\Gamma \vdash B : T \qquad \Gamma \vdash A : S \qquad S.N = T}{\Gamma \vdash A.N \mathrel{\triangleleft\!\!-} B : S}$$

`val write'`

$$\frac{\Gamma, \alpha : S \vdash B : T \qquad \Gamma, \alpha : S \vdash A : \alpha \qquad S.N = T}{\Gamma, \alpha : S \vdash A.{\downarrow}N \mathrel{\triangleleft\!\!-} B : \alpha}$$

`val exec`

$$\frac{\Gamma \vdash A : S \qquad \sigma M \in S}{\Gamma \vdash A.\sigma M : S}$$

`val exec'`

$$\frac{\Gamma, \alpha : S \vdash A : \alpha \qquad \sigma M \in S}{\Gamma, \alpha : S \vdash A.\sigma M : \alpha}$$

`val obj`

$$\frac{\Gamma, \alpha : S \vdash B_j : \alpha \qquad \alpha \notin \mathrm{FV}(\Gamma) \qquad \Gamma \vdash A_i : T_i \qquad S = \{N_i : T_i, \sigma M_j\}}{\Gamma \vdash [N_i \texttt{=} A_i, \sigma M_j \texttt{=} B_j] : S}$$

Abbildung 2.3: Der Grundkalkül: Termtypen der `val subs`-Regel ergibt sich sofort, daß jeder Term auch mit einem abgeschwächten Typ versehen werden kann. Die Aussage $\Gamma \vdash A : T$ bedeutet also nicht, daß T der einzige Typ für A ist, sondern daß es möglicherweise auch noch stärkere geben kann.

Die Namen für die Regeln orientieren sich an denen in [AC96].

Man beachte den symmetrischen Aufbau für die gestrichenen und ungestrichenen *read*, *write* und *exec*-Regeln.

Damit ist das Typsystem für die Grundversion unsers Objektkalküls vollständig. In [AC94] wird argumentiert, daß die gleichzeitige Verwendung von ko- und kontravarianten Feldern nicht möglich ist, da sie zu Widersprüchen führt. Das ist zwar für das dort konstruierte Typsystem richtig, aber, wie wir im nächsten Abschnitt beweisen werden, ist dies nicht der Fall für das eben definierte Typsystem. In [QKB96] wird aus demselben Grund für die Eingabeparameter Invarianz verlangt, wobei argumentiert wird, daß in der Praxis kontravariante Eingabeparameter nie benötigt werden. Selbst wenn dies stimmen sollte, ist es aber es sicherlich von Vorteil, wenn man auch den allgemeineren Fall formal behandeln kann.

2.6 Typerhaltung

In diesem Abschnitt wollen wir zeigen, daß der Typ eines Terms bei der Reduktion erhalten bleibt. Daraus werden wir im nächsten Abschnitt einige interessante Eigenschaften des getypten Kalküls herleiten.

Im vorigen Abschnitt haben wir dazu den Begriff des varianzkonsistenten Terms eingeführt. Und nur für diese Terme können wir die Typerhaltung auch zeigen. Das ist allerdings keine wirkliche Einschränkung, da jeder Term, der von einem in der Praxis auftretenden Programm herrührt, auch konsistent sein wird.

Bei dem Versuch, die Typerhaltung für konsistente Terme direkt zu beweisen, ergibt sich aber das Problem, daß die Konsistenz selbst von der Reduktion nicht erhalten wird. Die Lösung besteht darin, zu etwas stärkeren Typregeln überzugehen, die für konsistente Terme mit den alten Typregeln übereinstimmen, aber noch genügend zusätzliche Information beinhalten, um die Typerhaltung beweisen zu können. Für das stärkere Typsystem benutzen wir $\Vdash$ statt $\vdash$.

Definition 2.23 Das Urteil $\Gamma \vdash A : T$ wird aufgeteilt in die zwei Urteile $\Gamma \Vdash A :: T$ und $\Gamma \Vdash A : T$. Die Intuition dabei ist die folgende: Das Urteil $\Gamma \Vdash A :: T$ soll gelten, falls die Herleitung ohne die Abschwächungsregel

`val subs`

$$\frac{\Gamma \Vdash A : S \qquad \Gamma \vdash S \leq T}{\Gamma \Vdash A : T}$$

`val self`

$$\frac{}{\Gamma, \alpha : S \Vdash \texttt{\$} :: \alpha}$$

`val read`

$$\frac{\Gamma \Vdash A : S \qquad S.N^{+} = T}{\Gamma \Vdash A.N^{+} : T}$$

`val write`

$$\frac{\Gamma \Vdash B : T \qquad \Gamma \Vdash A : S \qquad S.N^{-} = T}{\Gamma \Vdash A.N^{-} \Leftarrow B : S}$$

`val read’`

$$\frac{\Gamma, \alpha : S \Vdash A :: \alpha \qquad S.N = T}{\Gamma, \alpha : S \Vdash A.{\downarrow}N : T}$$

`val write’`

$$\frac{\Gamma, \alpha : S \Vdash B : T \qquad \Gamma, \alpha : S \Vdash A :: \alpha \qquad S.N = T}{\Gamma, \alpha : S \Vdash A.{\downarrow}N \Leftarrow B :: \alpha}$$

`val read’’`

$$\frac{\Gamma \Vdash A :: S \qquad S.N = T}{\Gamma \Vdash A.N : T}$$

`val write’’`

$$\frac{\Gamma \Vdash B : T \qquad \Gamma \Vdash A :: S \qquad S.N = T}{\Gamma \Vdash A.N \Leftarrow B :: S}$$

`val exec`

$$\frac{\Gamma \Vdash A : S \qquad \sigma M \in S}{\Gamma \Vdash A.\sigma M : S}$$

`val exec’`

$$\frac{\Gamma, \alpha : S \Vdash A :: \alpha \qquad \sigma M \in S}{\Gamma, \alpha : S \Vdash A.\sigma M :: \alpha}$$

`val exec’’`

$$\frac{\Gamma \Vdash A :: S \qquad \sigma M \in S}{\Gamma \Vdash A.\sigma M :: S}$$

`val conv`

$$\frac{\Gamma \Vdash A :: S}{\Gamma \Vdash A : S}$$

`val obj`

$$\frac{\Gamma, \alpha : S \Vdash B_j :: \alpha \qquad \alpha \notin \mathrm{FV}(\Gamma) \qquad \Gamma \Vdash A_i : T_i \qquad S = \{N_i : T_i, \sigma M_j\}}{\Gamma \Vdash [N_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j] :: S}$$

Abbildung 2.4: Der Reduktionskalkül: Termtypen `val subs` ausgekommen ist. Dagegen ist es erlaubt, bei der Herleitung von $\Gamma \Vdash A : T$ den Typ T abzuschwächen. Die Regeln sind in Abbildung 2.4 aufgelistet.

Man erhält die neuen aus den alten Regeln, indem man jeweils das passende der beiden Urteile auswählt. Außerdem muß bei den ungestrichenen `read`, `write` und `exec`-Regeln sowohl die konsistente Verwendung von Feldern, möglicherweise mit der Abschwächungsregel, als auch die beliebige Verwendung von Feldern ohne die Abschwächungsregel erlaubt werden.

Den Zusammenhang zwischen beiden Typsystemen zeigt das folgende

Lemma 2.24 Sei A konsistent. Dann gilt $\Gamma \vdash A : T$ genau dann, wenn $\Gamma \Vdash A : T$ oder $\Gamma \Vdash A :: T$ gilt.

Beweis: Durch Inspektion der Regeln erkennt man, daß bei konsistenten Regeln die Herleitungen ineinander überführt werden können. $\square$

Typvariablen können auf die übliche Art und Weise substituiert werden:

Definition 2.25 Wir schreiben $S\,[\alpha := T]$ für den Typ, der entsteht, wenn man in S alle Vorkommen von α durch T ersetzt.

Das es in Typen keine Bindung von Typvariablen gibt, muß man hier vor der Substitution auch nicht umbenennen. Für die Typsubstitution gilt wie üblich ein Vertauschungslemma:

Lemma 2.26 (Vertauschbarkeit der Typsubstitution) Es gilt

$$R\,[\alpha := S]\,[\beta := T] \quad = \quad R\,[\beta := T]\,[\alpha := S\,[\beta := T]\,]$$

Beweis: Eine einfache Induktion über den Aufbau des Typs R. $\square$

Als nächstes zeigen wir ein Substitutionslemma für alle drei Arten von Typregeln:

Lemma 2.27 (Substitutionslemma für Typen) Aus

$$\Gamma, \alpha : S, \Gamma' \vdash T \ \mathsf{type} \qquad \text{und} \qquad \Gamma, \Gamma' \vdash S \ \mathsf{type}$$

folgt die Aussage $\Gamma, \Gamma' \vdash T\,[\alpha := S]\ \mathsf{type}$.

Beweis: Man induziert über die Herleitung von T. Bei Anwendung der Regel `type var` fährt man fort mit der gegebenen Herleitung von S. Bei der Regel `type obj` verwendet man die induktive Definition der Typsubstitution und die Induktionsvoraussetzung. Auf diese Weise erhält man eine Herleitung von $T\,[\alpha := S]$. $\square$

Lemma 2.28 (Substitutionslemma für Untertypen) Ist der Typ S im Kontext Γ wohlgeformt, das heißt, gilt $\Gamma \vdash S$ `type`, und weiß man außerdem $\Gamma, \alpha : S \vdash R \leq T$, dann folgt $\Gamma \vdash R\,[\alpha := S] \leq T\,[\alpha := S]$.

Beweis: Induktion über die Herleitung von $R \leq T$: Die Regel `sub refl` ist klar. Bei der Regel `sub var` treten zwei Fälle auf. Ist die Variable β in `sub var` von α verschieden, dann muß $\alpha \notin \mathrm{FV}(T)$ gelten, da der Kontext wohlgeformt ist. Damit ist $R\,[\alpha := S] = \beta\,[\alpha := S] = \beta$, und $T\,[\alpha := S] = T$, weshalb die Behauptung gilt. Ist andererseits β gleich α, dann ergibt die Substitution $S \leq S$, was durch `sub refl` hergeleitet werden kann. Die Regeln `sub obj` und `sub trans` lassen sich wieder direkt durch Induktion abhandeln. Bei der Regel `sub ext` benutzt man zusätzlich das vorherige Lemma 2.27. □

Lemma 2.29 (Substitutionslemma für Termtypen) Es gelte das Urteil $\Gamma \Vdash A :: S$, und es sei $\alpha \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(S)$. Dann folgt:

(i)	$\Gamma, \alpha : S \Vdash B : T \implies \Gamma \Vdash B\,[\$:= A] : T\,[\alpha := S]$
(ii)	$\Gamma, \alpha : S \Vdash B :: T \implies \Gamma \Vdash B\,[\$:= A] :: T\,[\alpha := S]$

Beweis: Wieder mit Induktion über die Herleitung von $B : T$ beziehungsweise von $B :: T$. Zur Abkürzung sei $\Gamma' = \Gamma, \alpha : S$. Wir unterscheiden nach der zuletzt verwendeten Regel:

`val subs` Es gelte $\Gamma' \vdash R \leq T$ und $\Gamma' \Vdash B : R$. Aus dem vorigen Lemma folgt dann $\Gamma \vdash R\,[\alpha := S] \leq T\,[\alpha := S]$. Mittels Induktion erhält man außerdem $\Gamma \vdash B\,[\$:= A] : R\,[\alpha := S]$. Das heißt aber $\Gamma \Vdash B\,[\$:= A] : T\,[\alpha := S]$.

`val self` Führt man die Substitution aus, so erhält man genau die Voraussetzung $\Gamma \Vdash A :: S$.

`val read` Ist $B \equiv C.N^+$ und $\Gamma' \Vdash C : R$ mit $R.N^+ = T$, dann folgt mittels Induktion $\Gamma \Vdash C\,[\$:= A] : R\,[\alpha := S]$. Da aber $R\,[\alpha := S]\,.N^+ = T\,[\alpha := S]$, erhält man $\Gamma' \Vdash B\,[\$:= A] : T\,[\alpha := S]$.

`val read'` Gilt $B \equiv C.{\downarrow}N$, dann muß die Prämisse $\Gamma' \Vdash C :: \alpha$ lauten, und es gilt $S.N = T$. Mit Induktion folgt $\Gamma \Vdash C\,[\$:= A] :: S$, und durch Anwenden der Regel `val read''` erhält man $\Gamma \Vdash B\,[\$:= A] : T$. Da aber $\alpha \notin \mathrm{FV}(S) \supseteq \mathrm{FV}(T)$, gilt auch $\Gamma \Vdash B\,[\$:= A] : T\,[\alpha := S]$.

`val read''` Analog zu `val read`.

`val write` Ist $B \equiv C.N{\Leftarrow}C'$ und gilt $\Gamma' \Vdash C : R$ sowie $\Gamma' \Vdash C' : R'$, dann folgt mit Induktion $\Gamma \Vdash C\,[\$:= A] : R\,[\alpha := S]$ und $\Gamma \Vdash C'\,[\$:= A]$:

$R' [\alpha := S]$. Durch Anwenden der Regel `val write` erhält man dann das gewünschte Ergebnis.

`val write'` Gilt wieder $B \equiv C.N \Leftarrow C'$, dann muß die Prämisse hier von der Form $\Gamma' \Vdash C :: \alpha$ sein, wobei außerdem $S.N = T$ gilt. Da wieder $\alpha \notin \mathrm{FV}(T)$, kommt man analog zum Fall `val read'` durch Anwenden der Regel `val write''` zum Ziel.

`val write''` Analog zu `val write`.

`val exec` Direkt durch Induktion.

`val exec'` Analog zu `val read'` und `val write'` durch Ausführen der Substitution und Anwenden der Regel `val exec''`.

`val exec''` Wieder direkt durch Induktion.

`val obj` Sei $B = [N_i\texttt{=}C_i, \sigma M_j\texttt{=}D_j]$ und $T = \{N_i : R_i, \sigma M_j\}$. Mit der Induktionsannahme erhält man $\Gamma \Vdash C_i [\$:= A] : R_i [\alpha := S]$. Daraus folgt mit der Regel `val obj` $\Gamma \Vdash B [\$:= A] : T [\alpha := S]$.

`val conv` Trivial. $\square$

Bevor wir jetzt die Typerhaltung zeigen können, müssen wir noch beweisen, daß bei Objekttypen die möglicherweise mehrfache Anwendung der Abschwächungsregel auf eine kanonische Anwendung zurückgeführt werden kann:

Lemma 2.30 Falls $\Gamma \Vdash [L] : S'$, dann kann man einen Typ S finden, so daß schon $\Gamma \Vdash [L] :: S$ und $\Gamma \vdash S \leq S'$.

Beweis: Durch Inspektion der Regeln erkennt man, daß die Herleitung von $\Gamma \Vdash [L] : S'$ nur dadurch zustande gekommen sein kann, daß zuvor unter Umständen mehrmals die Regel `val subs`, davor die Regel `val conv`, und schließlich davor die Regel `val obj` angewandt worden ist. Die Regel `val obj` liefert eine Herleitung für $\Gamma \Vdash [L] :: S$, und durch Zusammenfassen der Abschwächungen in den Anwendungen von `val subs` mittels `sub trans` erhält man $\Gamma \vdash S \leq S'$. $\square$

Satz 2.31 (Typerhaltung) Es gelte $A \to A'$. Dann folgt:

(i) $\quad \Gamma \Vdash A : S \implies \Gamma \Vdash A' : S$
(ii) $\quad \Gamma \Vdash A :: S \implies \Gamma \Vdash A' :: S$

Beweis: Wir unterscheiden zunächst nach der Reduktionsregel und dann nach der Typregel, durch die man $\Gamma \Vdash A : S$ beziehungsweise $\Gamma \Vdash A :: S$ erhalten hat. Die Kontextregeln der Reduktion kann man jeweils durch die offensichtliche Induktion behandeln. Keiner dieser Fälle ist sonderlich interessant. Die Regeln für die Redexe sehen so aus:

`val read` Ist $A \equiv [N^+\texttt{=}B, L].N^+$ sowie $A' \equiv B$, und wurde $\Gamma \Vdash A : S$ mit `val read` hergeleitet, dann muß $\Gamma \Vdash [N^+\texttt{=}B, L] : T$ und $T.N^+\texttt{=}S$ gelten. Aus Lemma 2.30 folgt, daß es einen Typ T' gibt mit $\Gamma \Vdash [N^+\texttt{=}B, L] :: T'$ und $\Gamma \vdash T' \leq T$. Sei $S'\texttt{=}T'.N^+$. Nach der Herleitung von T' muß dann gelten $\Gamma \Vdash B : S'$. Da N^+ kovariant ist, gilt ferner $\Gamma \vdash S' \leq S$. Durch Anwenden der Regel `val subs` bekommt man dann eine Herleitung für $\Gamma \Vdash B : S'$.

`val read'` Kann nicht vorkommen, da $[L].{\downarrow}N$ kein Redex ist.

`val read''` Hier kann man argumentieren wie bei `val read`, nur daß Lemma 2.30 und die Abschwächung nicht verwendet werden müssen, weshalb hier auch nicht benötigt wird, daß das Feld kovariant ist.

`val write` Ist $A \equiv [N^-\texttt{=}B, L].N^- \Leftarrow C$ und $A' \equiv [N^-\texttt{=}C, L]$, dann muß wegen der Herleitung von S auch $\Gamma \Vdash [N^-\texttt{=}B, L] : S$ mit $S.N^- = T$ gelten. Wie oben findet man nach Lemma 2.30 einen Typ S' mit $\Gamma \Vdash [N^-\texttt{=}B, L] :: S'$ und $\Gamma \vdash S' \leq S$. Es sei $S'.N^- = T'$. Um die Herleitung für $[N^-\texttt{=}C, L] : S$ zu konstruieren, nehmen wir die Herleitung für $[N^-\texttt{=}B, L] :: S'$ gefolgt von einer Abschwächung auf S, wobei man dann aber statt $\Gamma \Vdash B : T'$ zeigen muß, daß $\Gamma \Vdash C : T'$. Bei der ursprünglichen Herleitung von S wurde schon $\Gamma \Vdash C : T$ gezeigt. Da N^- kontravariant ist, kann man aber $\Gamma \vdash T \leq T'$ aus $\Gamma \vdash S' \leq S$ folgern. Also erhält man mit einer weiteren Abschwächung wie gewünscht $\Gamma \Vdash C : T'$.

`val write'` Kann nicht vorkommen, da $[L].{\downarrow}N \Leftarrow B$ kein Redex ist.

`val write''` Man geht vor wie bei `val write''`, nur daß man keine Abschwächungen betrachten muß. Daher spielt es auch wieder keine Rolle, ob das Feld kontravariant ist.

`val exec` Es sei zur Abkürzung $B \equiv [\sigma M\texttt{=}C, L]$. Falls dann $A \equiv B.\sigma M$ und $A' \equiv C\,[\$:= B]$ ist, dann muß wegen der Herleitung von S schon $\Gamma \Vdash B : S$ gegolten haben. Nach Lemma 2.30 findet sich wieder ein Typ S' mit $\Gamma \vdash S' \leq S$ und $\Gamma \Vdash B :: S'$. Daher muß auch $\Gamma, \alpha : S' \Vdash C :: \alpha$ gegolten haben, wobei $\alpha \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(S')$. Nach dem

Substitutionslemma 2.29 folgt dann aber $\Gamma \Vdash C\,[\$:= B] :: \alpha\,[\alpha := S']$. Durch Abschwächung erhält man schließlich $\Gamma \Vdash C\,[\$:= B] : S$.

`val exec'` Kann nicht vorkommen, da $[L] :: \alpha$ nicht herleitbar ist, wie man durch Inspektion von `val obj` sieht.

`val exec''` Können wir wieder genauso wie `val exec` bewiesen, ohne Lemma 2.30 und Abschwächung zu benutzen. □

2.7 Zusammenfassung der Eigenschaften

Wir wollen noch einmal die Eigenschaften des Objektkalküls zusammenstellen:

- Der Kalkül hat die Church-Rosser-Eigenschaft, das heißt, verschiedene Auswertungsstrategien liefern dasselbe Ergebnis.
- Läßt sich für einen Term A ein Typ T durch $\Vdash A : T$ finden, dann läßt sich derselbe Typ auch für diejenigen Terme finden, die aus A durch Reduktion hervorgehen.
- Da der Fehler $\Diamond$ nicht typisierbar ist, folgt daraus, daß ein typisierbarer Term nicht zu einem Fehler reduzieren kann.
- Für konsistente Terme A reicht es dabei schon, wenn man durch $\vdash A : T$ für sie einen Typ finden kann.
- Schließlich folgt aus der Abschwächungsregel sofort, daß man in jedem Kontext einen Term durch einen anderen Term mit stärkerem Typ ersetzen kann.

Damit haben wir unsere wesentlichen Ziele erreicht: Durch das Typsystem kann man sicherstellen, daß bei der Ausführung kein Fehler entsteht, und man kann die Wiederverwendbarkeitsidee des objekt-orientierten Ansatzes im Typsystem wiederspiegeln.

Noch nicht klar ist, ob man auch alle Terme typisieren kann, die in der Praxis auftretenden Programmen entsprechen. Und in der Tat gibt es hier ein Problem: In den Beispielen in diesem Kapitel hatten wir Felder immer mit dem Fehlerterm $\Diamond$ belegt, wenn noch kein Wert bekannt war. Aber dann ist die Typisierung nicht möglich. Im nächsten Kapitel werden wir dazu eine Lösung entwickeln.

Kapitel 3

Initialisierung

3.1 Explizite Initialisierung

Wir erweitern das im letzten Kapitel vorgestellten Typsystem um den Begriff der *zu initialisierenden Typen*. In funktionalen Sprachen müssen den Funktionen Argumente gegeben werden, damit sie berechnet werden können. Bei Objekten entspricht dies der Forderung, daß Felder mit Werten überschrieben werden müssen, bevor eine Methode ausgeführt wird.

Die neuen Typen bilden wir wie folgt:

Definition 3.1 Ein Typ T wird nach der Grammatik

$$T \quad ::= \quad \alpha \mid \{N_i : T_i, \sigma M_j\} \mid \Lambda N.T$$

gebildet. $\Lambda N.T$ soll bedeuten, daß im Typ T des Feld N noch überschrieben werden muß. Will man notieren, daß mehrere Felder überschrieben werden müssen, so kann man $\Lambda N_1.\Lambda N_2.\cdots\Lambda N_k.T$ durch $\Lambda N_i.T$ abkürzen, wobei wir wieder den Bereich $i \in \{1, \ldots, k\}$ weglassen. Wenn wir einen Typ in dieser abgekürzten Form $\Lambda N_i.T$ hinschreiben, dann meinen wir damit ab sofort, daß T nicht von der Form $\Lambda N.T'$ ist, das heißt, die Liste ΛN_i schon alle zu überschreibenden Felder enthält.

Weiter spielt die Reihenfolge der ΛN keine Rolle: Die Typen $\Lambda N.\Lambda N'.T$ und $\Lambda N'.\Lambda N.T$ sind gleich. Man kann diese Gleichheit auch formal in die Typregeln einbauen, aber wir werden sie der Einfachheit halber ab sofort in der Notation unterdrücken.

Für die Typisierung geben wir gleich die Regeln für den Reduktionskalkül an. Die vereinfachten Regeln für konsistente Terme kann man sich sofort aus diesen Regeln herleiten.

`type init`
$$\frac{\Gamma \vdash T \ \mathsf{type}}{\Gamma \vdash \Lambda N.T \ \mathsf{type}}$$

`sub init right`
$$\frac{}{\Gamma \vdash S \leq \Lambda N.S}$$

`sub init both`
$$\frac{\Gamma \vdash S \leq T}{\Gamma \vdash \Lambda N.S \leq \Lambda N.T}$$

Abbildung 3.1: Der Initialisierungskalkül: Zusatzregeln für (Unter)-Typen

`val subs`
$$\frac{\Gamma \Vdash A : S \qquad \Gamma \vdash S \leq T}{\Gamma \Vdash A : T}$$

`val self`
$$\frac{}{\Gamma, \alpha : S \Vdash \$ :: \alpha}$$

`val read`
$$\frac{\Gamma \Vdash A : \Lambda N_i.S \qquad S.N^+ = T \qquad N^+ \neq N_i}{\Gamma \Vdash A.N^+ : T}$$

`val write`
$$\frac{\Gamma \Vdash B : T \qquad S.N^- = T \qquad \Gamma \Vdash A : \Lambda N^-.\Lambda N_i'.S}{\Gamma \Vdash A.N^- \Leftarrow B : \Lambda N_i'.S}$$

`val write'`
$$\frac{\Gamma, \alpha : S \Vdash B : T \qquad S.N = T \qquad \Gamma, \alpha : S \Vdash A :: \Lambda N.\Lambda N_i'.\alpha}{\Gamma, \alpha : S \Vdash A.{\downarrow}N \Leftarrow B :: \Lambda N_i'.\alpha}$$

`val read'`
$$\frac{\Gamma, \alpha : S \Vdash A :: \alpha \qquad S.N = T}{\Gamma, \alpha : S \Vdash A.{\downarrow}N : T}$$

`val read''`
$$\frac{\Gamma \Vdash A :: \Lambda N_i.S \qquad S.N = T \qquad N \neq N_i}{\Gamma \Vdash A.N : T}$$

`val write''`
$$\frac{\Gamma \Vdash B : T \qquad S.N = T \qquad \Gamma \Vdash A :: \Lambda N.\Lambda N_i'.S}{\Gamma \Vdash A.N \Leftarrow B :: \Lambda N_i'.S}$$

`val exec`
$$\frac{\Gamma \Vdash A : S \qquad \sigma M \in S \qquad S \neq \Lambda N.S}{\Gamma \Vdash A.\sigma M : S}$$

`val exec'`
$$\frac{\Gamma, \alpha : S \Vdash A :: \alpha \qquad \sigma M \in S}{\Gamma, \alpha : S \Vdash A.\sigma M :: \alpha}$$

`val obj`
$$\frac{\Gamma, \alpha : S \Vdash B_j :: \alpha \qquad \alpha \notin \mathrm{FV}(\Gamma) \qquad \Gamma \vdash T_i \ \mathsf{type} \qquad S = \{N_i : T_i, \sigma M_j\}}{\Gamma \Vdash [N_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j] :: \Lambda N_i.S}$$

`val exec''`
$$\frac{\Gamma \Vdash A :: S \qquad \sigma M \in S}{\Gamma \Vdash A.\sigma M :: S}$$

`val init`
$$\frac{\Gamma \Vdash A : T \qquad S.N\texttt{=}T \qquad \Gamma \Vdash [N\texttt{=}A, L] :: \Lambda N.\Lambda N_i.S}{\Gamma \Vdash [N\texttt{=}A, L] :: \Lambda N_i.S}$$

`val conv`
$$\frac{\Gamma \Vdash A :: S}{\Gamma \Vdash A : S}$$

Abbildung 3.2: Der Initialisierungskalkül: Termtypen

Definition 3.2 Für den Typkalkül mit expliziter Initialisierung fügen wir zu den `type` und `sub`-Regeln für den Grundkalkül die Regeln in Abbildung 3.1 hinzu. Ein Term des Grundkalküls kann dann nach den `val`-Regeln in Abbildung 3.2 getypt werden.

Die Untertyp-Regeln ermöglchen es, jederzeit durch Abschwächung die Forderung hinzuzufügen, daß noch Felder zu initialisieren sind. Dies paßt einerseits in das Konzept dessen, was durch die Untertyprelation ausgedrückt werden soll (ein Term eines umfassenderen Typs soll überall da eingesetzt werden können, wo ein Term eines schwächeren Typs steht) und erleichtert andererseits die Formulierung der übrigen Regeln.

Diese erlauben ein Lesen von Feldern nur, wenn das Feld bereits initialisiert ist, das heißt, N in `val read` von allen N_i verschieden ist. Bei Überschreiben eines Feldes N ist es möglich, das zugehörige ΛN im Typ zu entfernen. Eine Methode schließlich darf nur ausgeführt werden, wenn keine Felder mehr initialisiert werden müssen. Nach der Regel `val init` kann man bei einem Objekttyp ein ΛN entfernen, wenn man nachweisen kann, daß das Feld den passenden Typ hat. Dies vereinfacht auch die Regel `val obj`, bei der nur noch geprüft werden muß, ob die Feldtypen in Γ auch wohlgeformt sind.

Beispiel 3.3 Mit Hilfe der Initialisierung kann man auch Terme typen, die den Fehler $\Diamond$ enthalten. So findet man zum Beispiel für den Term

$$\mathsf{ID} \equiv [\mathsf{arg}^-\mathtt{=}\Diamond, \mathsf{res}^+\mathtt{=}\Diamond, \sigma\mathsf{id}\mathtt{=}\$.\mathsf{res}^+ \Leftarrow (\$.\mathsf{arg}^-)]$$

den Typ

$$\Lambda\mathsf{arg}^-.\Lambda\mathsf{res}^+.\{\mathsf{arg}^- : T, \mathsf{res}^+ : T, \sigma\mathsf{id}\}$$

für jeden beliebigen Typ T.

Obwohl es also sein kann, daß das Fehlersumbol $\Diamond$ in einem typbaren Term vorkommt, so soll der Term dennoch nicht zu einem Fehler reduzieren können. Dazu müssen wir für den neuen Typkalkül wieder die Typerhaltung bei der Reduktion beweisen.

Lemma 3.4 Das Vertauschbarkeitslemma 2.26 und die Substitutionslemmas 2.27, 2.28 und 2.29 gelten auch für den neuen Typkalkül.

Beweis: Die Anwesenheit von ΛN in den Typen ändert nichts am Beweis der Lemmas. Die neue Regel `val init` erfordert eine triviale Anwendung der Induktionsbehauptung, und bei `val obj` muß man zusätzlich auf Lemma 2.27 verweisen. Auch das Zurückführen der einfach gestrichenen Regeln auf die zweifach gestrichenen Regeln nach der Substitution bereitet keine Schwierigkeiten, da man in den zweifach gestrichenenen Regeln den leeren ΛN_i-Präfix wählen kann. $\square$

Das Lemma über Objekttypen benötigen wir jetzt in etwas stärkerer Form:

Lemma 3.5 Es gelte $\Gamma \Vdash [L] : \Lambda N_i.S$, wobei alle N_i auch in S vorkommen sollen. Dann gibt es einen Typ S', so daß $\Gamma \Vdash [L] :: \Lambda N_i.\Lambda N'_j.S'$ und $\Gamma \vdash S' \leq S$. Außerdem ist $[L]$ von der Form $[L] \equiv [N'_j\texttt{=}A_j, L']$, und es gilt $\Gamma \Vdash A_j : S.N'_j$ für jedes j.

Beweis: Man erkennt durch Inspektion der Regeln, daß die Herleitung von $\Gamma \Vdash [L] : \Lambda N_i.S$ geendet haben muß mit mehrmaliger oder keinmaliger Anwendung von `val subs` nach einer Anwendung von `val conv`. Davor wurde `val init` unter Umständen mehrfach auf das Ergebnis von `val obj` angewendet.

Die mehrmalige Anwendung von `val subs` läßt sich wieder mittels `sub trans` zu einer Anwendung zusammenfassen. Diese Regel habe als Prämisse $\Gamma \vdash T \leq \Lambda N_i.S$. Dann sind auch alle Felder N_i in T vorhanden, da die Regel `sub ext` höchstens Felder zu T hinzufügen kann. Die Regel `val obj` muß also eine Herleitung der Form $\Gamma \Vdash [L] :: \Lambda N_i.\Lambda N'_j.S'$ liefern, da alle N_i in T vorhanden waren und nachfolgenden `val init`-Regeln nur voranstehende ΛN beeinflussen. Alle $\Lambda N'_j$ müssen aber irgendwann durch `val init` Regeln entfernt werden. Also gilt $\Gamma \Vdash A_j : S.N'_j$ für diese Felder. Schließlich ergibt sich $\Gamma \vdash S' \leq S$, wenn man die zusammengefaßte Abschwächung betrachtet und dabei die `sub init`-Regeln ignoriert. $\square$

Und schließlich muß die Typerhaltung neu bewiesen werden:

Satz 3.6 Auch für den erweiterten Typkalkül gilt: Reduziert A in einem Schritt zu A', dann folgt

(i) $\Gamma \Vdash A : S \Longrightarrow \Gamma \Vdash A' : S$
(ii) $\Gamma \Vdash A :: S \Longrightarrow \Gamma \Vdash A' :: S$

Beweis: Wir listen nur noch die Fälle für die ungestrichenen Typregeln auf. Die anderen Typregeln kann man wie in Satz 2.31 auf deren Beweis zurückführen, sofern sie überhaupt vorkommen können.

`val read` Falls $A \equiv [N^+\texttt{=}B, L].N^+$ und $A' \equiv B$, dann gilt wegen `val read` auch $\Gamma \Vdash [N^+\texttt{=}B, L] : \Lambda N_i.T$ und $T.N^+\texttt{=}S$. Daher folgt aus Lemma 3.5, daß es einen Typ T' gibt mit $\Gamma \Vdash [N^+\texttt{=}B, L] :: \Lambda N_i.\Lambda N'_j.T'$ und $\Gamma \vdash T' \leq T$. Da N von allen N_i verschieden ist, muß außerdem $\Gamma \Vdash B : T'.N^+$ gelten. Nun ist N^+ kovariant, also hat man

$\Gamma \vdash T'.N^+ \leq T.N^+$. Durch Anwenden der Regel `val subs` bekommt man dann eine Herleitung für $\Gamma \Vdash B : S'$.

`val write` Ist $A \equiv [N^-\texttt{=}B, L].N^- \Leftarrow C$ und $A' \equiv [N^-\texttt{=}C, L]$, dann ist S von der Form $S = \Lambda N.\Lambda N_i.R$, und es gilt $\Gamma \Vdash [N^-\texttt{=}B, L] : S$ mit $R.N^- = T$. Lemma 3.5 liefert einen Typ $S' = \Lambda N.\Lambda N_i.\Lambda N_j.R'$ mit $\Gamma \Vdash [N^-\texttt{=}B, L] :: S'$ und $\Gamma \vdash R' \leq R$. Es sei $T' = R'.N^-$.

Wir wollen eine Herleitung für $[N^-\texttt{=}C, L] : S$ konstruieren. Dazu starten wir mit der Herleitung für $[N^-\texttt{=}B, L] :: S'$. Im Beweis von Lemma 3.5 haben wir gesehen, daß darauf Anwendungen von `val init` und schließlich eine Anwendung von `val conv` folgen. Wir fügen eine zusätzliche Anwendung von `val init` ein, die das ΛN entfernt. Dazu benötigen wir eine Herleitung von $\Gamma \Vdash C : T'$. Diese existiert, da wegen der Kontravarianz von N^- gilt $\Gamma \vdash T \leq T'$ und wir daher die Prämisse $\Gamma \Vdash C : T$ aus der ursprünglichen `val write`-Regel nehmen und abschwächen können. Schließlich schwächen wir nach der `val conv`-Regel noch S' auf S ab und erhalten so $\Gamma \Vdash [N^-\texttt{=}C, L] : S$.

`val exec` Der Beweis verläuft genau wie in Satz 2.31, nur daß man bei der Anwendung von 2.30 bemerken muß, daß die `val init`-Regel für alle Felder angewandt worden ist, so daß man sie auch nach der Substitution wieder für alle Felder anwenden kann.

□

3.2 Implizite Initialisierung

In einem weiteren Schritt wollen wir den Kalkül so erweitern, daß auch beim Aufruf von Methoden eine implizite Initialisierung durchgeführt werden kann. Insbesondere soll kenntlich gemacht werden, welche Ausgabefelder eine Methode auf jeden Fall überschreibt und welche Eingabefelder sie normalerweise erwartet. Damit ist der Typkalkül dann stark genug, um Funktionen, die als Objekte ausgedrückt werden, wirklich typisieren zu können.

Definition 3.7 Ein Typ kann jetzt auf folgende Art gebildet werden:

$$T \;::=\; \alpha \mid \{N_i : T_i, \sigma M_j(N_{jk} \rightarrow N_{jl})\} \mid \Lambda N.T$$

Dabei sollen die Feldnamen hinter einer Methode ausdrücken, welche Felder N_{jk} beim Aufruf Werte erhalten sollen und welche Felder N_{jl} auf jeden Fall durch die Methode überschrieben werden.

Wir schreiben $\sigma M(N_i) \in S$, wenn S garantiert, daß die Methode σM die Felder N_i überschreibt, also S von der Form $S = \{\sigma M(N_j' \rightarrow N_i), \ldots\}$ ist.

`val self`

$$\frac{}{\Gamma, \alpha : \Lambda N_i.S \Vdash \$ :: \Lambda N_i.\alpha}$$

`val read'`

$$\frac{\Gamma, \alpha : \Lambda N_i.S \Vdash A :: \Lambda N_i.\alpha \qquad S.N = T \qquad N \neq N_i}{\Gamma, \alpha : \Lambda N_i.S \Vdash A.{\downarrow}N : T}$$

`val write'`

$$\frac{\Gamma, \alpha : \Lambda N_i.S \Vdash B : T \qquad S.N = T \qquad \Gamma, \alpha : \Lambda N_i.S \Vdash A :: \Lambda N.\Lambda N'_j.\alpha}{\Gamma, \alpha : \Lambda N_i.S \Vdash A.{\downarrow}N \mathrel{\lhd\!\!-} B :: \Lambda N'_j.\alpha}$$

`val exec`

$$\frac{\Gamma \Vdash A : \Lambda N_i.S \qquad \sigma M(N_i) \in S}{\Gamma \Vdash A.\sigma M : S}$$

`val exec'`

$$\frac{\Gamma, \alpha : \Lambda N_i.S \Vdash A :: \Lambda N'_j.\alpha \qquad \sigma M(N'_j) \in S}{\Gamma, \alpha : \Lambda N_i.S \Vdash A.\sigma M :: \alpha}$$

`val exec''`

$$\frac{\Gamma \Vdash A :: \Lambda N_i.S \qquad \sigma M(N_i) \in S}{\Gamma \Vdash A.\sigma M :: S}$$

`val obj`

$$\frac{\Gamma, \alpha : \Lambda N_{jk}.S \Vdash B_j :: \alpha \qquad \alpha \notin \mathrm{FV}(\Gamma) \qquad \Gamma \vdash T_i \ \mathsf{type} \qquad S = \{N_i : T_i, \sigma M_j(N_{jk})\}}{\Gamma \Vdash [N_i\mathtt{=}A_i, \sigma M_j\mathtt{=}B_j] :: \Lambda N_i.S}$$

Abbildung 3.3: Der implizite Initialisierungskalkül

Bemerkung 3.8 Für die Typherleitung Bedeutung werden dabei nur die Feldnamen hinter dem Pfeil haben. Die Feldnamen vor dem Pfeil werden aus folgendem Grund eingeführt: Die Schreibweise

$$A.\sigma M(N_1 \mathrel{\lhd\!\!-} A_1, N_2 \mathrel{\lhd\!\!-} A_2, \ldots, N_n \mathrel{\lhd\!\!-} A_n)$$

die an einen Funktionenaufruf mit den benannten Argumenten A_1 bis A_n erinnern soll, kann als Abkürzung dienen für den Term

$$A.N_1 \mathrel{\lhd\!\!-} A_1.N_2 \mathrel{\lhd\!\!-} A_2.\cdots.N_n \mathrel{\lhd\!\!-} A_n.\sigma M$$

Die in dieser Abkürzung erlaubten Feldnamen sind diejenigen, die beim dem Methodentyp vor dem Pfeil stehen.

Für den Rest des Kapitels werden wir diesen syntaktischen Zucker der Einfachheit halber weglassen und nur noch die Feldnamen hinter dem Pfeil berücksichtigen.

Die Typregeln, die diesmal geändert werden müssen, sind im wesentlichen die Regeln für die Methodenausführung und für den Zugriff auf eigene Felder innerhalb von Methoden.

Definition 3.9 Für den Typkalkül mit impliziter Initialisierung verändern wir die Typregeln gemäß Abbildung 3.3.

Die Herleitung wurde dahingehend verändert, daß für eine Methode mit Ergebnisfeldern N_i beim Auftreten von $ innerhalb des Methodenrumpfes $\Lambda N_i.\alpha$ als Typ gewählt wird. Diese ΛN_i müssen dann erst durch Überschreiben in `val write'` entfernt werden, damit die Regel `val obj` zur Anwendung kommen kann.

Beispiel 3.10 Wir können jetzt die Identitätsfunktion ID aus Beispiel 3.3 wirklich als Funktion darstellen und typisieren: Der Term

$$\mathsf{ID} \equiv [\mathsf{arg}^-\texttt{=}\Diamond, \mathsf{res}^+\texttt{=}\Diamond, \sigma\mathsf{id}\texttt{=}\$.\mathsf{res}^+ \Leftarrow (\$.\mathsf{arg}^-)]$$

erhält den Typ

$$\Lambda \mathsf{arg}^-.\Lambda \mathsf{res}^+.\{\mathsf{arg}^- : \mathsf{T}, \mathsf{res}^+ : \mathsf{T}, \sigma\mathsf{id}(\mathsf{arg}^- \to \mathsf{res}^+)\}$$

Wenden wir nun ID beispielsweise auf $\mathsf{X}\texttt{=}[\mathsf{x} : []]$ an: Der Term X hat den Typ $\mathsf{X} : \{\mathsf{x} : \{\}\}$, und auch das Ergebnis der Anwendung läßt sich durch

$$\mathsf{ID}.\sigma\mathsf{id}(\mathsf{arg}^- \Leftarrow \mathsf{X}).\mathsf{res}^- : \{\mathsf{x} : \{\}\}$$

typisieren, was vorher nicht möglich war.

Wiederum ist die Typerhaltung zu zeigen. Alle Lemmas außer dem Substitutionslemma übertragen sich sofort.

Lemma 3.11 Das Substitutionslemma 2.29 gilt auch für den Typkalkül mit impliziter Initialisierung.

Beweis: Außer den in Lemma 3.4 angesprochenen Veränderungen bleibt der Beweis gleich. Die einfach gestrichenen Regeln passen nun exakt zu den zweifach gestrichenen Regeln, so daß es auch hier bei der Zurückführung der einen auf die anderen nach der Substitution keine Schwierigkeiten gibt. □

Satz 3.12 Für den Kalkül mit impliziter Initialisierung gilt die Typerhaltung.

Beweis: Alle Fälle bis auf die `val exec`-Regeln bleiben gleich. Sei also mit den Bezeichnungen von 3.6 $A \equiv B.\sigma M$ und $A' \equiv C\,[\$:= B]$, wobei B eine Abkürzung für das Objekt $B \equiv [\sigma M\texttt{=}C, L]$ ist. Wegen der Voraussetzung $\Gamma \Vdash B.\sigma M : S$ muß auch $\Gamma \Vdash B : \Lambda N_i.S$ gelten. Mit Lemma 3.5 finden wir einen Typ S', so daß $\Gamma \Vdash B :: \Lambda N_j'.S'$ und $\Gamma \vdash S' \leq S$. Da außerdem $\sigma M(N_i) \in S$, ist auch $\sigma(N_i) \in S'$. Also wurde in `val obj` die Prämisse $\Gamma, \alpha : \Lambda N_i.S \Vdash C :: \alpha$ bewiesen. Weil zusätzlich auch $\alpha \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(S')$, kann man das Substitutionslemma anwenden, welches $\Gamma \Vdash C\,[\$:= B] :: \alpha\,[\alpha := S']$ liefert. Durch Abschwächung erhält man dann $\Gamma \Vdash C\,[\$:= B] : S$.

Der Beweis für die `val exec''` Regel verläuft wieder ganz analog, aber ohne Abschwächung; die `val exec'`-Regel kommt wie schon vorher nicht in Frage. □

Die Eigenschaften, die wir im zweiten Kapitel für den getypten Kalkül zusammengefaßt hatten, übertragen sich wörtlich auf den Kalkül mit Initialisierung.

Kapitel 4

Polymorphismus

Das Ziel dieses Kapitels soll es sein, den Kalkül mit einem Typsystem höherer Ordnung zu versehen, so daß man für im funktionalen Sinn polymorphe Terme (wie zum Beispiel die Identitätsfunktion) passende Typen finden kann. Für die Darstellung dieses Typsystems ist es dabei einfacher, vom Curry-Stil der letzten beiden Kapitel zum Church-Stil des Typisierens überzugehen, das heißt, die Typen mit in die Terme aufzunehmen.

Wie soll nun ein Typsystem höherer Ordnung für unseren Kalkül aussehen? Bei den Typsystemen höherer Ordnung für den Lambda-Kalkül ist der wesentliche Schritt, die Lambda-Abstraktion statt nur für Terme auch für Typen einzuführen. Überträgt man diese Idee auf unseren Kalkül, so bedeutet das, daß ein Objekt nicht nur Felder für Terme, sondern auch für Typen haben muß und dieser Typzustand die Typen der Felder beeinflussen kann. Diese Idee wollen wir nun umsetzen.

4.1 Syntax und Reduktionsregeln

Zuerst führen wir Terme und Typen ein.

Definition 4.1 Terme A und Typen T des Kalküls werden nach folgender Grammatik gebildet:

$$\begin{array}{lcl}
A & ::= & A.N \mid A.N \Leftarrow A' \mid A.\sigma N \mid [L \mid H] \mid \$ \mid \Diamond \mid \\
 & & A.\downarrow N \mid A.\downarrow N \Leftarrow A' \mid A.\beta \Leftarrow T \\
L & ::= & N_i : T_i \texttt{=} A_i, \sigma M_j \texttt{=} B_j \\
H & ::= & \beta_i \texttt{=} T_i \\
T & ::= & \alpha \mid \{N_i : T_i \mid H\}
\end{array}$$

Die Bezeichnungen sind fast die gleichen wie im zweiten Kapitel. L ist eine Liste von Felddefinitionen, die jetzt aber Typen haben, und von Methodendefinitionen. H ist eine Liste von Typfeldern. Ein Objekt ist nun in zwei Teile geteilt, und enthält Typfelder sowie Termfelder und Methoden. Auch der Typ eines Objektes hat passend dazu zwei Teile. Schließlich gibt es zusätzlich eine Operation, die Typfelder überschreiben kann.

Beispiel 4.2 Es sei A irgendein Term vom Typ T. Der Term ID, der die Identitätsfunktion darstellt, sieht in der polymorphen Version mit schon vorbelegten Felder so aus:

$$\mathsf{ID} \equiv [\mathsf{arg}^{-}{:}\alpha\texttt{=}A, \mathsf{res}^{+}{:}\alpha\texttt{=}A, \sigma\mathsf{id}\texttt{=}\$.\mathsf{res}{\Leftarrow}(\$.\mathsf{arg}) \mid \alpha\texttt{=}T]$$

und hat den Typ

$$\{\mathsf{arg}^{-}{:}T, \mathsf{res}^{+}{:}T, \sigma\mathsf{id} \mid \alpha\texttt{=}T\}$$

Bevor wir die Reduktionsregeln angeben können, müssen wir zunächst freie Variablen für Typen und Terme definieren:

Definition 4.3 Die *freien Variablen* $\mathrm{FV}(T)$ eines Typs T sind alle Typvariablen, die in T vorkommen, sofern sie nicht als Typfelder auftreten. Formal:

$$\begin{aligned}
&\mathrm{FV}(\alpha) = \{\alpha\}\\
&\mathrm{FV}(\{N_i : T_i, \sigma M_j \mid \beta_k : R_k\}) = (\textstyle\bigcup_i \mathrm{FV}(T_i) - \bigcup_k \{\beta_k\}) \cup \bigcup_k \mathrm{FV}(R_k)
\end{aligned}$$

Genauso sind die *freien Typvariablen* $\mathrm{FV}(A)$ eines Terms A diejenigen Typvariablen, die nicht durch ein darüberliegendes Typfeld gebunden sind, also:

$$\begin{aligned}
\mathrm{FV}([N_i : T_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j \mid \beta_k\texttt{=}R_k]) &= (\textstyle\bigcup_i \mathrm{FV}(T_i) - \bigcup_k \{\beta_k\}) \cup \bigcup_k \mathrm{FV}(R_k)\\
\mathrm{FV}(A.N) &= \mathrm{FV}(A)\\
\mathrm{FV}(A.N{\Leftarrow}A') &= \mathrm{FV}(A) \cup \mathrm{FV}(A')\\
\mathrm{FV}(A.\sigma N) &= \mathrm{FV}(A)\\
\mathrm{FV}(A.{\downarrow}N) &= \mathrm{FV}(A)\\
\mathrm{FV}(A.{\downarrow}N{\Leftarrow}A') &= \mathrm{FV}(A) \cup \mathrm{FV}(A')\\
\mathrm{FV}(A.\beta{\Leftarrow}T) &= \mathrm{FV}(A) \cup \mathrm{FV}(T)\\
\mathrm{FV}(\$) &= \{|\}\\
\mathrm{FV}(\Diamond) &= \{|\}
\end{aligned}$$

Als nächstes müssen wir zwei Substitutionen definieren:

Definition 4.4 Die Substitution von $\$$ in einem Term A durch einen Term B wird wie schon vorher durch $A\,[\$:= B]$ notiert. Sie entspricht genau der in Definition 2.3 eingeführten Substitution.

Außerdem können wir auch eine Typvariable α in einem Term A durch einen Typ T ersetzen, was durch $A\,[\alpha := S]$ ausgedrückt wird. Dabei müssen wie üblich die gebundenen Typvariablen in A unter Umstanden umbenannt werden. Eine Typvariable, die als Teilterm eines Feldes auftritt, wird dabei durch ein entsprechendes Typfeld im Objekt *gebunden*. Formal:

$$
\begin{array}{ll}
[L \mid \beta_k\texttt{=}R_k]\,[\alpha := S] & \equiv [L\,[\alpha := S] \mid \beta_k\texttt{=}R_k\,[\alpha := S]\,] \\
 & \qquad \text{falls } \bigcup_k \{\beta_k\} \cap \mathrm{FV}(S) \neq \emptyset \\
(A.N)\,[\alpha := S] & \equiv A\,[\alpha := S]\,.N \\
(A.N\Leftarrow A')\,[\alpha := S] & \equiv A\,[\alpha := S]\,.N\Leftarrow A'\,[\alpha := S] \\
(A.\sigma N)\,[\alpha := S] & \equiv A\,[\alpha := S]\,.\sigma N \\
(A.\downarrow N)\,[\alpha := S] & \equiv A\,[\alpha := S]\,.\downarrow N \\
(A.\downarrow N\Leftarrow A')\,[\alpha := S] & \equiv A\,[\alpha := S]\,.\downarrow N\Leftarrow A'\,[\alpha := S] \\
(A.\alpha\Leftarrow T)\,[\alpha := S] & \equiv A\,[\alpha := S]\,.\alpha\Leftarrow T\,[\alpha := S] \\
\$\,[\alpha := S] & \equiv \$ \\
\Diamond\,[\alpha := S] & \equiv \Diamond
\end{array}
$$

wobei für $L \equiv N_i : T_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j$ der substituierte Term $L\,[\alpha := S]$ gegeben ist durch $N_i{:}T_i\,[\beta := S]\,\texttt{=}A_i\,[\beta := S], \sigma M_j\texttt{=}B_j\,[\beta := S]$.

Die parallele Substitution $A\,[\beta_1 := T_1, \ldots, \beta_n := T_n]$ ist analog definiert. Ist $H \equiv \beta_1\texttt{=}T_1, \ldots, \beta_n\texttt{=}T_n$ eine Liste von Typfeldern, dann schreiben wir die obige Substitution auch abkürzend als $A\,[H]$.

Die Reduktionsregeln sind ebenfalls fast dieselben wie in 2.2:

Definition 4.5 Ein Redex des Kalküls kontrahiert auf folgende Weise:

$$
\begin{array}{ll}
[N\texttt{=}A, L \mid H].N & \to A\,[H] \\
[N\texttt{=}A, L \mid H].N\Leftarrow B & \to [N\texttt{=}B, L] \\
[\sigma N\texttt{=}A, L \mid H].\sigma N] & \to A\,[\$:= [\sigma N\texttt{=}A, L]] \\
[L \mid \beta\texttt{=}T, H].\beta\Leftarrow S & \to [L \mid \beta\texttt{=}S, H]
\end{array}
$$

Außerdem erhält man wie schon in Definition 2.2 auch dann einen Redex, wenn man in einer der obigen Regeln das Objekt durch das Fehlersymbol $\Diamond$ ersetzt. In diesem Fall kontrahiert der Redex zu $\Diamond$.

Der Hauptunterschied zum Grundkalkül ist, daß beim Lesen eines Feldes die Typvariablen durch die aktuellen Werte ersetzt werden und zusätzlich natürlich das Überschreiben von Typfeldern geeignet kontrahiert.

Satz 4.6 Der wie oben erweiterte Kalkül hat immer noch die Church-Rosser Eigenschaft.

Der Beweis dazu verläuft genau analog zum Beweis für den Grundkalkül. Man sieht sofort, daß sich Typsubstitution und die Substitution der Selbst-Referenz gegenseitig nicht beeinflussen. Also kann man die entsprechenden Substitutionslemmata ohne Probleme beweisen. Mit deren Hilfe zeigt man dann die Konfluenz der Reduktion wieder mit der Tait-Methode.

Von den dazu benötigten Lemmas formulieren wir nur eines, da wir es später noch einmal benötigen:

Lemma 4.7 (Vertauschbarkeit der Typsubstitution) Es gilt

$$R\,[\alpha := S]\,[\beta := T] \quad = \quad R\,[\beta := T]\,[\alpha := S\,[\beta := T]\,]$$

Beweis: Eine einfache Induktion über den Aufbau des Typs R. $\square$

4.2 Das Typsystem

Für das Typsystem müssen wir zunächst alle Konzepte aus dem zweiten Kapitel um die Typfelder erweitern. Zuerst führen wir eine zweite Art von Kontexten ein:

Definition 4.8 Ein Kontext Δ für freie Typvariablen besteht aus einer ungeordneten Liste von Typvariablen.

Definition 4.9 Ein Paar von Kontexten $\Delta; \Gamma$ heißt *wohlgeformt*, wenn Γ wohlgeformt ist im Sinne von Definition 2.20, wobei jede Deklaration in Γ aber auch freie Typvariable aus Δ enthalten kann. Außerdem darf keine der Typvariablen aus Δ in Γ deklariert werden.

Die *freien Variablen* $\mathrm{FV}(\Delta; \Gamma)$ sind gegeben durch $\mathrm{FV}(\Delta) \cup \mathrm{FV}(\Gamma)$. Für wohlgeformte Kontexte, auf die wir uns wieder ab sofort beschränken wollen, stimmt das mit der intuitiven Begriffsbildung überein: Es sind alle Typvariablen, die in den Deklarationen von Γ frei vorkommen.

Definition 4.10 Ist $\beta; \Gamma$ wohlgeformt, dann wollen wir die Substitution in allen Deklarationen eines Kontextes Γ durch $\Gamma\,[\beta := S]$ notieren. Formal gilt:

$$\Box\,[\beta := S] = \Box \qquad \text{und } (\Gamma, \alpha : T)\,[\beta := S] = (\Gamma\,[\beta := S], \alpha : T\,[\beta := S])$$

Für die Typregeln gehen wir zunächst wieder vom Grundkalkül aus, da man so die wesentlichen Unterschiede klarer sieht. Wir versuchen es zunächst mit den Regeln in den Abbildungen 4.1, 4.2 und 4.3. Dabei sind die Termtypregeln am interessantesten:

Bei den Lese- und Schreibregeln wird der Typ des betroffenen Feldes immer mit den aktuellen Inhalten der bindenden Typfelder substituiert. Daher `type var`

$$\frac{}{\Delta;\Gamma,\alpha:T,\Gamma'\vdash\alpha\ \mathsf{type}}$$

`type free`

$$\frac{}{\Delta,\beta;\Gamma\vdash\beta\ \mathsf{type}}$$

`type obj`

$$\frac{\Delta;\Gamma\vdash S_k\ \mathsf{type}\qquad \Delta,\beta_k;\Gamma,\vdash T_i\ \mathsf{type}}{\Delta;\Gamma\vdash\{N_i:T_i,\sigma M_j\mid\beta_k:S_k\}\ \mathsf{type}}$$

Abbildung 4.1: Der polymorphe Kalkül: Typwohlgeformtheit

`sub refl`

$$\frac{\Delta;\Gamma\vdash S\ \mathsf{type}}{\Delta;\Gamma\vdash S\leq S}$$

`sub trans`

$$\frac{\Delta;\Gamma\vdash S\leq R\qquad \Delta;\Gamma\vdash R\leq T}{\Delta;\Gamma\vdash S\leq T}$$

`sub var`

$$\frac{}{\Delta;\Gamma,\alpha:T,\Gamma'\vdash\alpha\leq T}$$

`sub obj`

$$\frac{\Delta,\beta_l;\Gamma\vdash S'_i\leq S_i\qquad \Delta,\beta_l;\Gamma\vdash T_j\leq T'_j}{\begin{aligned}\Delta;\Gamma\vdash\{&N_i^-:S_i,N_j^+:T_j,\sigma M_k\mid\beta_l:R_l\}\\ &\leq\{N_i^-:S'_i,N_j^+:T'_j,\sigma M_k\mid\beta_l:R_l\}\end{aligned}}$$

`sub ext`

$$\frac{\Delta;\Gamma\vdash\{F'\mid H\}\ \mathsf{type}}{\Delta;\Gamma\vdash\{F,F'\mid H\}\leq\{F\mid H\}}$$

Abbildung 4.2: Der polymorphe Kalkül: Untertypen

`val subs`

$$\frac{\Delta;\Gamma\vdash A:S \qquad \Delta;\Gamma\vdash S\leq T}{\Delta;\Gamma\vdash A:T}$$

`val self`

$$\frac{}{\Delta;\Gamma,\alpha:S\vdash \$:\alpha}$$

`val read`

$$\frac{\Delta;\Gamma\vdash A:S \qquad S=\{F\mid H\} \qquad S.N\,[H]=T}{\Delta;\Gamma\vdash A.N:T}$$

`val write`

$$\frac{\Delta;\Gamma\vdash B:T \qquad \Delta;\Gamma\vdash A:S \qquad S=\{F\mid H\} \qquad S.N\,[H]=T}{\Delta;\Gamma\vdash A.N\Leftarrow B:S}$$

`val read'`

$$\frac{\Delta;\Gamma,\alpha:S\vdash A:\alpha \qquad S=\{F\mid H\} \qquad S.N\,[H]=T}{\Delta;\Gamma,\alpha:S\vdash A.{\downarrow}N:T}$$

`val write'`

$$\frac{\Delta;\Gamma,\alpha:S\vdash B:T \qquad \Delta;\Gamma,\alpha:S\vdash A:\alpha \qquad S=\{F\mid H\} \qquad S.N\,[H]=T}{\Delta;\Gamma,\alpha:S\vdash A.{\downarrow}N\Leftarrow B:\alpha}$$

`val exec`

$$\frac{\Delta;\Gamma\vdash A:S \qquad \sigma M\in S}{\Delta;\Gamma\vdash A.\sigma M:S}$$

`val exec'`

$$\frac{\Delta;\Gamma,\alpha:S\vdash A:\alpha \qquad \sigma M\in S}{\Delta;\Gamma,\alpha:S\vdash A.\sigma M:\alpha}$$

`val obj`

$$\frac{\begin{array}{ll}\Delta,\beta'_k;\Gamma,\alpha:S'\vdash B_j:\alpha & \alpha,\beta'_k\notin \mathrm{FV}(\Delta;\Gamma)\\ \Delta;\Gamma\vdash R_k\ \mathsf{type} & S=\{N_i:T_i,\sigma M_j\mid \beta_k:R_k\}\\ \Delta;\Gamma\vdash A_i:T_i\,[\beta_k:=R_k] & S'=\{N_i:T_i,\sigma M_j\mid \beta_k:\beta'_k\}\end{array}}{\Delta;\Gamma\vdash [N_i\texttt{=}A_i,\sigma M_j\texttt{=}B_j\mid \beta_k\texttt{=}R_k]:S}$$

`val type`

$$\frac{\Delta;\Gamma\vdash T\ \mathsf{type} \qquad \Delta;\Gamma\vdash A:\{F\mid \beta:T',H\}}{\Delta;\Gamma\vdash A.\beta\Leftarrow T:\{F\mid \beta:T,H\}}$$

Abbildung 4.3: Der polymorphe Kalkül: Termtypen (vorläufig) sehen Zugriffe von außen immer den gerade aktuellen Typ. Obwohl dies bei den einfach gestrichenen Regeln formal sehr ähnlich aussieht, ist die Situation hier doch etwas anders: Die Kontexte wurden in der Regel `val obj` so vorbereitet, daß die aktuellen Feldinhalte freie Typvariablen sind. Es kann also innerhalb eines Methodenrumpfes keine konkrete Annahme über den Inhalt der bindenden Typfelder gemacht werden.

Allerdings gibt es mit diesen Typregeln noch eine Schwierigkeit: Man wird es nicht schaffen, für die Regel `val type` in dieser Form die Typerhaltung nachzuweisen. Denn durch diese Regel kann sich der Typ der Felder eines Objektes verändern, ohne daß der Inhalt der Felder dem neuen Typ entsprechen muß. Um diesen Mangel zu beheben, verwenden wir den Typkalkül mit Initialisierung und ändern die Regel so ab, daß die betroffenen Felder nach dem Überschreiben des Typfeldes als uninitialisiert gewertet werden.

Damit können wir den Typkalkül endgültig angeben:

Definition 4.11 Der polymorphe Typkalkül entsteht, indem man den Kalkül mit Initialisierung aus dem dritten Kapitel mit den oben angebenen Regeln verschmilzt. Insbesondere wird die Regel `val type` modifiziert zu

`val type`

$$\frac{\Delta;\Gamma \vdash T\ \mathsf{type} \qquad \Delta;\Gamma \vdash A : \Lambda N_i.\{F \mid \beta : T', H\}}{\Delta;\Gamma \vdash A.\beta \Leftarrow T : \Lambda N_i.\{F \mid \beta : T, H\}}$$

wobei die N_i alle diejenigen Feldnamen sind, in deren Typ die Variable β frei vorkommt.

Der Kalkül in seiner endgültigen Form ist recht komplex. Zur Referenz ist er im Anhang vollständig angegeben.

Für die Typerhaltung müssen wir wieder zuerst die Substitutionslemmas formulieren.

Lemma 4.12 (Substitutionslemma für Typen)

(i) Ist S ein Typ in einem bestimmten Kontext, gilt also $\Delta;\Gamma,\Gamma' \vdash S\ \mathsf{type}$, dann läßt sich aus der Herleitung $\Delta;\Gamma,\alpha : S,\Gamma' \vdash T\ \mathsf{type}$ folgern, daß $\Delta;\Gamma,\Gamma' \vdash T\,[\alpha := S]\ \mathsf{type}$.

(ii) Ist wieder $\Delta;\Gamma \vdash S\ \mathsf{type}$, dann läßt sich aus $\Delta,\alpha;\Gamma \vdash T\ \mathsf{type}$ folgern, daß $\Delta;\Gamma \vdash T\,[\alpha := S]\ \mathsf{type}$.

Beweis: Teil (i) wird genau wie in Lemma 2.27 bewiesen. Die Erweiterung des Kontextes Δ in `type obj` kann direkt durch Verwenden der Induktionsannahme behandelt werden. Teil (ii) zeigt man dann ganz analog. $\square$

Lemma 4.13 (Substitutionslemma für Untertypen) Es sei S ein Typ in einem bestimmten Kontext, also $\Delta; \Gamma \vdash S$ type. Dann gilt

(i) Aus $\Delta; \Gamma, \alpha : S \vdash R \leq T$ folgt $\Delta; \Gamma \vdash R\,[\alpha := S] \leq T\,[\alpha := S]$.

(ii) Läßt sich das Urteil $\Delta, \alpha; \Gamma, \Gamma' \vdash R \leq T$ herleiten, dann folgt auch $\Delta; \Gamma, \Gamma'\,[\alpha := S] \vdash R\,[\alpha := S] \leq T\,[\alpha := S]$.

Beweis: Auch hier läuft der Beweis für Teil (i) genau wie bei Lemma 2.28. Teil (ii) wird wieder ganz analog gezeigt. Bei `sub var` muß zusätzlich berücksichtigt werden, daß T in Γ' liegen muß, falls T die zu substituierende Typvariable α enthält und deshalb die Substitution auf beiden Seiten durchgeführt wird. $\square$

Lemma 4.14 (Erstes Substitutionslemma für Termtypen) Es gelte die Voraussetzung $\Delta; \Gamma \Vdash A :: S$. Dann gilt auch

(i) $\quad \Delta; \Gamma, \alpha : S \Vdash B : T \implies \Delta; \Gamma \Vdash B\,[\$:= A] : T\,[\alpha := S]$
(ii) $\quad \Delta; \Gamma, \alpha : S \Vdash B :: T \implies \Delta; \Gamma \Vdash B\,[\$:= A] :: T\,[\alpha := S]$

Beweis: Man induziert wie in Lemma 2.29 über die Herleitung von $B : T$ beziehungsweise über die Herleitung von $B :: T$. Wir diskutieren der Übersichtlichkeit halber die Fälle im Kalkül ohne Initialisierung; für den kompletten Kalkül muß man die Argumentation aus Lemma 3.4 noch hinzunehmen. Zur Abkürzung sei $\Gamma' = \Gamma, \alpha : S'$ und $\Delta' = \Delta, \beta_k$. Wir beschreiben zunächst den Fall `val read` ausführlich:

Es sei $B \equiv C.N^+$ und es gelte $\Gamma' \Vdash C : R$ sowie $R.N^+\,[H] = T$ für $S = \{F \mid H\}$. Durch Induktion erhält man dann $\Gamma' \Vdash C\,[\$:= A] : R\,[\alpha := S]$. Aufgrund von Lemma 4.7 und der Nebenbedingung der Substitution ist $R\,[\alpha := S]\,.N^+\,[H] = R.N^+\,[H]\,[\alpha := S]$, da die bindenden Typvariablen aus H in S nicht frei sind. Also bekommt man durch Anwenden der Regel `val read` eine Herleitung für $\Gamma' \Vdash B\,[\$:= A] : T\,[\alpha := S]$.

Für die Fälle `val read'`, `val read''`, `val write'`, `val write'` und `val write''` kombiniert man die obige Argumentation zur Substitution mit der Behandlung des Falles in Lemma 2.29. Die Fälle `val subs`, `val self` und alle `val exec`-Fälle kann man direkt aus Lemma 2.29 übernehmen. Auch den Fall `val obj` kann man direkt übernehmen, da die Zusatzannahmen über die Form des Typs S den Induktionsschritt nicht verändern. Den Fall `val type` betrachten wir wieder ausführlich (der Fall `val type''` funktioniert ganz analog):

Es sei $B \equiv C.\gamma \Leftarrow R$ und $T \equiv \{F \mid \beta : R, H\}$. Nach Prämisse der Regel gibt es eine Herleitung für $\Delta; \Gamma' \Vdash C : T'$ mit $T' \equiv \{F \mid \beta : R', H\}$. Durch Induktion erhält man $\Delta; \Gamma \Vdash C\,[\$:= A] : T'\,[\alpha := S]$. Aus $\Delta; \Gamma' \vdash R$ type folgt nach Lemma 4.12 $\Delta; \Gamma \vdash R\,[\alpha := S]$ type. Anwenden der Regel `val obj` liefert dann $\Delta; \Gamma \Vdash B\,[\$:= A] : T\,[\alpha := S]$. □

Lemma 4.15 (Zweites Substitutionslemma für Termtypen) Es gelte allgemein $\Delta; \Gamma \Vdash A :: S$. Dann gilt:

(i) Aus der Voraussetzung $\Delta, \alpha; \Gamma, \Gamma' \Vdash B : T$ läßt sich die Aussage $\Delta; \Gamma, \Gamma'\,[\alpha := S] \Vdash B\,[\alpha := S] : T\,[\alpha := S]$ herleiten.

(ii) Aus der Voraussetzung $\Delta, \alpha; \Gamma, \Gamma' \Vdash B :: T$ läßt sich die Aussage $\Delta; \Gamma, \Gamma'\,[\alpha := S] \Vdash B\,[\alpha := S] :: T\,[\alpha := S]$ herleiten.

Beweis: Wir induzieren über die Herleitung von $B : T$ beziehungsweise $B :: T$. Die Induktion verläuft für alle Fälle nach dem gleichen Schema: Enthalten die Prämissen `sub`- oder `typ`-Urteile, dann können wir Lemma 4.12 beziehungsweise Lemma 4.13 andwenden. Auf die anderen Prämissen läßt sich direkt die Induktionsannahme anwenden; und durch Anwenden der Regel des untersuchten Falles erhält man die gewünschte Konklusion. Bei der Regel `val obj` muß man etwas genauer hinsehen:

Sei $B \equiv [N_i : C_i, \sigma M_j : D_j \mid \beta_k\texttt{=}R_k]$. Bis auf die Prämisse, die den Methodenrumpf überprüft, also konkret $\Delta, \alpha, \beta_k; \Gamma, \Gamma', \gamma : T' \Vdash D_j : \gamma$ kann man wieder das obige Schema verwenden. Aus dieser Prämisse folgt durch Induktion jedoch $\Delta, \beta_k; \Gamma, \Gamma'\,[\alpha := S], \gamma : T'\,[\alpha := S] \Vdash D_j\,[\alpha := S] : \gamma$. Da aber in der gewünschten Konklusion auch T' substituiert wird, paßt wieder alles zusammen, und durch Anwenden der Regel `val obj` ist dieser Fall auch gezeigt. □

Das Lemma 2.30 für die Objekttypen überträgt sich wörtlich, und wir können nun endlich die Typerhaltung zeigen:

Satz 4.16 (Typerhaltung) Es gelte $A \to A'$. Dann folgt:

(i) $\Gamma \Vdash A : S \Longrightarrow \Gamma \Vdash A' : S$
(ii) $\Gamma \Vdash A :: S \Longrightarrow \Gamma \Vdash A' :: S$

Beweis: Der Beweis geht fast genauso wie in Satz 2.31. Zusätzlich muß noch folgendes beachtet werden:

- Bei den Leseregeln paßt die Kontraktion des Redex mit der in der Regel vorgenommenen Substitution zusammen. Lemma 4.13 erlaubt es, auch bei den nötigen Abschwächungen die Substitution mit zu berücksichtigen.

- Bei den Schreibregeln wiederum stimmt die Substitution mit der Prämisse in `val obj` beziehungweise `val init` überein. Also gibt es auch hier keine Schwierigkeiten.

- Für die Ausführungsregel muß die Prämisse aus `val obj` für den Methodenrumpf benutzt werden. Dazu wendet man erst das zweite Substitutionslemma 4.15 an, um die β'_k durch die jetzt konkret gegebenen Inhalte der Typfelder zu ersetzen. Damit bringt man den Typ S und den Typ S' aus der Regel `val obj` in Übereinstimmung, so daß man jetzt weiter wie in Satz 2.31 vorgehen und das erste Substitutionslemma anwenden kann.

- Für die Regeln `val type` und `val type''` schließlich benutzt man, daß für alle Felder, deren Typen die Variable β frei enthalten, ein ΛN in der Konklusion der Regel hinzukommt. Daher muß für diese Felder die `val init`-Regel nicht angewandt werden. Alle anderen Vorkommen der `val init`-Regel, die man aus dem Lemma für Objekttypen erhält, können bedenkenlos weiter verwendet werden. Also kann man auch hier die gewünschte Herleitung konstruieren. $\square$

Kapitel 5

Vollständigkeit

In diesem Kapitel wollen wir untersuchen, inwieweit jetzt Funktionen in dem Kalkül als typbare Objekte darstellbar sind. Dazu wollen wir zunächst einige Beispiele für solche Darstellungen untersuchen. Schließlich werden wir eine Übersetzung des getypten Lambda-Kalküls angeben. Damit haben wir dann gleichzeitig nachgewiesen, daß unser Kalkül vollständig im Sinne der Berechenbarkeit von Funktionen ist.

5.1 Funktionen als Objekte

Zuerst wollen wir die einfachste Funktion, nämlich die Identitätsfunktion, nochmal betrachten.

Beispiel 5.1 Der Term

$$\mathsf{ID} \equiv [\mathsf{arg}^-{:}\alpha\texttt{=}\Diamond, \mathsf{res}^+{:}\alpha\texttt{=}\Diamond, \sigma\mathsf{id}\texttt{=}\$.{\downarrow}\mathsf{res}^+{\Leftarrow}(\$.{\downarrow}\mathsf{arg}^-) \mid \alpha\texttt{=}\{\}]$$

stellt die polymorphe Identitätsfunktion dar. Sie hat den Typ

$$\Lambda\mathsf{arg}^-.\Lambda\mathsf{res}^+.\{\mathsf{arg}^-{:}\alpha, \mathsf{res}^+{:}\alpha, \sigma\mathsf{id}(\mathsf{arg}^-{\rightarrow}\mathsf{res}^+) \mid \alpha\texttt{=}\{\}\}$$

Wir wollen ID wie im letzten Kapitel auf das Objekt $\mathsf{X}\texttt{=}[\mathsf{x} : []]$ anwenden und dabei genau verfolgen, wie die Typen während der Reduktion aussehen.

X hat den Typ $T = \{\mathsf{x} : \{\}\}$. Der Term, der ID auf X anwndet, lautet

$$A_1 \equiv \mathsf{ID}.\alpha{\Leftarrow}T.\sigma\mathsf{id}(\mathsf{arg}^-{\Leftarrow}\mathsf{X}).\mathsf{res}^+$$

Er hat wieder den Typ T. Betrachten wir jetzt die Reduktion. Zunächst reduziert A_1 zu $A_2 \equiv B_2.\sigma\mathsf{id}(\mathsf{arg}^-{\Leftarrow}\mathsf{X}).\mathsf{res}^+$ mit

$$B_2 \equiv [\mathsf{arg}^-{:}\alpha\texttt{=}\Diamond, \mathsf{res}^+{:}\alpha\texttt{=}\Diamond, \sigma\mathsf{id}\texttt{=}\$.{\downarrow}\mathsf{res}^+{\Leftarrow}(\$.{\downarrow}\mathsf{arg}^-) \mid \alpha\texttt{=}T].$$

Dabei erhält B_2 den Typ

$$\vdash B_2 : \Lambda\mathsf{arg}^-.\Lambda\mathsf{res}^+.\{\mathsf{arg}^-{:}\alpha, \mathsf{res}^+{:}\alpha, \sigma\mathsf{id}(\mathsf{arg}^-{\rightarrow}\mathsf{res}^+) \mid \alpha{=}\mathsf{T}\}$$

Wir lösen die Abkürzung $B.\sigma\mathsf{id}(\mathsf{arg}^-{\twoheadleftarrow}\mathsf{X})$ zu $B.arg^-{\twoheadleftarrow}\mathsf{X}.\sigma\mathsf{id}$ auf. Dann reduziert A_2 zu $A_3 \equiv B_3.\sigma\mathsf{id}.\mathsf{res}^+$ mit

$$B_3 \equiv [\mathsf{arg}^-{:}\alpha{=}\mathsf{X}, \mathsf{res}^+{:}\alpha{=}\Diamond, \sigma\mathsf{id}{=}\$.{\downarrow}\mathsf{res}^+{\twoheadleftarrow}(\$.{\downarrow}\mathsf{arg}^-) \mid \alpha{=}T]$$

Für B_3 bestimmen wir den Typ

$$\vdash B_3 : \Lambda\mathsf{res}^+.\{\mathsf{arg}^-{:}\alpha, \mathsf{res}^+{:}\alpha, \sigma\mathsf{id}(\mathsf{arg}^-{\rightarrow}\mathsf{res}^+) \mid \alpha{=}\mathsf{T}\}$$

Wird nun die Methode in A_3 ausgeführt, dann reduziert A_3 weiter zu

$$A_4 \equiv B_3.\mathsf{res}^+.{\twoheadleftarrow}(B_3.\mathsf{arg}^-).\mathsf{res}^+$$

Spätestens hier müssen wir jetzt zum Reduktionskalkül übergehen, um diesen Term noch typisieren zu können. Er erhält dann natürlich den Typ $\Vdash A_4 : T$. Wir reduzieren weiter zu

$$A_5 \equiv B_3.\mathsf{res}^+.{\twoheadleftarrow}(\mathsf{X}).\mathsf{res}^+$$

und schließlich zu $A_6 \equiv B_6.\mathsf{res}^+$ mit

$$B_6 \equiv [\mathsf{arg}^-{:}\alpha{=}\mathsf{X}, \mathsf{res}^+{:}\alpha{=}\mathsf{X}, \sigma\mathsf{id}{=}\$.{\downarrow}\mathsf{res}^+{\twoheadleftarrow}(\$.{\downarrow}\mathsf{arg}^-) \mid \alpha{=}T]$$

Dabei hat B_6 den Typ

$$\Vdash B_6 :: \{\mathsf{arg}^-{:}\alpha, \mathsf{res}^+{:}\alpha, \sigma\mathsf{id}(\mathsf{arg}^-{\rightarrow}\mathsf{res}^+) \mid \alpha{=}\mathsf{T}\}$$

Und zum Schluß reduziert A_6 weiter zu X mit dem Typ T.

Analog zur Identität kann man Projektionsfunktionen für Tupel definieren: Man muß nur mehrere Argumente $\mathsf{arg}_1^-, \ldots, \mathsf{arg}_n^-$ vorsehen. Dieses Beispiel ist natürlich eher trivial, da Objekte von sich aus schon Tupel darstellen. Interessanter ist da schon der Rekursionsoperator:

Beispiel 5.2 Die polymorphe Rekursionsfunktion Y verarbeitet eine Funktion mit einem Argumentfeld arg^- und einem Resultatsfeld res^+ sowie einer Methode $\sigma\mathsf{f}$. Dabei können Argument- und Resultatsfeld von einem beliebigen Typ α sein.

$$\begin{aligned}&\mathsf{Y}{=}[\mathsf{func}^-{:}\{\mathsf{arg}^-{:}\alpha, \mathsf{res}^+{:}\alpha \mid\}{=}\Diamond, \mathsf{limit}^+{:}\alpha{=}\Diamond,\\ &\quad \sigma\mathsf{rec}{=}\$.{\downarrow}\mathsf{limit}{\twoheadleftarrow}(\$.{\downarrow}\mathsf{func}^-.\sigma\mathsf{f}(\mathsf{arg}^-{\twoheadleftarrow}(\$.\sigma\mathsf{rec}.\mathsf{limit}^+)).\mathsf{res}^+) \mid \alpha{=}\{\mid\}]\end{aligned}$$

Der Term Y hat den Typ

$$\Lambda\mathsf{func}^-.\Lambda\mathsf{limit}^+.\{\mathsf{func}^-{:}\{\mathsf{arg}^-{:}\alpha, \mathsf{res}^+{:}\alpha \mid\}, \mathsf{limit}^+{:}\alpha, \sigma(\mathsf{func}^-{\rightarrow}\mathsf{limit}^+) \mid \alpha{=}\{\mid\}\}$$

var

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}$$

$\to$-elim

$$\frac{\Gamma \vdash A : S{\to}T \qquad \Gamma \vdash B : S}{\Gamma \vdash A\ B : T}$$

$\to$-intro

$$\frac{\Gamma, x : S \vdash A : T}{\Gamma \vdash \lambda x : S.A : S{\to}T}$$

$\forall$-elim

$$\frac{\Gamma \vdash A : \forall\alpha.T}{\Gamma \vdash A\ S : T\,[\alpha := S]}$$

$\forall$-intro

$$\frac{\Gamma \vdash A : T \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \Lambda alpha.A : \forall\alpha.T}$$

Abbildung 5.1: Der polymorphe Lambda-Kalkül

5.2 Übersetzen des Lambda-Kalküls

In diesem Abschnitt wollen wir den polymorphen Lambda-Kalkül in der Church-Version, wie er zum Beispiel in [Bar92] steht, in unseren Objektkalkül übersetzen. Dabei wollen wir sowohl Terme wie Typen übersetzen. Wir passen die Darstellung der Regeln für den Lambda-Kalkül an unsere Buchstabenkonvention an. Die Regeln für den polymorph getypten Lambda-Kalkül sind zur Referenz noch einmal in Abbildung 5.1 angegeben.

Wir beschränken uns zunächst auf den nicht-polymorphen Teil der Kalküle und geben eine Übersetzung der Terme ohne zusätzliche Typinformation an, damit das Prinzip klar wird.

Die Idee bei der Übersetzung ist, einen Lambda-Term durch einen *closure* zu interpretieren, das heißt, für die freien Variablen des Terms Platz zu reservieren und diesen dann zu füllen, sobald eine Applikation vorliegt. Diese Methode wird auch in der Praxis für die Implementierung des Lambda-Kalküls verwendet.

Definition 5.3 Wir bezeichnen mit $\Phi(A)_\Delta$ die Übersetzung des Lambda-Terms A, der die freien Variablen in Δ hat, in den Objektkalkül. Konkret ist Φ durch

$$\begin{array}{ll}
\Phi(x)_{y_1,\ldots,y_n,x} & \equiv \mathsf{\$.arg}^- \\
\Phi(y_i)_{y_1,\ldots,y_n,x} & \equiv \mathsf{\$.y_i} \\
\Phi(A\ B)_{y_1,\ldots,y_n} & \equiv \Phi(A)_{y_1,\ldots,y_n}.\sigma\mathsf{exec}(\mathsf{arg}^- \Leftarrow \Phi(B)_{y_1,\ldots,y_n}).\mathsf{res}^+ \\
\Phi(\lambda x.A) & \equiv [\mathsf{arg}^-{=}\Diamond, \mathsf{res}^+{=}\Diamond, \sigma\mathsf{exec}{=}\mathsf{\$.res}^+ \Leftarrow \Phi(A)_x] \\
\Phi(\lambda x.A)_{y_1,\ldots,y_n,z} & \equiv [\mathsf{arg}^-{=}\Diamond, \mathsf{res}^+{=}\Diamond, \mathsf{y_1}{=}\mathsf{\$.y_1}, \ldots, \mathsf{y_n}{=}\mathsf{\$.y_n}, \mathsf{z}{=}\mathsf{\$.arg}, \\
 & \quad \sigma\mathsf{exec}{=}\mathsf{\$.res}^+ \Leftarrow \Phi(A)_{y_1,\ldots,y_n,z,x}]
\end{array}$$

gegeben.

Will man beweisen, daß die Übersetzung und der Ursprungsterm adäquat zueinander reduzieren, dann muß man zusätzlich eine Rückübersetzung der *closures* angeben, in der die freien Variablen mit ihrer aktuellen Belegung im *closure* subtitutiert werden. Das ist etwas technisch; und da hier unser Hauptaugenmerk auf dem Typsystem liegt, werden wir diesen Beweis hier nicht durchführen.

Jetzt wollen wir zu den Typen höherer Ordnung übergehen. Zunächst geben wir dazu an, wie Typen übersetzt werden, da wir dies für die Übersetzung der Terme benötigen.

Definition 5.4 Die Übersetzung eines Typs T des Lambda-Kalküls wird durch $\Phi(T)$ notiert. Dabei ist

$$\begin{aligned}
\Phi(\alpha) &= \alpha \\
\Phi(S{\to}T) &= \Lambda\mathsf{arg}^-.\Lambda\mathsf{res}^+.\{\mathsf{arg}^-{:}\Phi(S), \mathsf{res}^+{:}\Phi(T) \mid\} \\
\Phi(\forall\alpha.T) &= \{\mathsf{body}{:}\Phi(T) \mid \alpha : \{\mid\}\}
\end{aligned}$$

Nun können wir auch die Terme des polymorphen Lambda-Kalküls übersetzen. Allerdings benötigen wir dazu mehr Typinformation, als uns in den Termen selbst zur Verfügung steht, da auch die Ergebnisfelder im Objektkalkül einen Typ bekommen müssen. Also übersetzen wir gleich das ganze Typurteil $\Gamma \vdash A : T$, das uns die gewünschten Typinformationen liefern kann.

Definition 5.5 Wir bezeichnen mit $\Phi(\Gamma \vdash A : T)$ die Übersetzung des Lambda-Terms A, der durch $\Gamma \vdash A : T$ getypt werden kann. Man beachte, daß wir für den Lambda-Kalkül *geordnete* Kontexte verwenden. Die Übersetzung sieht folgendermaßen aus:

$$\begin{aligned}
&\Phi(\Gamma, x : T \vdash x : T) && \equiv \$.\mathsf{arg}^- \\
&\Phi(\Gamma, y_i : T, \Gamma' \vdash y_i : T) && \equiv \$.\mathsf{y_i} \\
&\Phi(\Gamma \vdash A\ B : T) && \equiv \\
&\quad \Phi(\Gamma \vdash A : S{\to}T).\sigma\mathsf{exec}(\mathsf{arg}^- {\Leftarrow} \Phi(\Gamma \vdash B : S)).\mathsf{res}^+ \\
&\Phi(\vdash \lambda x : S.A : S{\to}T) && \equiv [\mathsf{arg}^-{:}\Phi(S){=}\Diamond, \mathsf{res}^+ : \Phi(T){=}\Diamond, \\
& && \qquad \sigma\mathsf{exec}{=}\$.\mathsf{res}^+ {\Leftarrow} \Phi(x : S \vdash A : T) \mid] \\
&\Phi(\Gamma, z : R \vdash \lambda x : S.A : S{\to}T) && \equiv [\mathsf{arg}^-{:}\Phi(S){=}\Diamond, \mathsf{res}^+ : \Phi(T){=}\Diamond, \\
&\quad \mathsf{y_1}{:}\Phi(R_1){=}\$.\mathsf{y_1}, \ldots, \mathsf{y_n}{:}\Phi(R_n){=}\$.\mathsf{y_n}, \mathsf{z}{:}\Phi(R){=}\$.\mathsf{arg}^-, \\
&\quad \sigma\mathsf{exec}{=}\$.\mathsf{res}^+ {\Leftarrow} \Phi(\Gamma, z : R, x : S \vdash A : T) \mid] \\
&\quad \text{wobei } \Gamma = y_1 : R_1, \ldots, y_n : R_n \\
&\Phi(\Gamma \vdash \Lambda\alpha.A : \forall\alpha.T) && \equiv [\mathsf{body}{:}T{=}\Phi(\Gamma \vdash A : T) \mid \alpha{=}\{\mid\}] \\
&\Phi(\Gamma \vdash A\ S : T) && \equiv \Phi(\Gamma \vdash A : \forall\alpha.T').\alpha {\Leftarrow} \Phi(S).\mathsf{body}
\end{aligned}$$

Satz 5.6 Sei A ein Lambda-Term mit Typ T. Dann ist $\Phi(A)$ im Objektkalkül typbar mit dem Typ $\Phi(T)$.

Beweis: Es ist offensichtlich, daß die Übersetzung der Typherleitung folgt. Man muß also nur noch für jeden Fall prüfen, ob sich eine passende Typherleitung im Objektkalkül finden läßt. Dabei muß noch berücksichtigt werden, daß man durch Abschwächung die Felder für die freien Variablen des Lambda-Terms verschwinden lassen kann. Damit läßt sich dann wirklich jeder Fall überprüfen. □

Der getypte Objektkalkül hat also die gleiche Ausdrucksstärke wie der polymorphe Lambda-Kalkül.

Kapitel 6

Diskussion

Nachdem der Kalkül jetzt vollständig ist, wollen wir kurz einige Beobachtungen zusammenfassen, die man durch ihn gewinnen kann.

Zunächst fällt auf, daß der Kalkül selbst zwar relativ einfach, aber das letztendliche nötige Typsystem schon sehr komplex ist. Das ist einerseits gut, da es zeigt, daß der Ansatz nun wirklich vom funktionalen Denken weiter entfernt ist. Andererseits ist es schlecht, da man komplexe Systeme mathematisch natürlich schlechter in den Griff bekommt. Allerdings ist es auch möglich, daß die Grundlagen des imperativen oder auch des objektorientierten Programmierens eben komplex sind, und es keine einfache elegante Modellierung dafür gibt.

Auch die Art der Übersetzung des Lambda-Kalküls weist darauf hin, daß mit dem Kalkül ungefähr der richtige Weg eingeschlagen wurde. Die natürlichste Übersetzung, die sich anbietet, ist auch in der Praxis weit verbreitet. Das zeigt, daß unser Kalkül offenbar wirklich zumindest einige der wichtigsten Grundlagen gut repräsentiert.

Eine völlig offene Frage ist es, wie ein geeignete Semantik für den Objektkalkül aussehen könnte. Die üblichen mathematischen Strukturen, auf die man normalerweise zurückgreifen würde, sind vermutlich ein Schritt in die falsche Richtung, da mit ihnen eher das funktionale Paradigma gut modelliert wird. Auch für die Typen wäre es schön, wenn man von den syntaktischen Strukturen wegkommen könnte und die gemischt-varianten Tupel statt dessen in einem Modell besser verstehen könnte. Dies würde vielleicht auch einen Hinweis darauf liefern, wie man einige der nun zu erwähnenden Defekte ausbessern könnte.

6.1 Mögliche Ergänzungen

In dem Kalkül gibt es Terme, für die man keinen Typ finden kann: Hat ein Objekte zum Beispiel zwei Felder x und y und die beiden Methoden σf=\$.x$\Leftarrow$(\$.y.a) und σg=\$.y$\Leftarrow$(\$.x.a), dann müßten sich die beiden Felder gegenseitig referieren, was in dem Typsystem nicht möglich ist.

Als Lösung kann man rekursive Typen einführen. Dabei muß man sich allerdings überlgen, wie sie mit den Untertypen sowie der Ko- und Kontravarianz richtig zusammenspielen sollen.

Um Funktionen darzustellen, haben wir für eine Methode besondere Ein- und Ausgabefelder reserviert. Genauso könnte man sich vorstellen, auch Felder nur für den Gebrauch innerhalb der Methode zu reservieren. Auf diese Art hätte man *lokale Variablen* für Methoden.

Die vielleicht interessanteste Erweiterung schließlich wäre es, den Kalkül mit einer imperativen Semantik auszustatten. Dazu müßte man sich die Objekte als persistent vorstellen. Felder verweisen dann auf andere Objekte, und man müßte eine *clone*-Operation oder ähnliches hinzunehmen, die neue Objekte erzeugt.

Bei dieser Art von Semantik geht die Church-Rosser Eigenschaft sofort verloren, da jetzt Seiteneffekte auftreten können. Man müßte dann also die Reduktionsreihenfolge festlegen. Auch bei der Polymorphie gibt es Probleme: Ein Objekt, daß von mehreren Feldern aus referiert wird, darf nicht einfach mit neuen Typfeldinhalten überschrieben werden, da ansonsten nicht gewährleistet ist, daß die referierenden Felder noch auf ein Objekt mit dem korrekten Typ zeigen. Die nicht-polymorphe Version des Typkalküls aber kann man auch mit der imperativen Semantik unverändert verwenden, wie man sich leicht überlegen kann.

Es gibt auch noch einige offene theoretische Fragen bezüglich des Typkalküls. So ist nicht geklärt, ob das *Typisierbarkeitsproblem* hier entscheidbar ist. Auch das *Typherleitbarkeitsproblem* ist noch offen. Man kann sich leicht einen Algorithmus überlegen, der in vielen Fällen auch den gewünschten Typ liefert. Schaut man sich diesen Algorithmus genauer an, so erkennt man, daß es Fälle gibt, wo der Algorithmus keinen minimalen Typ finden kann. Dies deutet darauf hin, daß das polymorphe Typsystem noch nicht allgemein genug ist.

6.2 Zusammenfassung

Was hat man jetzt durch diesen Kalkül gelernt? Der Kalkül zeigt, daß es ist möglich ist, die Grundlagen der objekt-orientierten und teilweise auch der imperativen Programmierung direkt zu modellieren, ohne den Umweg über funktionale Modelle zu gehen. Dies ist sogar mit der vollen Ko- und Kontravarianz und einer eingeschränkten Selbst-Referenz möglich.

Gleichzeitig hat man den Zusammenhang zwischen funktionalem und imperativen Programmieren sichtbar gemacht. Es wird nicht nur klar, auf welche Weise einen gegenseitige Simulation möglich ist, man kann auch genau sehen, wie sich die Wirkung einer Funktion in der imperativem getypten Sichtweise niederschlägt: Der ursprüngliche atomare Vorgang der Applikation wird dreigeteilt, mit der Konsequenz, daß das Typsystem dem Rechnung tragen muß.

Auch der Begriff der Polymorphie ist genauer beleuchtet worden: Zum einen schlägt sich die Polymorphie in den Untertypen nieder, zum anderen wird sie als parametrische Polymorphie ein Teil des Zustandes. Dies bietet vielleicht einen Ansatzpunkt, um auch Typsysteme höherer Ordnung für imperative Sprachen besser in den Griff zu bekommen.

Zusammenfassend kann man also sagen, daß — auch wenn unser Kalkül noch nicht eine endgültig zufriedenstellende Lösung ist — er doch zumindest einige Einblicke in die Zusammenhänge und Grundlagen der verschiedenen Programmierparadigma vermittelt.

Literaturverzeichnis

[AC94] Martin Abadi and Luca Cardelli. A theory of primitive objects: untyped and first order systems. In *Proc. Theor. Aspects of Computer Software*, pages 296–320, Japan, 1994. Expanded version to appear in *Information and Computation*.

[AC96] Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Verlag, 1996.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 2nd edition, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–309. Clarendon Press, 1992.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4:127–206, April 1994.

[GM94] C. Gunter and J. Mitchel, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[MMS95] Tom Mens, Kim Mens, and Patrick Steyaert. OPUS: a calculus for modelling object-oriented concepts. In D. Patel, Y. Sun, and S. Patel, editors, *Proceedings of the 1994 International Conference on Object Oriented Information Systems*, pages 152–165. Springer-Verlag, 1995.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4:207–247, April 1994.

[QKB96] Zhenyu Qian and Bernd Krieg-Brckner. Typed object-oriented programming with late binding. In *Proc. 10th European Conf. on Object-Oriented Programming*, LNCS, pages 48–72. Springer-Verlag, 1996.

[Tak92] Masako Takahashi. Parallel reductions in λ-calculus. Technical report, Tokyo Institute of Technology, 1992.

Anhang A

Übersicht über alle Typregeln

In diesem Anhang sind noch einmal die vollständigen Typregeln für den polymorphen Typkalkül mit Initialisierung zusammengestellt.

Regeln für Typwohlgeformtheit:

`type var`
$$\frac{}{\Delta;\Gamma,\alpha:T,\Gamma'\vdash\alpha\ \mathsf{type}}$$

`type free`
$$\frac{}{\Delta,\beta;\Gamma\vdash\beta\ \mathsf{type}}$$

`type init`
$$\frac{\Delta;\Gamma\vdash T\ \mathsf{type}}{\Delta;\Gamma\vdash\Lambda N.T\ \mathsf{type}}$$

`type obj`
$$\frac{\Gamma\vdash S_k\ \mathsf{type}\qquad\Gamma,\beta_k:\{|\}\vdash T_i\ \mathsf{type}}{\Gamma\vdash\{N_i:T_i,\sigma M_j\mid\beta_k:S_k\}\ \mathsf{type}}$$

Regeln für Untertypen:

`sub refl`
$$\frac{\Delta;\Gamma\vdash S\ \mathsf{type}}{\Delta;\Gamma\vdash S\leq S}$$

`sub trans`
$$\frac{\Delta;\Gamma\vdash S\leq R\qquad\Delta;\Gamma\vdash R\leq T}{\Delta;\Gamma\vdash S\leq T}$$

`sub init right`
$$\frac{\Delta;\Gamma\vdash S\ \mathsf{type}}{\Delta;\Gamma\vdash S\leq\Lambda N.S}$$

`sub var`
$$\frac{}{\Delta;\Gamma,\alpha:T,\Gamma'\vdash\alpha\leq T}$$

`sub init both`
$$\frac{\Delta;\Gamma\vdash S\leq T}{\Delta;\Gamma\vdash\Lambda N.S\leq\Lambda N.T}$$

`sub obj`
$$\frac{\Delta,\beta_l;\Gamma\vdash S'_i\leq S_i\qquad\Delta,\beta_l;\Gamma\vdash T_j\leq T'_j}{\begin{array}{l}\Delta;\Gamma\vdash\{N_i^-:S_i,N_j^+:T_j,\sigma M_k\mid\beta_l:R_l\}\\\qquad\leq\{N_i^-:S'_i,N_j^+:T'_j,\sigma M_k\mid\beta_l:R_l\}\end{array}}$$

`sub ext`
$$\frac{\Delta;\Gamma\vdash\{F'\mid H\}\ \mathsf{type}}{\Delta;\Gamma\vdash\{F,F'\mid H\}\leq\{F\mid H\}}$$

Regeln für das Typisieren konsistenter Terme:

`val subs`

$$\frac{\Delta;\Gamma \vdash A : S \qquad \Delta;\Gamma \vdash S \leq T}{\Delta;\Gamma \vdash A : T}$$

`val self`

$$\frac{}{\Delta;\Gamma, \alpha : \Lambda N_i.S \vdash \$: \Lambda N_i.\alpha}$$

`val write`

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash B : T \\ \Delta;\Gamma \vdash A : \Lambda N.\Lambda N'_i.S \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma \vdash A.N \leftarrow B : \Lambda N'_i.S}$$

`val read`

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash A : \Lambda N_i.S \qquad N \neq N_i \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma \vdash A.N : T}$$

`val read'`

$$\frac{\begin{array}{l}\Delta;\Gamma, \alpha : \Lambda N_i.S \vdash A : \Lambda N_i.\alpha \\ N \neq N_i \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma, \alpha : S \vdash A.{\downarrow}N : T}$$

`val write'`

$$\frac{\begin{array}{l}\Delta;\Gamma, \alpha : \Lambda N_i.S \vdash B : T \\ \Delta;\Gamma, \alpha : \Lambda N_i.S \vdash A : \Lambda N.\Lambda N'_j.\alpha \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma, \alpha : \Lambda N_i.S \vdash A.{\downarrow}N \leftarrow B : \Lambda N'_j.\alpha}$$

`val exec`

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash A : \Lambda N_i.S \\ \sigma M(N_i) \in S\end{array}}{\Delta;\Gamma \vdash A.\sigma M : S}$$

`val exec'`

$$\frac{\begin{array}{l}\Delta;\Gamma, \alpha : \Lambda N_i.S \vdash A : \Lambda N'_j.\alpha \\ \sigma M(N'_j) \in S\end{array}}{\Delta;\Gamma, \alpha : \Lambda N_i.S \vdash A.\sigma M : \alpha}$$

`val obj`

$$\frac{\begin{array}{ll}\Delta, \beta'_k;\Gamma, \alpha : \Lambda_{jl}.S' \vdash B_j : \alpha & \alpha, \beta'_k \notin \mathrm{FV}(\Delta;\Gamma) \\ \Delta;\Gamma \vdash R_k \ \mathsf{type} & S = \{N_i : T_i, \sigma M_j(N_{jl}) \mid \beta_k : R_k\} \\ \Delta;\Gamma \vdash T_i\,[\beta_k := R_k] \ \mathsf{type} & S' = \{N_i : T_i, \sigma M_j(N_{jl}) \mid \beta_k : \beta'_k\}\end{array}}{\Delta;\Gamma \vdash [N_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j \mid \beta_k\texttt{=}R_k] : S}$$

`val type`

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash T \ \mathsf{type} \\ \Delta;\Gamma \vdash A : \Lambda N_i.\{F \mid \beta : T', H\}\end{array}}{\Delta;\Gamma \vdash A.\beta \leftarrow T : \Lambda N_i.\{F \mid \beta : T, H\}}$$

`val init`

$$\frac{\begin{array}{l}\Delta;\Gamma \Vdash A : T\,[H] \\ \Delta;\Gamma \Vdash [N : T\texttt{=}A, L \mid H] : \Lambda N.\Lambda N_i.S\end{array}}{\Delta;\Gamma \Vdash [N : T\texttt{=}A, L \mid H] : \Lambda N_i.S}$$

Regeln für das Typisieren von Termen auch während der Reduktion:

`val subs`

$$\frac{\Delta;\Gamma \Vdash A : S \qquad \Delta;\Gamma \vdash S \leq T}{\Delta;\Gamma \Vdash A : T}$$

`val read`

$$\frac{\begin{array}{l}\Delta;\Gamma \Vdash A : \Lambda N_i.S \qquad N^+ \neq N_i \\ S = \{F \mid H\} \qquad S.N^+\,[H] = T\end{array}}{\Delta;\Gamma \Vdash A.N^+ : T}$$

`val read'`

$$\frac{\begin{array}{l}\Delta;\Gamma,\alpha : \Lambda N_i.S \Vdash A :: \Lambda N_i.\alpha \\ N \neq N_i \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma,\alpha : S \Vdash A.{\downarrow}N : T}$$

`val read''`

$$\frac{\begin{array}{l}\Delta;\Gamma \Vdash A :: \Lambda N_i.S \qquad N \neq N_i \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma \Vdash A.N^+ : T}$$

`val exec`

$$\frac{\begin{array}{l}\Delta;\Gamma \Vdash A : \Lambda N_i.S \\ \sigma M(N_i) \in S\end{array}}{\Delta;\Gamma \Vdash A.\sigma M : S}$$

`val exec'`

$$\frac{\begin{array}{l}\Delta;\Gamma,\alpha :: \Lambda N_i.S \Vdash A : \Lambda N'_j.\alpha \\ \sigma M(N'_j) \in S\end{array}}{\Delta;\Gamma,\alpha :: \Lambda N_i.S \Vdash A.\sigma M : \alpha}$$

`val exec''`

$$\frac{\begin{array}{l}\Delta;\Gamma \Vdash A :: \Lambda N_i.S \\ \sigma M(N_i) \in S\end{array}}{\Delta;\Gamma \Vdash A.\sigma M :: S}$$

`val self`

$$\frac{}{\Delta;\Gamma,\alpha : \Lambda N_i.S \Vdash \$: \Lambda N_i.\alpha}$$

`val write`

$$\frac{\begin{array}{l}\Delta;\Gamma \Vdash B : T \\ \Delta;\Gamma \Vdash A : \Lambda N^-.\Lambda N'_i.S \\ S = \{F \mid H\} \qquad S.N^-\,[H] = T\end{array}}{\Delta;\Gamma \Vdash A.N \Leftarrow B : \Lambda N'_i.S}$$

`val write'`

$$\frac{\begin{array}{l}\Delta;\Gamma,\alpha : \Lambda N_i.S \Vdash B : T \\ \Delta;\Gamma,\alpha : \Lambda N_i.S \Vdash A :: \Lambda N.\Lambda N'_j.\alpha \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma,\alpha : \Lambda N_i.S \Vdash A.{\downarrow}N \Leftarrow B :: \Lambda N'_j.\alpha}$$

`val write''`

$$\frac{\begin{array}{l}\Delta;\Gamma \Vdash B : T \\ \Delta;\Gamma \Vdash A :: \Lambda N.\Lambda N'_i.S \\ S = \{F \mid H\} \qquad S.N\,[H] = T\end{array}}{\Delta;\Gamma \Vdash A.N \Leftarrow B :: \Lambda N'_i.S}$$

`val type`

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash T \ \mathsf{type} \\ \Delta;\Gamma \Vdash A : \Lambda N_i.\{F \mid \beta : T', H\}\end{array}}{\Delta;\Gamma \Vdash A.\beta \Leftarrow T : \Lambda N_i.\{F \mid \beta : T, H\}}$$

`val type''`

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash T \ \mathsf{type} \\ \Delta;\Gamma \Vdash A :: \Lambda N_i.\{F \mid \beta : T', H\}\end{array}}{\Delta;\Gamma \Vdash A.\beta \Leftarrow T :: \Lambda N_i.\{F \mid \beta : T, H\}}$$

`val obj`

$$\frac{\begin{array}{ll}\Delta,\beta'_k;\Gamma,\alpha : \Lambda_{jl}.S' \Vdash B_j :: \alpha & \alpha,\beta'_k \notin \mathrm{FV}(\Delta;\Gamma) \\ \Delta;\Gamma \vdash R_k \ \mathsf{type} & S = \{N_i : T_i, \sigma M_j(N_{jl}) \mid \beta_k : R_k\} \\ \Delta;\Gamma \vdash T_i\,[\beta_k := R_k] \ \mathsf{type} & S' = \{N_i : T_i, \sigma M_j(N_{jl}) \mid \beta_k : \beta'_k\}\end{array}}{\Delta;\Gamma \Vdash [N_i\texttt{=}A_i, \sigma M_j\texttt{=}B_j \mid \beta_k\texttt{=}R_k] :: S}$$

`val init`

$$\frac{\Delta;\Gamma \Vdash A : T\,[H] \qquad \Delta;\Gamma \Vdash [N : T\texttt{=}A, L \mid H] :: \Lambda N.\Lambda N_i.S}{\Delta;\Gamma \Vdash [N : T\texttt{=}A, L \mid H] :: \Lambda N_i.S}$$

`val conv`

$$\frac{\Delta;\Gamma \Vdash A :: S}{\Delta;\Gamma \Vdash A : S}$$

wobei bei den `val typ`-Regeln die N_i genau diejenigen Felder sind, die in ihrem Typ β als freie Typvariable enthalten.

Erklärung:

Hiermit erkläre ich, daß ich diese Arbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe.
Ich bin damit einverstanden, daß der Betreuer meiner Diplomarbeit diese nach Abschluß der Prüfung zu wissenschaftlichen Zwecken weitergeben kann.

Darmstadt, im Juli 1997