Universität München
Centrum für Informations- und
Sprachverarbeitung

Typsysteme funktionaler Programmiersprachen
Teil I

Hans Leiß

CIS-Bericht-97-107

31.Oktober 1997

Verantwortlich:
Petra Maier, Centrum für Informations- und Sprachverarbeitung,
Universität München, Oettingenstr. 67, 80538 München
EMAIL: pmaier@cis.uni-muenchen.de

Typsysteme funktionaler Programmiersprachen
Teil I

Dr. Hans Leiß

Centrum für Informations-
und Sprachverarbeitung
Universität München

Sommersemester 1997

Inhaltsverzeichnis

1 Einführung **5**

1.1 Getypte und ungetypte Programme 5

1.2 Wozu sollen Typen dienen ? 7

1.3 Eigenschaften von Typsystemen 9

2 Der typfreie λ–Kalkül **11**

2.1 Syntax des λ–Kalküls 11

2.2 Die Gleichungstheorien $\lambda\beta$ und $\lambda\beta\eta$ 12

2.3 Die Reduktionstheorien β und $\beta\eta$ 13

2.4 Interpretation von λ–Termen durch Umgebungsmodelle 14

2.5 Ein Modell des typfreien λ-Kalküls 20

2.6 Interpretationen von Typen in λ–Modellen 21

3 Reduktion und Typisierbarkeit **23**

3.1 Typisierungen mit Funktions– und Durchschnittstypen 24

3.2 Abschwächungen von D-Typisierungen 29

3.3 Stark normalisierbare Terme 31

3.4 D–typisierbare Terme 33

3.5 Normalisierbarkeit und E-Typisierbarkeit 36

4 Entscheidbarkeit der C-Typisierbarkeit **39**

4.1 Haupttypen im C–Kalkül 39

4.2 Typsynthese für den C–Kalkül 40

5 Polymorphismus in der Programmiersprache *ML* **45**

5.1 *ML*–Programme und *ML*–Reduktionstheorie 46

5.2 *ML*–Typisierungen 47

5.3 Typsynthese für *ML* 53

6 Typsicherheit von *ML* **57**

6.1 Operationale Semantik 57

6.2 Typsicherheit des funktionalen Kerns von *ML* 58

6.3 Haupttypen und Typsicherheit für *ML* mit Referenzen 63

7 Entscheidungskomplexität der *ML*-Typisierbarkeit **71**

7.1 Entscheidungskomplexität 71

7.2 Die Größe von *ML*–Haupttypen 73

7.3 Programmieren auf der Ebene der Typen 74

7.4 Kodierung monotoner Boole'scher Algebra 76

7.5 Kodierung endlicher Mengen 78

7.6 Kodierung von Turingmaschinen 79

Vorwort

Das vorliegende Skriptum ist der Anfangsteil einer Vorlesung, die 1990 und 1992 im Mathematischen Institut der Universität München gehalten wurde. Der zweite Teil umfaßte unter anderem das Ideal-Modell von *ML*, polymorphe Rekursion, ein Modell für rekursive und dynamische Typen, den zweitstufigen λ-Kalkül mit $\forall$-Typen, abstrakte Datentypen und $\exists$-Typen sowie ein Relationenmodell für Subtypen und beschränkte Typquantoren.

Den hier wiedergegebenen ersten Teil habe ich für eine Vorlesung im Sommersemester 1997 am Centrum für Informations- und Sprachverarbeitung der Universität München ausgearbeitet, die sich an Studenten der Informatik und Computerlinguistik richtete. Für Hinweise auf verbliebene Fehler bin ich dankbar.

Ich danke beiden Zuhörerkreisen für das lebhafte Interesse.

München, den 31.10.1997

H.Leiß
email: leiss@cis.uni-muenchen.de

Kapitel 1

Einführung: Typfreie oder getypte Programmiersprachen ?

An Beispielen werden wir kurz einige Vor- und Nachteile von typfreien und getypten Programmiersprachen erläutern. Anschließend werden der Zweck und die erwünschten Eigenschaften von Typsystemen skizziert. Dies soll eine grobe Vorstellung davon vermitteln, weshalb die Entwicklung von „starken“ Typsystemen eine wesentliche Rolle bei der Entwicklung neuer Programmiersprachen spielt.

1.1 Beispiele für getypte und ungetypte Programme

Beispiel 1.1.1 Die Quadrierung ganzer Zahlen programmiert man in verschiedenen Sprachen (siehe Kernighan/Ritchie[3], Banahan[1] und [5]) unterschiedlich:

C–Programm:
```
int squareC(int n)
    { return(n * n) }
```

SCHEME-Programm:
```
(define (squareS n) (* n n))
```

Man beachte folgende Unterschiede:

(i) Die Definition von `squareC` enthält eine explizite (Typ–)Deklaration, die den Typ des Arguments und des Ergebnisses festlegt.

(ii) Der Ausdruck `squareC("abc")` ist syntaktisch inkorrekt. Der Typfehler, `int` $\neq$ `string` (bzw. `const char-array` $\neq \uparrow$ `const-char`), wird zur Übersetzungszeit erkannt.

(iii) Der Ausdruck `(squareS "abc")` ist dagegen syntaktisch korrekt. Es tritt aber ein Typfehler zur Laufzeit des Programms auf, da `(*` `"abc"` `"abc")` nicht ausgewertet werden kann. Den Werten der Teilausdrücke werden Typmerkmale zugeordnet, die zur Laufzeit eine Typkorrektheitsprüfung ermöglichen.

Nach (iii) ist SCHEME, wie die meisten LISP–Dialekte, keine völlig ungetypte Sprache, sondern „schwach“ getypt. Ungetypte Sprachen sind aber Sprachen, bei denen die Inhalte von Speicherbereichen direkt manipuliert werden, also Assemblersprachen oder Teile von C.

Beispiel 1.1.1 deutet folgende Vorteile explizit getypter Sprachen an:

(i) Die Typinformation kann vom Übersetzer ausgenutzt werden; etwa für Optimierungen und Speicherplatzanforderungen.

(ii) Es ist keine Verwaltung von Typmerkmale zur Laufzeit nötig, was die Effizienz bei der Ausführung erhöht.

(iii) Typfehler werden zur Übersetzungszeit entdeckt; das erleichtert die Fehlersuche.

(iv) Die Typinformation ist als Dokumentation der beabsichtigten Verwendung bzw. als (sehr rudimentäre) Spezifikation der Bedeutung des Programms nützlich.

Beispiel 1.1.2 Die Spiegelung von Listen kann man in SCHEME und Pascal wie folgt programmieren:

- SCHEME–Programm:

```
(define (reverseS L)
   (cond ((equal L nil) L)
         (true (append (reverseS (cdr L)) (list (car L)))) ))
```

- Pascal–Programm:

```
type prt-list.elem = | list-elem;
     list-elem     = record
                        content: integer;
                        next:    ptr-list-elem
                     end;
procedure reverse-list(var first:ptr-list-elem)
   begin <Programmdetails> end;
```

Der hier wesentliche Unterschied besteht darin, daß man im Pascal–Programm den Typ der Listenelemente festlegen *muß* (z.B. `integer`), während das SCHEME–Programm auch auf Listen angewendet werden kann, deren Elemente von verschiedenem Typ sind.

Das Beispiel 1.1.2 deutet folgende Vorteile ungetypter Sprachen an:

(i) Typfreie Programme sind i.a. vielseitiger verwendbar als getypte.

(ii) Typfreie Sprachen ersparen dem Programmierer das Aufschreiben der Typinformation.

Es liegt nahe, daß man die Vorteile getypter und typfreier Sprachen verbinden möchte. Das Problem ist, wie weit das überhaupt möglich ist. Eine Lösung wird in der funktionalen Sprache *SML* (siehe [4]) verwirklicht:

Beispiel 1.1.3 Die Spiegelung von Listen kann man in *SML* wie folgt programmieren, wobei die Fallunterscheidung durch Mustervergleiche lesbar gestaltet wird:

```
fun reverseM [] = []
  | reverseM (head :: tail) = append (reverseM tail) [head];
```

Obwohl das Programm keine Typinformation enthält, ist der Übersetzer in der Lage, einen Typ (bzw. ein Schema für alle korrekten Typisierungen) aus dem Programmtext abzuleiten. Das Schema ist

$$\texttt{reverseM} : \alpha\texttt{-list} \to \alpha\texttt{-list},$$

wobei α eine Typvariable ist, die bei verschiedenen Verwendungen von reverseM durch unterschiedliche Typen belegt werden kann.

Man beachte:

i) `reverseM 17` und `reverseM [0,"ab",2,"cd"]` sind syntaktisch inkorrekt, was zur Übersetzungszeit erkannt wird.

ii) `reverseM [0,1,2,3]` und `reverseM ["ab","cd"]` sind syntaktisch korrekt und erzeugen auch keine Typfehler zur Laufzeit.

iii) Im Unterschied zu `reverseS` kann `reverseM` nur homogene Listen spiegeln, d.h. solche, deren Elemente vom gleichen Typ sind.

In *SML* hat man also eine Kombination von Vorteilen getypter und ungetypter Sprachen:

(i) Typen braucht der Programmierer nicht anzugeben, sie werden automatisch ermittelt.

(ii) Typfehler werden zur Übersetzungszeit entdeckt.

(iii) Durch *ein* Programm wird eine Schar von Funktionen definiert, die uniform in den Typen ihrer Argumente operieren.

1.2 Wozu sollen Typen dienen ?

Die herkömmliche Motivation für explizite Typisierung von Programmen besagt, daß Typinformation benötigt wird, damit ein Übersetzer effizienten Code erzeugen kann (z.B. durch optimale Speicherallokation und den Wegfall von Typprüfungen zur Laufzeit). Eine andere, zunehmend wichtigere Begründung, kommt aus der Softwaremethodologie:

These 1.2.1 Typen sind wesentlich für die geordnete Evolution von großen Softwaresystemen.

Evolvierende Softwaresysteme —im Unterschied zu kleinen Programmen — sind in der Regel

- fehlerhaft: Die Beseitigung vorhandener Fehler erzeugt oft neue Fehler,
- verbesserungsbedürftig: Die Beseitigung von ineffizienten Teilen erfordert oft größere Programmänderungen,
- mangelhaft strukturiert: Die Wartung des Systems stellt eigene Anforderungen an die Programmstruktur,
- funktional unvollständig: Ein von vielen Benutzern verwendetes Softwaresystem muß wachsenden Anforderungen (z.B. Bedienungskomfort, Kommunikation mit anderer Software, zusätzliche Aufgaben) angepaßt werden.

Softwaresysteme müssen daher laufend geändert werden. Evolvierende Systeme müssen aber trotz diesen Änderungen verlässlich sein. Typsysteme können dabei helfen, gewisse Programmeigenschaften zu garantieren oder mechanisch zu überprüfen. Dazu sind Typen von Bedeutung, die eine logische Interpretation zulassen ('types as propositions').

These 1.2.2 Subtypen sind wesentlich für die geordnete Erweiterung von Softwaresystemen.

Die Erweiterung eines Softwaresystems sollte

- die bestehende Funktionalität, Programmstruktur und weitere Eigenschaften erhalten,
- die Funktionalität unter weitgehender Ausnutzung der vorhandenen erweitern,
- die hinzukommende Funktionalität in wenigen Programmteilen lokalisieren.

Subtypen bilden eine Möglichkeit, die Erweiterung eines Systems unter diesen Gesichtspunkten zu gestalten (Sie werden benötigt, um objektorientierte Programmierung typtheoretisch zu interpretieren.) Unter einer *polymorph* getypten Sprache verstehen wir eine Programmiersprache, in der ein Ausdruck verschiedene Typen haben kann, je nach dem Kontext, in dem der Ausdruck vorkommt.

These 1.2.3 Typpolymorphismus ist wesentlich für die vielseitige Verwendbarkeit von Programmen, bei gleichzeitiger Vermeidung von Laufzeitfehlern.

Polymorph getypte Sprachen (vgl. 1.1.3) unterstützen die Entwicklung großer Softwaresysteme

- durch automatische Entdeckung von Typfehlern (als Folgefehler von logischen und Schreibfehlern!) zur Übersetzungszeit,
- durch den Ausschluß gewisser Laufzeitfehler (d.h. Laufzeit-Typunverträglichkeiten), wodurch der Grad der Korrektheit erhöht wird,
- durch die Möglichkeit, Programme auf höherer Abstraktionsebene zu schreiben, z,B, typuniforme Programme, oder solche, die Typen als Argumente verwenden.

1.3 Eigenschaften von Typsystemen

Damit die in 1.2 angedeuteten Zwecke erreicht werden können, müssen beim Sprachentwurf verschiedene Möglichkeiten erwogen werden, z.B.

(i) Welche Basistypen hat das Typsystem, und welche Typkonstruktionen ?

(ii) Sind höherstufige, rekursive, freie, algebraische, abstrakte Datentypen definierbar ?

(iii) Sind Typen „first-class-citizens“, d.h. können sie wie andere Objekte an Variable gebunden, an Funktionen als Argument übergeben, und durch Funktionen analysiert und berechnet werden ?

(iv) Kann man über Typen abstrahieren (mit λ, $\forall$, $\exists$), auch über Typkonstruktoren ?

(v) Sind Subtypen, auch auf höheren Stufen, definierbar ?

(vi) Ist die Gesamtheit der Typen selbst wieder ein Typ ?

Verschiedene Entwurfsentscheidungen für (i)–(vi) beeinflussen die Eigenschaften des Typsystems. An folgenden Eigenschaften besteht dabei besonderes Interesse:

1. Typisierungen sollten die Ausdruckskraft der Sprache nicht zu sehr einschränken.

Am Beispiel 1.1.2 haben wir gesehen, daß das „starre“ Typsystem von Pascal es erforderlich macht, die Spiegelung von (homogenen) Listen für jeden Element-Typ gesondert zu programmieren. Größere Flexibilität erreicht man durch verschiedene Formen von Typpolymorphismus, d.h. durch Verwendung von Typschemata, Typquantoren oder Typabstraktion und –Applikation.

2. Typisierungen sollten die Strukturierungen von Programmen erleichtern.

Dazu benötigt man abstrakte Datentypen, höherstufige Typen, sowie Modul– und Interfacekonzepte. Diese sollten es erlauben, eine strenge Blockstrukturierung in geordneter Weise zu umgehen, und es ermöglichen, ein großes Programm in abgeschlossenen Einheiten unabhängig voneinander zu compilieren.

3. Typisierungen sollten weitgehend statische Programmeigenschaften erfassen.

Das bedeutet, daß Typprüfungen zur Laufzeit des Programms nur dort, wo es unbedingt nötig ist, vorgenommen werden: etwa in der Systemprogrammierung, bei der Speicherbereinigung, beim ‘Bootstrapping’ des Übersetzers und bei Typisierungen, die die Übersetzungsphase überdauern müssen („dynamic types“). Insbesondere ist erwünscht:

3a: Entscheidbarkeit der Typkorrektheit oder Typisierbarkeit von Programmen

3b: Transparenz von Typisierungsalgorithmen.

Hierbei ist unter Transparenz gemeint, daß in den Fällen, wo automatisch Typisierungen für ungetypte oder nur partiell getypte Programme ermittelt werden, diese Typen vom Programmierer nachvollziehbar sind. Das beschränkt die Komplexität der verwendeten Typisierungsalgorithmen.

4. Typen sollten als logische Aussagen interpretierbar sein.

Diese Eigenschaft erlaubt es, Gemeinsamkeiten von Programmen und Beweisen als Aspekte derselben Sache anzusehen und methodisch auszunutzen: Programmausführung als Beweisnormalisierung, Klassen von Aussagen als Typsysteme. Natürlich ist diese Analogie nicht durchgängig; z.B. entsprechen den rekursiv definierten Typen normalerweise keine logisch sinnvollen Aussagen mehr (bzw. keine beweisbaren, so daß die Programme von diesem Typ nicht mehr als Beweise verstanden werden können). Auf diese Eigenschaft wird im folgenden kaum näher eingegangen.

Literatur

[1] Mike Banahan. *The C Book. Featuring the Draft ANSI–C Standard.* Addison Wesley, 1988.

[2] Luca Cardelli. Typeful programming. Technical report, Digital Equipment Corporation, Palo Alto, 1989.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[4] Robin Milner, Robert Harper, and Mads Tofte. *The Definition of Standard ML.* MIT Press, 1990.

[5] Jonathan Rees and William Clinger. The revised^3 report on the algorithmic language scheme. Technical report, MIT,Artificial Intelligence Laboratory, 1986.

Kapitel 2

Der typfreie λ–Kalkül

In diesem Abschnitt betrachten wir die β–Reduktion typfreier λ–Terme als idealisierte operationale Semantik einer funktionalen Programmiersprache.

2.1 Syntax des λ–Kalküls

Definition 2.1.1 (λ–Terme) (i) Variable $x_0, x_1, \ldots$ und Konstante $c_0, c_1, \ldots$ sind λ-Terme.

(ii) Sind t_1 und t_2 λ–Terme, so auch $(t_1 \cdot t_2)$.

(iii) Ist x eine Variable und t ein λ–Term, so ist auch λxt ein λ–Term.

In Zukunft schreiben wir solche Definitionen in der Kurzform

$$t := x \quad | \quad c \quad | \quad (t \cdot t) \quad | \quad \lambda xt.$$

Dies ist als eine simultane induktive Definition

$$\begin{array}{lcl|l|l|l|l} x & = & v & x' & & & \\ c & = & a & c' & & & \\ t & = & x & c & (t \cdot t) & \lambda xt. & \end{array}$$

der Menge x aller Variablen, der Menge c aller Konstanten, und der Menge t aller Terme zu verstehen. Es werden x, c, und t aber auch weiterhin als Bezeichnungen für individuelle Variable, Konstante und Terme benutzt.

Beispiel 2.1.2 Wir schreiben

$$\begin{array}{lll} (ts) & \text{statt} & (t \cdot s), \\ t_1t_2t_3 & \text{statt} & ((t_1t_2)t_3), \\ (\lambda x_1 \ldots x_n.t) & \text{statt} & \lambda x_1 \lambda x_2 \ldots \lambda x_n t, \end{array}$$

und lassen Außenklammern normalerweise weg.

Beispiel 2.1.3 Häufig vorkommende Terme mit ihren üblichen Bezeichnungen sind:

$$\begin{array}{lcllcl} I & = & \lambda x.x, & K & = & \lambda xy.x, \\ S & = & \lambda xyz.xz(yz), & B & = & \lambda xyz.x(yz), \\ \mathit{true} & = & \lambda xy.x, & \mathit{false} & = & \lambda xy.y, \\ \Delta & = & \lambda x.xx, & \Omega & = & \Delta\Delta, \\ Y & = & \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)). & & & \end{array}$$

Definition 2.1.4 (Freie Variable)

$$\begin{array}{lcl} \mathit{frei}(x) & = & \{x\}, \\ \mathit{frei}(c) & = & \emptyset, \\ \mathit{frei}(ts) & = & \mathit{frei}(t) \cup \mathit{frei}(s), \\ \mathit{frei}(\lambda xt) & = & \mathit{frei}(t) - \{x\}. \end{array}$$

Definition 2.1.5 (Substituition) Für λ–Terme s, t und Variable x definiere die *Einsetzung* $[s/x]t$ von s in t für freie Vorkommen von x durch:

$$\begin{array}{lcll} [s/x]x & \equiv & s, & \\ [s/x]y & \equiv & y, & \text{für } y \not\equiv x \\ [s/x]c & \equiv & c, & \\ [s/x](t_1 t_2) & \equiv & ([s/x]t_1\ [s/x]t_2), & \\ [s/x]\lambda xt & \equiv & \lambda xt, & \\ [s/x]\lambda yt & \equiv & \lambda y.[s/x]t, & \text{für } y \not\equiv x,\ y \notin \mathit{frei}(s) \text{ oder } x \notin \mathit{frei}(t) \\ [s/x]\lambda yt & \equiv & \lambda z.[s/x][z/y]t, & \text{für } y \not\equiv x,\ y \in \mathit{frei}(s) \text{ und } x \in \mathit{frei}(t). \end{array}$$

Im letzten Fall ist z die nächste Variable in $x_0, x_1, \ldots$, die nicht frei in st vorkommt.

2.2 Die Gleichungstheorien $\lambda\beta$ und $\lambda\beta\eta$

Die λ–Terme sollen als Funktionen und Daten interpretiert werden. Syntaktisch verschiedene Terme können dabei dieselbe Interpretation haben. Es wird axiomatisch festgelegt, welche Gleichungen bei jeder Interpretation mindestens gelten sollen.

Definition 2.2.1 ($\lambda\beta$ und $\lambda\beta\eta$–Kalkül) Für alle Variablen x, y und λ-Terme t, t_i, s enthält der $\lambda\beta$–Kalkül folgende Axiome und Regeln:

Axiome: (α) $\lambda xt = \lambda y.[y/x]t$ für $y \notin \mathit{frei}(t)$
(β) $(\lambda xt)s = [s/x]t$
(ρ) $t = t$

Regeln: (σ) $\dfrac{t = s}{s = t}$ (τ) $\dfrac{t = s, \quad s = r}{t = r}$

(μ) $\dfrac{t_1 = t_2}{st_1 = st_2}$ (ν) $\dfrac{t_1 = t_2}{t_1 s = t_2 s}$ (ξ) $\dfrac{t_1 = t_2}{\lambda xt_1 = \lambda xt_2}$

Der $\lambda\beta\eta$–Kalkül enthält zusätzlich das Axiom

$$(\eta) \quad \lambda x.tx = t, \quad \text{falls } x \notin \mathit{frei}(t).$$

Für die Beweisbarkeit einer Gleichung $t_1 = t_1$ in diesen Kalkülen schreiben wir

$$\lambda\beta \vdash t_1 = t_2 \quad \text{bzw.} \quad \lambda\beta\eta \vdash t_1 = t_2.$$

λ–Terme, die sich durch „gebundene Umbenennung“, d.h. α–Konversion, ineinander überführen lassen, betrachten wir als gleich. Was eine Interpretation dieser Theorien ist, muß noch gesagt werden.

2.3 Die Reduktionstheorien β und $\beta\eta$

Gleichungstheorien besagen, welche λ–Terme dieselbe denotationale Bedeutung haben, z.B. dieselbe Funktion definieren. Wir sind aber auch an der operationalen Bedeutung der Terme interessiert, durch die die Auswertung von Programmen erfaßt wird.

Der Prozeß der Auswertung wird im λ–Kalkül als ein Termvereinfachungsprozeß dargestellt. Dabei werden die Gleichungen nur „in einer Richtung“ verwendet.

Definition 2.3.1 (β– und $\beta\eta$–Reduktionskalkül) Die Kalküle enthalten dieselben Axiome und Regeln wie die entsprechenden Gleichungstheorien, außer:

(i) Statt $t_1 = t_2$ wird $t_1 \to t_2$ geschrieben.

(ii) Die Symmetrieregel (σ) entfällt.

Insbesondere haben wir die Reduktionsregeln

$$\begin{array}{ll} (\to_\beta) & (\lambda x.t)s \to [s/x]t \\ (\to_\eta) & (\lambda x.tx) \to t, \qquad \text{falls } x \notin \mathit{frei}(t). \end{array}$$

Hierbei heißt $(\lambda xt)s$ ein *β–Redex*, $(\lambda x.tx)$ ein *η–Redex*. Ein Term ist *β–Normalform*, wenn er keine β–Redexe enthält.

Ein Übergang zu $t_1 \to t_2$ gemäß einer der Reduktionsregeln entspricht einem Auswertungsschritt eines Programms.

Falls t_1 zu t_2 in endlich vielen Schritten gemäß den Reduktionskalkülen reduziert werden kann (d.h. wenn $t_1 \to t_2$ abgeleitet werden kann), so schreiben wir

$$\lambda\beta \vdash t_1 \to t_2 \text{ bzw. } \lambda\beta\eta \vdash t_1 \to t_2.$$

Beispiel 2.3.2 (i) Nicht jeder Term kann zu einer β–Normalform reduziert werden: für $(\Delta\Delta)$ ergibt sich $(\Delta\Delta) = ((\lambda x.xx)\Delta) \to_\beta (\Delta\Delta) \to \ldots$

(ii) Es gibt Terme, bei denen manche Reduktionsfolgen zu einer β–Normalform führen, während andere unendlich viele β–Reduktionen enthalten.

$$\begin{array}{lcllll} Kx\Omega & \equiv & (\lambda xy.x)x\Omega & \to_\beta & (\lambda y.x)\Omega & \to_\beta \; x \\ Kx\Omega & \equiv & Kx(\Delta\Delta) & \to_{\beta,(i)} & Kx(\Delta\Delta) & \to_\beta \; \ldots \end{array}$$

(iii) Fallunterscheidung kann wie folgt simuliert werden: Mit $true := \lambda xy.x$ und $false := \lambda xy.y$ erhält man für $(if\ b\ then\ s\ else\ t) := bst$:

$$\begin{array}{rcl} \lambda\beta \vdash b = true & \Rightarrow & \lambda\beta \vdash (if\ b\ then\ s\ else\ t) = s \\ \lambda\beta \vdash b = false & \Rightarrow & \lambda\beta \vdash (if\ b\ then\ s\ else\ t) = t. \end{array}$$

(iv) Rekursive Definitionen erhält man über den *Fixpunktoperator*

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

Für $\tilde{f} := \lambda x.f(xx)$ ist

$$\begin{array}{lcll} \lambda\beta \vdash Yf & \to_\beta & \tilde{f}\tilde{f} & = & (\lambda x.f(xx))\tilde{f} \\ & \to_\beta & f(\tilde{f}\tilde{f}) & = & f(Yf). \end{array}$$

Also gilt $\lambda\beta \vdash Yf = f(Yf)$, d.h. Yf ist ein Fixpunkt von f. Daher kann für jeden λ–Term $e(x)$ die Rekursionsgleichung

$$(*) \quad x = e(x)$$

durch $(rec\ x.e) := Y(\lambda x.e)$ gelöst werden:

$$\begin{array}{rl} rec\ x.e = Y(\lambda x.e) = (\lambda x.e)(Y(\lambda x.e)) \\ = [Y(\lambda x.e)/x]e = e(rec\ x.e). \end{array}$$

Beachte, daß $rec\ x.e$ auch rekursiv definierte Daten erlaubt; die meisten Programmiersprachen lassen nur rekursive Definitionen von Funktionen zu, wo e die Form $\lambda y.t$ haben muß.

Bemerkung 2.3.3 Die β–Reduktion entspricht einer call-by-name Auswertung; beachte, daß nach (ii) ja $Kxy = (\lambda xy.x)xy = x$ auch für divergierendes Argument Ω statt y). Die Auswertung in Programmiersprachen, die sich am λ–Kalkül orientieren –LISP, *SML*–, verwendet aber eine call-by-value-Auswertung: die Argumente müssen zuerst reduziert werden.

2.4 Interpretation von λ–Termen durch Umgebungsmodelle

Wir haben keinen Unterschied gemacht zwischen Termen, die Objekte (Daten) und solchen, die Funktionen (Programme) bezeichnen. Die Extensionalitätsforderung

$$\forall y\ (y = \lambda x(y \cdot x))$$

besagt sogar, daß jedes Objekt eine Funktion sei. Dies kann aber nicht im wörtlichen Sinn gemeint sein, wenn wir den mengentheoretischen Funktionsbegriff beibehalten wollen: Um $(t \cdot t)$ zu interpretieren, bräuchten wir Funktionen f mit $f(f)$ als Wert für $(t \cdot t)$. Solche Funktionen, die ihr eigenes Argument sein können, gibt es aber nicht. (Es sei denn, man ändert die Mengenlehre!) Wir schwächen daher unsere Auffassung etwas ab: λ–Terme werden nicht durch Funktionen interpretiert, sondern durch Objekte, die Funktionen repräsentieren.

Definition 2.4.1 Sei $(D, \cdot)$ mit einer Operation $\cdot : D \times D \to D$ gegeben. Eine Funktion $f : D^n \to D$ wird durch $a \in D$ *repräsentiert*, wenn $f(b_1, \ldots, b_n) = ((\ldots((a \cdot b_1) \cdot b_2)\ldots) \cdot b_n)$.

$$(D^n \to D) := \{f : D^n \to D \mid f \text{ ist repräsentierbar.}\}$$

Mit $(D \to D)$ sei die Menge der einstelligen repräsentierbaren Funktionen von $(D, \cdot)$ gemeint.

Definition 2.4.2 $(D, \cdot, \Psi)$ ist ein *Funktionalbereich*, wenn $\Psi : (D \to D) \to D$, und $\Psi(f)$ die Funktion $f \in (D \to D)$ repräsentiert. $\Psi(f)$ heiße der *kanonische Repräsentant* von f.

Statt durch $(D, \cdot, \Psi)$ kann man einen Funktionalbereich auch als Tripel (D, Φ, Ψ) angeben, wobei

$$\Phi \circ \Psi = Id_{(D \to D)} \quad \text{mit} \quad (D \to D) := \{\, \Phi(a) \mid a \in D \,\} \subseteq \{\, f \mid f : D \to D \,\}.$$

Dann muß man $\cdot$ durch $a \cdot d := \Phi(a)(d)$ definieren, womit $\Phi(a) = \lambda d \in D.\ a \cdot d$ wird.

Definition 2.4.3 Sei $\mathcal{D} = (D, \cdot^D, \Psi)$ ein Funktionalbereich, $\eta : Var \to D$ eine Belegung der Variablen. Definiere den *Wert* $[\![t]\!]^D_\eta$ des λ–Terms t in $\mathcal{D}$ bei η durch

$$\begin{array}{lllll} (i) & [\![x]\!]^D_\eta & = & \eta(x), \\ (ii) & [\![(t_1 \cdot t_2)]\!]^D_\eta & = & [\![t_1]\!]^D_\eta \cdot^D [\![t_2]\!]^D_\eta \\ (iii) & [\![\lambda x t]\!]^D_\eta & = & \Psi(\lambda d \in D.[\![t]\!]^{\mathcal{D}}_{\eta[d/x]}). \end{array}$$

Dies ist nur dann eine korrekte Definition, wenn für jeden λ–Term t tatsächlich

$$(iv) \qquad \lambda d \in D.[\![t]\!]^{\mathcal{D}}_{\eta[d/x]} \ \in (D \to D)$$

ist. Dann heißt $\mathcal{D}' = (D, \cdot, [\![\ ]\!])$ das durch $\mathcal{D}$ bestimmte *Umgebungsmodell* des λ-Kalküls.

Proposition 2.4.4 $[\![t]\!]^{\mathcal{D}}_{\eta_1} = [\![t]\!]^{\mathcal{D}}_{\eta_2}$, falls $\eta_1(x) = \eta_2(x)$ für alle $x \in \mathit{frei}(t)$.

Durch Nachrechnen zeigt man, daß Umgebungsmodelle die beweisbaren Gleichungen erfüllen. Die (ξ)-Regel ist korrekt, d.h. man hat

$$(\xi) \qquad D \models \forall \bar{y}(\forall x(t_1 = t_2) \to \lambda x t_1 = \lambda x t_2),$$

da $[\![\lambda x t_i]\!]^D_\eta = \Psi(\lambda d \in D.[\![t_i]\!]^{\mathcal{D}}_{\eta[d/x]})$ nur von der Funktion abhängt, nicht von den Termen t_i.

Lemma 2.4.5 In dem durch $\mathcal{D}$ bestimmten Umgebungsmodell $(D, \cdot, [\![\ ]\!])$ gelten:

$$\begin{array}{lrcl} \text{(Substitution)} & [\![[s/x]t]\!]^D_\eta & = & [\![t]\!]^{\mathcal{D}}_{\eta[[\![s]\!]^D_\eta/x]} \\ (\alpha\text{–Konversion}) & [\![\lambda xt]\!]^D_\eta & = & [\![(\lambda y.[y/x]t)]\!]^D_\eta \text{ für } y \notin \mathit{frei}(t), \\ (\beta\text{–Konversion}) & [\![(\lambda xt)s]\!]^D_\eta & = & [\![[s/x]t]\!]^D_\eta. \end{array}$$

Bemerkung 2.4.6 Sei $(I\!N, \cdot, \Psi)$ mit $e \cdot n := \varphi_e(n)$ und $\Psi(\varphi) :=$ kleinstes e mit $\varphi = \varphi_e$, bezüglich einer Aufzählung $\{\varphi_e \mid e \in I\!N\}$ aller einstelligen partiell rekursiven Funktionen. Dann ist $(I\!N, \cdot, \Psi)$ *kein* Funktionalbereich mit $(i) - (iv)$. Denn sonst müßte jedes φ_e einen Fixpunkt haben.

Kombinatorische Vollständigkeit

Wir können die β–Konversionsaxiome auch in der folgenden Form schreiben:

$$(\beta) \quad \forall y_1 \ldots y_n \forall x_1 \ldots x_m((\lambda x_1 \ldots x_m.t)x_1 \ldots x_m = t)$$

für jeden λ–Term $t(x_1, \ldots, x_m, y_1, \ldots, y_n)$. Dies drückt zweierlei aus:

(i) Für feste $(b_1, \ldots, b_n)$ ist die durch $(a_1, \ldots, a_m) \mapsto [\![t]\!]^{\mathcal{D}}[a_1/x_1, \ldots, a_m/x_m, b_1/y_1, \ldots, b_n/y_n]$ definierte Funktion repräsentierbar, und zwar durch $[\![\lambda x_1 \ldots x_m.t]\!]^{\mathcal{D}}[b_1/y_1, \ldots, b_n/y_n]$.

(ii) Die Auswahl dieses Repräsentanten, $(b_1, \ldots, b_n) \mapsto [\![\lambda x_1 \ldots x_m.t]\!]^{\mathcal{D}}[b_1/y_1, \ldots, b_n/y_n]$, ist selbst repräsentierbar, und zwar durch $[\![\lambda y_1 \ldots y_n.\lambda x_1 \ldots x_m.t]\!]^{\mathcal{D}}$.

Beachte, daß beides aus

$$\exists z \forall y_1 \ldots y_n \forall x_1 \ldots x_m((zy_1 \ldots y_n)x_1 \ldots x_n = t)$$

folgt. Das motiviert folgende Definition:

Definition 2.4.7 $(D, \cdot)$ heißt *kombinatorisch vollständig*, wenn für jeden Term $t(x_1, \ldots, x_n)$ über $\{\cdot\}$ gilt:

$$(D, \cdot) \models \exists z \forall x_1 \ldots x_n(zx_1 \ldots x_n = t)$$

Terme über $L = \{\cdot\}$ nennen wir auch *Kombinatorenterme*, im Unterschied zu λ–Termen.

Wir stellen gleich einen Zusammenhang zwischen den Axiomen und Regeln des λ–Kalküls und mehr algebraischen Forderungen wie der kombinatorischen Vollständigkeit her.

Satz 2.4.8 (Schönfinkel) Für $D = (D, \cdot)$ sind folgende Aussagen äquivalent:

(i) $D \models \exists k \forall xy(kxy = x) \ \wedge \ \exists s \forall xyz(sxyz = xz(yz))$.

(ii) $D \models \exists z \forall x_1 \ldots x_n(zx_1 \ldots x_n = t)$ für jeden Kombinatorenterm $t(x_1, \ldots, x_n)$.

(iii) $D \models \forall y_1 \dots y_m \exists z \forall x_1 \dots x_n (z x_1 \dots x_n = t)$ für jeden Kombinatorenterm $t(x_1, \dots, y_m)$.

(iv) $D \models \forall y_1 \dots y_m \exists z \forall x_1 \dots x_n (z x_1 \dots x_n = t)$ für jeden Kombinatorenterm $t(x_1, \dots, y_m, z)$.

Beweis: Offenbar gelten (iv) ⇒ (iii) ⇒ (ii) ⇒ (i). Wir zeigen (i) ⇒ (iii) ⇒ (iv).
(iii) ⇒ (iv): Nach (iii) gibt es zu $t' = y(xxy)$ ein $w \in D$ mit $D \models \forall xy(wxy = t')$, also

$$D \models \forall y(wwy = y(wwy)),$$

d.h. wwy ist Fixpunkt von y. Ebenso gibt es mit (iii) zu $y_1, \dots, y_m$ ein $e \in D$, so daß

$$D \models \forall z \forall x_1 \dots x_n (e \cdot z x_1 \dots x_n = t)$$

Da $z := wwe$ ein Fixpunkt von e ist nach obiger Bemerkung, folgt $z x_1 \dots x_n = (ez) x_1 \dots x_n = t$ für dieses z, also

$$D \models \forall y_1 \dots y_m \exists z \forall x_1 \dots x_n (z x_1 \dots x_n = t)$$

(i) ⇒ (iii): Wähle k,s wie in (i). Zu $t(x_1, \dots, x_n, \bar{y})$ definiere einen Term $(\lambda^* x_1 \dots x_n.t)$ über $L = \{\cdot, k, s\}$, so daß

$$(*) \qquad D \models \forall \bar{y} \forall \bar{x} ((\lambda^* \bar{x}.t)\bar{x} = t)$$

Hieraus folgt direkt (iii). Wir definieren induktiv über t:

$$(\lambda^* x.t), \quad \text{mit } \mathit{frei}(\lambda^* x.t) = \mathit{frei}(t) - \{x\}$$

durch

$$\begin{array}{lcl} \lambda^* x.x & := & skk, \\ \lambda^* x.(t_1 \cdot t_2) & := & s(\lambda^* x.t_1)(\lambda^* x.t_2) \\ \lambda^* x.y & := & ky \end{array}$$

Dann gilt $D \models \forall \bar{y} x((\lambda^* x.t) \cdot x = t)$, für jedes $t \in Term_{\{\cdot,k,s\}}$, wegen $[\![(\lambda^* x.t) \cdot x]\!]^D_\eta = [\![t]\!]^D_\eta$, was man leicht nachrechnet:

$$\begin{array}{lcl} [\![(\lambda^* x.x) \cdot x]\!]^D_\eta & = & skk \cdot [\![x]\!]^D_\eta \\ & = & k[\![x]\!]^D_\eta (k[\![x]\!]^D_\eta) \\ & = & [\![x]\!]^D_\eta \\ [\![(\lambda^* x.y) \cdot x]\!]^D_\eta & = & k \cdot [\![y]\!]^D_\eta \cdot [\![x]\!]^D_\eta \\ & = & [\![y]\!]^D_\eta \\ [\![(\lambda^* x.(t_1 \cdot t_2)) \cdot x]\!]^D_\eta & = & s \cdot [\![(\lambda^* x.t_1)]\!]^D_\eta \cdot [\![(\lambda^* x.t_2)]\!]^D_\eta \cdot [\![x]\!]^D_\eta \\ & = & [\![(\lambda^* x.t_1) \cdot x]\!]^D_\eta \cdot [\![(\lambda^* x.t_2) \cdot x]\!]^D_\eta \\ & = & [\![t_1]\!]^D_\eta \cdot [\![t_2]\!]^D_\eta \\ & = & [\![(t_1 t_2)]\!]^D_\eta . \end{array}$$

Die Behauptung folgt nun durch Iteration, wenn man

$$(\lambda^* x_1 \dots x_n.t) := (\lambda^* x_1.(\dots (\lambda^* x_n.t) \dots))$$

setzt. □

Nach (i) des Satzes folgt die kombinatorsiche Vollständigkeit schon daraus, daß zwei bestimmte definierbare Funktionen repräsentierbar sind: k repräsentiert die Bildung konstanter Funktionen, $kx = \lambda y.x$, und s repräsentiert die Einsetzung (Substitution) $x(z, y(z))$ einer einstelligen Funktion y in eine zweistellige Funktion x, also $sxy = \lambda z.(xz(yz))$ in der Darstellung von Curry.

Nach (iv) sind auch *rekursive* Definitionen $zx_1 \dots x_m = t(x_1 \dots x_m, z)$ möglich, bei denen z in t frei vorkommt.

(Evtl: Grob gesagt:)

Kombinatorenmodelle

Wir haben noch kein Verfahren, Umgebungsmodelle des λ–Kalküls zu konstruieren, da die Bedingung

$$(iv) \qquad \lambda d \in D.[\![t]\!]^{\mathcal{D}}\eta[d/x] \in (D \to D)$$

eine Abschlußbedingung an $(D \to D)$ ist, die nicht in offensichtlicher Weise realisiert werden kann. Eine einfache algebraische Beschreibung erhalten wir aber wie folgt. Erfüllt $D = (D, \cdot, \Psi)$ die Bedingungen (i)–(iv), so muß für jedes $a \in D$ die Funktion

$$\lambda d \in D.\ a \cdot d\ \in (D \to D) \quad \text{sein, da} \quad [\![\lambda x(y \cdot x)]\!]^{\mathcal{D}}[a/y] = \Psi(\lambda d \in D.\ a \cdot d)$$

sein soll. Aber auch der Übergang zum kanonischen Repräsentanten, d.h. die Funktion $a \mapsto [\![\lambda x.ax]\!]^{\mathcal{D}}$, muß repräsentierbar sein, da $[\![\lambda y \lambda x.yx]\!]^{\mathcal{D}} = \Psi(\lambda a \in D.[\![(\lambda x(ax))]\!]^{\mathcal{D}})$. Wir wählen einen Repräsentanten ϵ hierfür aus.

Definition 2.4.9 $\mathcal{D} = (D, \cdot, \epsilon)$ ist ein *Kombinatorenmodell*, wenn D mindestens zwei Elemente hat, $(D, \cdot)$ kombinatorisch vollständig ist, und wenn für die durch $a \sim b :\iff \forall c\,(a \cdot c = b \cdot c)$ definierte Äquivalenzrelation gilt:

$$(i) \quad (D, \cdot) \models \forall x(\epsilon x \sim x), \qquad \text{und} \qquad (ii) \quad (D, \cdot) \models \forall xy(x \sim y \to \epsilon x = \epsilon y).$$

$\mathcal{D}$ heißt *stabiles Kombinatorenmodell*, wenn zusätzlich $\epsilon \cdot \epsilon = \epsilon$ gilt.

Aufgabe 2.4.1 Ist $(D, \cdot, e)$ ein Kombinatorenmodell, so ist $(D, \cdot, ee)$ ein stabiles Kombinatorenmodell. Sind $(D, \cdot, e_1)$ und $(D, \cdot, e_2)$ stabile Kombinatorenmodelle mit $e_1 \sim e_2$, so ist $e_1 = e_2$.

(Evtl: Beweis)

Da ϵ nach (i) aus jeder $\sim$–Klasse ein Element auswählt, ordnet es jeder repräsentierbaren Funktion einen Repräsentanten zu. Es liegt nahe, diesen als „kanonischen“ Repräsentanten zu nehmen:

Satz 2.4.10 (A.Meyer 1982) (i) Sei $(D, \cdot, \epsilon)$ ein stabiles Kombinatorenmodell. Definiere $\Psi : (D \to D) \to D$ durch

$$\Psi(\lambda d \in D.\ a \cdot d) \ :=\ \epsilon \cdot a$$

Dann erfüllt $(D, \cdot, \Psi)$ die Bedingungen (i)–(iv), so daß das mit Ψ gebildetete $(D, \cdot, [\![\ ]\!])$ ein Umgebungsmodell des λ–Kalküls ist.

(ii) Erfüllt $(D, \cdot, \Psi)$ die Bedingungen (i)–(iv), so ist $(D, \cdot, \epsilon)$ mit $\epsilon := [\![\lambda xy.xy]\!]^{\mathcal{D}}$ ein stabiles Kombinatorenmodell.

Beweis: (i): Sei t ein λ-Term mit $frei(t) \subseteq \{x_1, \ldots, x_n\}$. Wir zeigen durch Induktion über t, daß für jede Belegung η die Abbildung

$$\lambda(d_1, \ldots, d_n) \in D^n \; [\![t]\!]\eta[d_1/x_1, \ldots, d_n/x_n]$$

repräsentierbar ist. Der Beweis zeigt, daß dann auch $\lambda d \in D[\![t]\!]^D_\eta[d/x_i]$ repräsentierbar ist, also die kritische Bedingung (iii) aus Definition 2.4.3 gilt.

$t \equiv x_i$: Für den λ-freien Term x_i gibt es wegen der kombinatorischen Vollständigkeit ein Element $c_{n,i} \in D$, so daß $c_{n,i}d_1 \cdots d_n = d_i = [\![x_i]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n]$ für alle $(d_1, \ldots, d_n) \in D^n$.

$t \equiv (t_1 \cdot t_2)$: Nach Induktion gibt es (von n abhängende) $c_1, c_2 \in D$ mit

$$c_i d_1 \cdots d_n = [\![t_i]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n] \qquad \text{für alle } (d_1, \ldots, d_n) \in D^n.$$

Nach der kombinatorischen Vollständigkeit gibt es ein $c \in D$, so daß

$$cd_1 \cdots d_n = (c_1 d_1 \cdots d_n)(c_2 d_1 \cdots d_n) = [\![t_1]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n] \cdot [\![t_2]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n].$$

Also ist c ein Repräsentant von $\lambda(d_1, \ldots, d_n) \in D^n \; [\![(t_1 \cdot t_2)]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n]$.

$t \equiv \lambda x_{n+1}\, s$: Nach Induktion gibt es ein $c \in D$ mit

$$cd_1 \cdots d_n d_{n+1} = [\![s]\!]^D_\eta[d_1/x_1, \ldots, d_{n+1}/x_{n+1}].$$

Hieran sieht man, daß $cd_1 \cdots d_n$ ein Repräsentant von

$$f := \lambda d_{n+1} \in D \; [\![s]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n][d_{n+1}/x_{n+1}]$$

ist, also $f \in (D \to D)$ liegt. Daher ist (iv) erfüllt und dann durch (iii)

$$[\![\lambda x_{n+1} s]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n] = \Psi(f) = e \cdot (cd_1 \cdots d_n)$$

Da $(D, \cdot)$ kombinatorisch vollständig ist, gibt es ein Element $\tilde{c}$ mit

$$\tilde{c}d_1 \cdots d_n = e(cd_1 \cdots d_n) = [\![\lambda x_{n+1} s]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n]$$

für alle $d_1, \ldots, d_n \in D$, und daher ist, wie behauptet, auch

$$\lambda(d_1, \ldots, d_n) \in D^n [\![\lambda x_{n+1} s]\!]^D_\eta[d_1/x_1, \ldots, d_n/x_n]$$

eine repräsentierbare Funktion.

(ii) Da ein Umgebungsmodell die beweisbaren Gleichungen erfüllt, erhält man für $a \in D$

$$\begin{aligned} \epsilon \cdot a &= [\![\lambda xy.xy]\!]^D_\eta[a/x] \cdot [\![x]\!]^D_\eta[a/x] = [\![(\lambda xy.xy) \cdot x]\!]^D_\eta[a/x] \\ &= [\![\lambda y.xy]\!]^D_\eta[a/x] = \Psi(\lambda d \in D \; [\![xy]\!]^D_\eta[d/x]) = \Psi(\lambda d \in D \; a \cdot d). \end{aligned}$$

Also repräsentiert $\epsilon \cdot a$ dieselbe einstellige Funktion wie a, d.h. $\epsilon a \sim a$. Ist $a \sim b$, so ist $\epsilon a = \epsilon b$, da ja $(\lambda d \in D \; a \cdot d) = (\lambda d \in D \; b \cdot d)$. Die Bedingung $\epsilon\epsilon = \epsilon$ folgt aus der Definition von ϵ wieder damit, daß in $(D, \cdot, [\![\;]\!])$ die beweisbaren Gleichungen gelten. □

2.5 Ein Modell des typfreien λ-Kalküls

Um ein Modell des λ-Kalküls zu konstruieren, muß man eine geeignete Teilmenge von Funktionen für die durch Elemente repräsentierbaren Funktionen finden. Dies ist D.Scott[4] als erstem gelungen, indem er die bezüglich einer speziellen Topologie *stetigen* Funktionen genommen hat. (Die Stetigkeit ist eine Verallgemeinerung der einseitigen Limesstetigkeit von Funktionen auf $\mathbb{R}$: ist $\lim a_n = a$ mit $a_n \leq a$, so ist $f(a) = \lim f(a_n)$ mit $f(a_n) \leq f(a)$.)

Wir geben nur die einfachste bekannte Konstruktion eines Modells an. Die Elemente sind hier Mengen a, die durch endliche Teilmengen $a_n \subseteq a$ approximiert werden:

Definition 2.5.1 (Engeler 1980, Plotkin 1972) Sei A eine nicht-leere Menge und $(\alpha \to m)$ ein aus α und m gebildetes geordnetes Paar, das nicht in A liegt, etwa $:= ((\alpha, m), A)$. Sei $G(A)$ die kleinste Menge mit

(i) $A \subseteq G(A)$,

(ii) Ist $\alpha \subseteq G(A)$ endlich und $m \in G(A)$, so ist $(\alpha \to m) \in G(A)$.

Sei $D_A = 2^{G(A)}$ die Menge aller Teilmengen von $G(A)$. Für $a, b \in D_A$ setze

$$\begin{aligned} a \cdot b &:= \{\, m \in G(A) \mid \text{Für ein endliches } \beta \subseteq b \text{ ist } (\beta \to m) \in a \,\} \\ e &:= \{\, (\alpha \to (\beta \to m)) \mid \alpha, \beta \subseteq G(A) \text{ endlich}, \ m \in \alpha \cdot \beta \,\} \end{aligned}$$

Beachte, daß

$$a \cdot b = \bigcup \{\, \alpha \cdot \beta \mid \alpha \subseteq a \text{ endlich}, \ \beta \subseteq b \text{ endlich} \,\}.$$

Satz 2.5.2 Das *Graphenmodell* $(D_A, \cdot, e)$ über A ist ein Kombinatorenmodell.

Beweis: Definiere die Elemente

$$\begin{aligned} k &:= \{\, (\alpha \to (\beta \to m)) \mid \alpha, \beta \subseteq G(A) \text{ endlich}, \ m \in \alpha \,\}, \\ s &:= \{\, (\alpha \to (\beta \to (\gamma \to m))) \mid \alpha, \beta, \gamma \subseteq G(A) \text{ endlich}, \ m \in \alpha \cdot \gamma \cdot (\beta \cdot \gamma) \,\} \end{aligned}$$

Es ist $k \neq s$ wegen $(\{a\} \to (\{a\} \to a)) \in k - s$ für $a \in A$. Man rechnet nach, daß $(D_A, \cdot) \models \forall x, y, z\, (kxy = x \wedge sxyz = xz(yz))$. Nach 2.4.8 ist $(D_A, \cdot)$ daher kombinatorisch vollständig.

Außerdem ist für alle $a, b \in D_A$ und jedes $m \in G(A)$

$$\begin{aligned} m \in a \cdot b &\iff \exists \alpha^{\text{endlich}} \subseteq a\ \exists \beta^{\text{endlich}} \subseteq b\ m \in \alpha \cdot \beta \\ &\iff \exists \alpha^{\text{endlich}} \subseteq a\ \exists \beta^{\text{endlich}} \subseteq b\ (\alpha \to (\beta \to m)) \in e \\ &\iff \exists \beta^{\text{endlich}} \subseteq b\ (\beta \to m) \in e \cdot a \\ &\iff m \in e \cdot a \cdot b. \end{aligned}$$

Daher ist $ea \sim a$. Ebenso rechnet man nach, daß

$$ea = \{\, (\beta \to m) \mid \beta \subseteq G(A) \text{ endlich},\ m \in a \cdot \beta \,\} = eb$$

für $a \sim b$. Also ist $(D_A, \cdot, e)$ ein Kombinatorenmodell. □

$(D_A, \cdot, e)$ heißt *Graphen*modell, da jedes $a \in D_A$ mit seinen Elementen $(\beta \to m) \in a - A$ den Graphen der durch a repräsentierten Funktion $\lambda b \in D_A\ a \cdot b$ bzw. seiner Einschränkung $\lambda\beta^{\text{endlich}} \subseteq G(A)\ a \cdot \beta$ kodiert.

Nach Satz 2.4.10 erhalten wir aus $(D, \cdot, ee)$ Modelle des typfreien λ-Kalküls. (Ist $ee = e$?)

2.6 Interpretationen von Typen in λ–Modellen

Ein Modell $D = (D, \cdot, \epsilon)$ des typfreien λ–Kalküls unterscheidet nicht zwischen Elementen verschiedenen Typs, z.B. Daten und Funktionen. Wir können aber die Elemente von D als Repräsentanten von Funktionen verschiedenen Typs auffassen und entsprechend eine Klassifizierung in Typen einführen:

Definition 2.6.1 Sei $D = (D, \cdot, \epsilon)$ ein λ–Modell, und $A, B \subseteq D$. Dann sei

$$(A \to_{simple} B) := \{d \in D \mid d \cdot A \subseteq B\}, \qquad (A \to_F B) := \epsilon \cdot (A \to_{simple} B).$$

Von jeder vernünftigen Interpretation $(A \to B)$ des Funktionenraumes erwarten wir

$$(A \to_F B) \ \subseteq\ (A \to B) \ \subseteq\ (A \to_{simple} B).$$

Die „F–Semantik“ akzeptiert nur die Funktionselemente $F = \{\epsilon \cdot d \mid d \in D\}$ als Funktion; beachte

$$(A \to_F B) = F \cap (A \to_{simple} B), \qquad \text{und} \qquad \epsilon \cdot d = [\![\lambda x.yx]\!]^{\mathcal{D}}[d/y],$$

unabhängig von d.

Literatur

[1] Henk Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North Holland, revised edition, 1984.

[2] Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* London Mathematical Sciences Student Texts 1. Cambridge University Press, Cambridge, 1986.

[3] Albert Meyer. What is a model of the λ-calculus? *Information and Control*, 52:87–122, 1982.

[4] Dana S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, 1976.

Kapitel 3

Reduktion und Typisierbarkeit

In einem Modell $(D, \cdot, \epsilon)$ des ungetypten λ-Kalküls repräsentiert ein Element $d \in D$ stets eine *totale* Funktion, da die Applikation $\cdot$ als total vorausgesetzt wird. Normalerweise interessiert man sich aber für Funktionen $f : A \to B$ mit bestimmtem Argumentbereich A und Wertebereich B. Der λ-Kalkül als eine allgemeine Theorie des Funktionsbegriffs sollte diesen Fall umfassen. Dazu möchte man A und B als Teilmengen eines Modells $(D, \cdot, \epsilon)$ auffassen und dann die Elemente $d \in D$, die Funktionen von A nach B repräsentieren,

$$(A \to B) := \{\, d \in D \mid \forall a \in A \; d \cdot a \in B \,\},$$

aussondern (vgl. Abschnitt 2.6. Wie sich die repräsentiere Funktion auf Elementen außerhalb von A verhält, interessiert nicht weiter.) Geht man von gewissen Grundmengen $A \subseteq D$, so erhält man mit den Bereichen $(A \to B)$ eine Klassifizierung der Elemente von D nach ihrem funktionalen Verhalten: eine Einteilung in *semantische Typen*.

Nun liegt es nahe, Eigenschaften des funktionalen Verhaltens an den Termen ablesen zu wollen, die die Elemente bezeichnen. Dazu führt man Typausdrücke oder *syntaktische Typen* ein. Solange man mit den Typen nicht wie mit Elementen rechnen möchte, ist die Bedeutung eines Typausdrucks ein semantischer Typ.[1]

Es gibt nun zwei unterschiedliche Wege, syntaktische Typen zu verwenden:

(i) Man versieht die Individuenterme, insbesondere die Individuenvariablen, explizit mit Typausdrücken (A. Church, 'getypter λ-Kalkül').

(ii) Man benutzt ungetypte λ-Terme wie bisher und beweist –aus Typannahmen für die freien Variablen–, daß ein Term einen Typ hat. (H.B. Curry)

Wir werden nur die zweite Auffassung weiter verfolgen. Dabei stellen sich verschiedene Probleme:

(i) Das Erkennungsproblem: Ist zu gegebenem Term t, gegebenen Typannahmen Γ für seine freien Variablen und gegebenem Typausdruck τ feststellbar, ob t unter Γ den Typ τ hat?

[1] Wenn man mit Typausdrücken rechnet, interpretiert man sie durch Elemente, die semantische Typen, also Mengen von Elementen, nur repräsentieren.

(ii) Das Typsyntheseproblem: Kann man zu einem gegebenen Term t und einer gegebenen Menge Γ von Typannahmen für seine freien Variablen einen Typausdruck τ angeben, so daß t unter Γ den Typ τ hat ?

(iii) Das Typisierbarkeitsproblem: Ist ein gegebener Term t typisierbar, d.h. gibt es Typannahmen Γ für seine freien Variablen und einen Typ τ, so daß t unter Γ den Typ τ hat?

(iv) Das Leerheitsproblem: Ist zu gegebenem Typausdruck τ feststellbar, ob es einen (geschlossenen) Term t gibt, der den Typ τ hat?

Diese Probleme stellen sich für verschiedene Typbegriffe, d.h. verschiedene Anforderungen daran, wie die Typausdrücke der Teilterme eines Terms unter einander und mit der Form des Terms zusammenhängen.

Das Typisierbarkeitsproblem ist die Frage, ob ein Term des typfreien Kalküls aus einem Term des getypten λ-Kalküls durch Weglassen der Typinformation entsteht. Die Typsynthese wird in der Literatur auch oft Typinferenz oder Typrekonstruktion genannt; wir werden die Version behandeln, bei der man die Annahmen Γ spezialisieren darf. Das Leerheitsproblem werden wir im Folgenden nicht betrachten.

Nützliche Typbegriffe schließen aus dem Bereich der ungetypten λ–Terme unerwünschte Terme – etwa Ω– als untypisierbar aus. In diesem Kapitel charakterisieren wir zwei wichtige Mengen von λ-Termen durch Typisierbarkeit: sowohl die Menge der Terme, von denen nur endliche Reduktionsfolgen ausgehen, als auch die Menge der Terme, von denen mindestens eine endliche maximale Reduktionsfolge ausgeht, ist genau die Menge aller typisierbaren Terme für einen geeigneten Typbegriff.

3.1 Typisierungen mit Funktions– und Durchschnittstypen

Wir betrachten hier zwei Typbegriffe, die sich in den erlaubten Typausdrücken und den Mitteln unterscheiden, mit denen man Typaussagen beweisen kann.

Ist t ein λ–Term und τ ein Typausdruck, so ist $t : \tau$ eine *Typaussage*. Ein *Typkontext* Γ ist eine endliche Liste $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ von Typaussagen, in denen die beteiligten Terme paarweise verschiedene Variablen sind. Eine *Typisierung* $\Gamma \rhd t : \tau$ (oder eine *bedingte Typaussage*) ist eine Typaussage $t : \tau$ mit einem Typkontext Γ, der für jede freie Variable von t eine Annahme enthält.

Definition 3.1.1 (H.B. Curry[2]) Die C-Typisierung hat die durch

$$\begin{array}{lrcl} \text{(Typkonstante)} & \iota & := & \iota_1 \mid \iota_2 \mid \ldots \\ \text{(Typausdrücke)} & \tau & := & \iota \mid (\tau_1 \to \tau_2) \end{array}$$

definierten Typausdrücke (oft kurz: Typen) und den Typisierungskalkül mit folgenden Axiomen

und Regeln, wobei t, s λ–Terme, x, y Variable, σ, τ Typausdrücke sind und Γ ein Typkontext ist:

$$(\text{V } 1) \quad \Gamma, x : \tau \rhd x : \tau, \qquad \text{falls } x \text{ nicht in } \Gamma \text{ vorkommt}$$

$$(\text{V } 2) \quad \frac{\Gamma \rhd x : \tau}{\Gamma, y : \sigma \rhd x : \tau}, \qquad \text{falls } y \text{ nicht in } \Gamma \text{ vorkommt}$$

$$(\to I) \quad \frac{\Gamma, x : \sigma \rhd t : \tau}{\Gamma \rhd \lambda x.t : \sigma \to \tau}, \qquad \text{falls } x \text{ nicht in } \Gamma \text{ vorkommt}$$

$$(\to E) \quad \frac{\Gamma \rhd t : \sigma \to \tau, \qquad \Gamma \rhd s : \sigma}{\Gamma \rhd (t \cdot s) : \tau}$$

Die Typen der C-Typisierung heißen auch *einfache Typen*. (In Kaptitel 4 werden wir dabei auch Typvariable zulassen.) Man erweitert sie zu sogenannten *Durchschnittstypen* wie folgt.

Definition 3.1.2 (M. Coppo und M. Dezani-Ciancalini[1]) Die D-Typisierung hat die durch

$$\begin{array}{lrcccccc} \text{(Typkonstante)} & \iota & := & \iota_1 & | & \iota_2 & | & \ldots \\ \text{(Typausdrücke)} & \tau & := & \iota & | & (\tau_1 \to \tau_2) & | & (\tau_1 \wedge \tau_2) \end{array}$$

definierten Typausdrücke und den Typisierungskalkül mit folgenden Axiomen und Regeln:

$$(\text{V } 1) \quad \Gamma, x : \tau \rhd x : \tau, \qquad \text{falls } x \text{ nicht in } \Gamma \text{ vorkommt}$$

$$(\text{V } 2) \quad \frac{\Gamma \rhd x : \tau}{\Gamma, y : \sigma \rhd x : \tau}, \qquad \text{falls } y \text{ nicht in } \Gamma \text{ vorkommt}$$

$$(\to I) \quad \frac{\Gamma, x : \sigma \rhd t : \tau}{\Gamma \rhd \lambda x.t : \sigma \to \tau}, \qquad \text{falls } x \text{ nicht in } \Gamma \text{ vorkommt}$$

$$(\to E) \quad \frac{\Gamma \rhd t : \sigma \to \tau \qquad \Gamma \rhd s : \sigma}{\Gamma \rhd (t \cdot s) : \tau}$$

$$(\wedge I) \quad \frac{\Gamma \rhd t : \sigma \qquad \Gamma \rhd t : \tau}{\Gamma \rhd t : (\sigma \wedge \tau)}$$

$$(\wedge E_l) \quad \frac{\Gamma \rhd t : (\sigma \wedge \tau)}{\Gamma \rhd t : \sigma} \qquad\qquad (\wedge E_r) \quad \frac{\Gamma \rhd t : (\sigma \wedge \tau)}{\Gamma \rhd t : \tau}$$

Eine Folge von Anwendungen der Regeln (V 1) und (V 2) fassen wir oft zusammen in der Form:

$$(\mathit{Var}) \quad \Gamma, x : \tau, \Delta \rhd x : \tau, \qquad \text{falls keine Typannahme für } x \text{ in } \Gamma, \Delta \text{ vorkommt}$$

Konvention: Wir betrachten nur Terme, deren gebundene Variablen paarweise verschieden sind und nicht unter ihren freien Variablen vorkommen. (Bei der syntaktischen Analyse eines Programms wird dies durch Einführen neuer Namen sichergestellt.) Wo mehrere voneinander unabhängige Terme auftreten, nehmen wir an, daß sie disjunkte Teilterme eines zu typisierenden 'globalen' Terms, sind.

Konvention: In Typisierungen $\Gamma \rhd t : \tau$ enthalte Γ keine Annahmen für gebundene Variablen von t. Dann ist die Nebenbedingung in $(\to I)$ überflüssig.

Falls Konstante in den λ–Termen vorkommen, sollen diese wie Variablen durch Annahmen im Kontext Γ getypt sein.

Im Folgenden steht K für einen der Kalküle C oder D mit den zugehörigen Typausdrücken. Ist die Typisierung $\Gamma \rhd t : \tau$ im Kalkül K herleitbar, schreiben wir $\Gamma \vdash_K t : \tau$.

Proposition 3.1.3 Folgende Strukturregeln sind herleitbar, d.h. führen von K-beweisbaren Obersequenzen zu K–beweisbaren Untersequenzen:

$$(W\ 1)\quad \frac{\Gamma, x : \rho, \Delta \rhd s : \sigma}{\Gamma, \Delta \rhd s : \sigma}, \qquad \text{falls } x \text{ nicht frei in } s \text{ vorkommt}$$

$$(W\ 2)\quad \frac{\Gamma, \Delta \rhd s : \sigma}{\Gamma, x : \rho, \Delta \rhd s : \sigma}, \qquad \text{falls } x \text{ nicht in } \Gamma, \Delta, s \text{ vorkommt}$$

$$(W\ 3)\quad \frac{\Gamma, x : \rho, y : \sigma, \Delta \rhd t : \tau}{\Gamma, y : \sigma, x : \rho, \Delta \rhd t : \tau}.$$

Beweis: Wir überlassen den Nachweis der anderen Behauptungen dem Leser als Übung und zeigen nur $(W\ 2)$ durch Induktion über die Herleitung von $\Gamma, \Delta \rhd s : \sigma$. Alle Fälle sind offensichtlich, bis auf den, wo die Herleitung in

$$(\to I)\ \frac{\begin{array}{c}\vdots\\ \Gamma, \Delta, y : \sigma \rhd t : \tau\end{array}}{\Gamma, \Delta \rhd \lambda y t : \sigma \to \tau}$$

endet. Nach Voraussetzung kommt x nicht in Γ, Δ und $\lambda y t$ vor, und nach Konvention kommt y nicht in Γ, Δ vor. Dann kommt x auch nicht frei in t vor, so daß nach Induktionsannahme schon

$$\Gamma, x : \rho, \Delta, y : \sigma \vdash_K t : \tau$$

gilt. Die Behauptung folgt wieder mit einer Anwendung von $(\to I)$. □

Lemma 3.1.4 Keine in t gebundene Variable komme frei in s vor. Sind $\Gamma, x : \sigma \rhd t : \tau$ und $\Gamma \rhd s : \sigma$ K-beweisbar, so auch $\Gamma \rhd [s/x]t : \tau$.

Beweis: durch Induktion über die Herleitung von $\Gamma, x : \sigma \triangleright t : \tau$. Unterscheide nach der zuletzt angewendeten Regel.

Fall 1: Die Herleitung endet mit einer Anwendung von (*Var*). Wegen $t : \tau \equiv x : \sigma$ ist $[s/x]t : \tau \equiv s : \sigma$, also nichts zu zeigen.

Fall 2: Die Herleitung endet in

$$(\rightarrow I)\ \frac{\begin{array}{c}\vdots\\ \Gamma, x : \sigma, y : \tau_1 \triangleright r : \tau_2\end{array}}{\Gamma, x : \sigma \triangleright \lambda yr : \tau_1 \rightarrow \tau_2}.$$

Falls x nicht frei in $t \equiv \lambda yr$ vorkommt, so ist $[s/x]t \equiv t$, und nach 3.1.3 dann $\Gamma \vdash_K [s/x]t : \tau$. Angenommen, x komme frei in $t \equiv \lambda yr$ vor. Dann ist x frei in r und $x \not\equiv y$. Da keine in t gebundene Variable frei in s vorkommt, gilt $[s/x]\lambda yr \equiv \lambda y[s/x]r$. Nach Proposition 3.1.3 ist $\Gamma, y : \tau_1, x : \sigma \vdash_K r : \tau_2$. Gemäß der Induktionsannahme gilt also $\Gamma, y : \tau_1 \vdash_K [s/x]r : \tau_2$, woraus mit $(\rightarrow I)$ folgt, daß $\Gamma \vdash_K \lambda y[s/x]r \equiv [s/x]\lambda yr : (\tau_1 \rightarrow \tau_2)$.

Fall 3: Die Herleitung endet mit einer Anwendung von $(\wedge I)$, $(\wedge E_j)$, oder $(\rightarrow E)$. Die Behauptung folgt leicht aus der Induktionsannahme. □

Bei C- und D-Typisierung ist jeder Teilterm eines typisierbaren Terms ebenfalls typisierbar:

Proposition 3.1.5 Ist $\Gamma \vdash_K t : \tau$ und s ein Teilterm von t, so gibt es Annahmen Δ und einen Typ σ mit $\Gamma, \Delta \vdash_K s : \sigma$.

Das sieht man leicht durch Induktion über t. Für verschiedene Vorkommen von s in t braucht man im allgemeinen verschiedene Erweiterungen Δ, in denen die Typannahmen für auf dem Pfad von der Wurzel von t zum Vorkommen von s gebundene Variablen gesammelt werden.

Definition 3.1.6 Für λ-Terme s und t und einen Typisierungskalkül K sei $s \subseteq_K t$ genau dann, wenn für alle Typen σ und Kontexte Γ gilt:

$$\Gamma \vdash_K s : \sigma \quad \text{impliziert} \quad \Gamma \vdash_K t : \sigma.$$

Aus der Form der Regeln ergibt sich für $K \in \{C, D\}$ leicht folgende Monotonie der Typisierbarkeit:

Proposition 3.1.7 Ist $s \subseteq_K t$, so auch auch $(fs) \subseteq_K (ft)$, $(se) \subseteq_K (te)$ und $\lambda x.s \subseteq_K \lambda x.t$.

Nicht ganz so offensichtlich ist

Lemma 3.1.8 $(\lambda x.t)s \subseteq_K [s/x]t$.

Beweis: Durch Induktion über die Herleitung von $\Gamma \rhd (\lambda x.t)s : \tau$.

Fall 1: Die letzte Regelanwendung war $(\to I)$ oder (*Var*). Wegen der Form des betrachteten Terms ist das unmöglich.

Fall 2: Die letzte Regelanwendung ist $(\to E)$: Dann haben wir einen Beweis der Form

$$\dfrac{\dfrac{\vdots}{\Gamma, x : \sigma \rhd t : \tau} \qquad \dfrac{\vdots}{\Gamma \rhd s : \sigma}}{\Gamma \rhd \lambda x t : \sigma \to \tau} \qquad \Gamma \rhd \lambda x t \cdot s : \tau.$$

Nach Umbenennung gebundener Variablen kommt keine in t gebundene Variable frei in s vor. Nach Lemma 3.1.4 ist $\Gamma \rhd [s/x]t : \tau$ beweisbar.

Fall 3: Die letzte Regelanwendung war $(\wedge E_r)$, $(\wedge E_l)$oder $(\wedge I)$: Die Behauptung ist klar nach Induktion. □

Es folgt, daß Typisierungen unter Reduktionen erhalten bleiben:

Korollar 3.1.9 $\lambda\beta(\eta) \vdash s \to t \quad \Rightarrow \quad s \subseteq_K t.$

Die Umkehrung von Lemma 3.1.8 gilt nicht für den C-Kalkül: es kann sein, daß die Kontraktion eines Redexes C-Typisierungen erlaubt, die der Redex selbst nicht erlaubt. Mit anderen Worten, es kann C-untypisierbare Programme geben, deren Anwendung aber C-typisierbare Werte liefert.

Beispiel 3.1.10 Für $I = \lambda x.x$ ist $\lambda\beta \vdash (\lambda x.xx)I \to I$. Eine Herleitung für C-Typisierungen von $(\lambda x.xx)I$ müsste so aussehen:

$$\dfrac{\dfrac{\dfrac{\vdots\ \text{a)}}{\Gamma, x : \sigma \rhd x : \rho \to \tau} \qquad \dfrac{\vdots\ \text{b)}}{\Gamma, x : \sigma \rhd x : \rho}}{\dfrac{\Gamma, x : \sigma \rhd xx : \tau}{\Gamma \rhd \lambda x.xx : \sigma \to \tau}} \qquad \dfrac{\vdots\ \text{c)}}{\Gamma \rhd I : \sigma}}{\Gamma \rhd (\lambda x.xx) \cdot I : \tau}$$

Dabei müssten a) und b) Anwendungen von (*Var*) sein, also $\sigma \equiv \rho \to \tau$ und $\sigma \equiv \rho$, was unmöglich ist. Anererseits ist $\Gamma \vdash_C I : \rho \to \rho$ für jedes ρ.

Für den D-Kalkül gilt aber auch die folgende Umkehrung von Lemma 3.1.8. Auf die Einschränkung, daß x in t frei vorkommt, kann dabei nicht verzichtet werden, da es sonst eine Typisiserung $\Gamma \vdash_D [s/x]t : \tau$ geben könnte, auch wenn s (bezüglich Γ) nicht typisiserbar wäre.

Lemma 3.1.11 Falls x in t frei vorkommt, so gilt $[s/x]t \subseteq_D \lambda x t \cdot s$.

Beweis: Sei $\Gamma \rhd [s/x]t : \tau$ D–beweisbar, und seien

$$\begin{array}{ccc} \vdots & & \vdots \\ \Gamma, \Delta_1 \rhd s : \sigma_1, & \cdots, & \Gamma, \Delta_n \rhd s : \sigma_n \end{array} \tag{3.1}$$

nach 3.1.5 die maximalen Teilbeweise der Typisierungen von s in einem Beweis für $\Gamma \rhd [s/x]t : \sigma$. Da x in t vorkommt, ist $n \geq 1$ und für jede freie Variable von s hat Γ eine Typannahme. Nach 3.1.3 ist daher jedes $\Gamma \rhd s : \sigma_i$ beweisbar. Für $\sigma = \sigma_1 \wedge \ldots \wedge \sigma_n$ haben wir dann folgenden Beweis:

$$\dfrac{\dfrac{\begin{array}{ccc} \vdots & & \vdots \\ \Gamma \rhd s : \sigma_1, & \cdots & , \Gamma \rhd s : \sigma_n \end{array}}{\Gamma \rhd s : \sigma}\,(\wedge E_r), (\wedge E_l) \qquad \dfrac{\begin{array}{c} \vdots \\ \Gamma, x : \sigma \rhd t : \tau \end{array}}{\Gamma \rhd \lambda x t : \sigma \to \tau}}{\Gamma \rhd \lambda x t \cdot s : \tau};$$

der rechte Teilbeweis hierbei entsteht aus dem Beweis von $\Gamma \rhd [s/x]t : \sigma$, indem man die Teilbeweise aus (3.1) durch die entsprechenden Beweise

$$(\wedge\ E)'s \quad \dfrac{\Gamma, x : \sigma, \Delta_i \rhd x : \sigma}{\Gamma, x : \sigma, \Delta_i \rhd x : \sigma_i}$$

ersetzt. □

3.2 Abschwächungen von D-Typisierungen

Nach den Regeln $(\wedge E_l)$ und $(\wedge E_r)$ ist jeder Term vom Typ $(\sigma \wedge \tau)$ auch vom Typ σ und vom Typ τ, d.h. $(\sigma \wedge \tau)$ ist *stärker* als σ und τ. Neben der Abschwächung durch Hinzunahme weiterer Typannahmen kann man daher Typisierungen im D-Kalkül auch dadurch abschwächen, daß man zu stärkeren Typen in den Annahmen oder einem schwächeren Typ in der hergeleiteten Typaussage übergeht.

Wir wollen zeigen, daß bedingte Typaussagen unter diesen Abschwächungen gültig bleiben. Dazu betrachten wir eine partielle Ordnung $\sigma \leq_D \tau$ zwischen Typausdrücken, die auf Eigenschaften der Durchschnittbildung beruht:

Definition 3.2.1 Sei $\hat{=}$ die kleinste Äquivalenzrelation auf Typausdrücken, die die Idempotenz, Kommutativität und Assoziativität von $\wedge$ umfaßt, d.h. es ist $\tau_1 \hat{=} \tau_2$, falls dies mit folgenden Axiomen und Regeln herleitbar ist:

$$(\wedge \text{ Idem}) \quad \sigma \hat{=} (\sigma \wedge \sigma) \qquad (\wedge \text{ Kom}) \quad (\sigma \wedge \tau) \hat{=} (\tau \wedge \sigma)$$

$$(\wedge \text{ Assoc}) \quad \rho \wedge (\sigma \wedge \tau) \hat{=} (\rho \wedge \sigma) \wedge \tau \qquad (\hat{=} \text{ Refl}) \quad \sigma \hat{=} \sigma$$

$$(\hat{=} \text{ Sym}) \quad \dfrac{\sigma \hat{=} \tau}{\tau \hat{=} \sigma} \qquad (\hat{=} \text{ Trans}) \quad \dfrac{\sigma \hat{=} \tau, \qquad \tau \hat{=} \rho}{\sigma \hat{=} \rho}$$

Wir schreiben $\sigma \leq_D \tau$, falls $(\sigma \wedge \tau) \hat{=} \sigma$ beweisbar ist.

Proposition 3.2.2 Ist $\sigma \hat{=} \tau$, so ist $\Gamma \vdash_D s : \sigma$ genau dann der Fall, wenn $\Gamma \vdash_D s : \tau$.

Beweis: Durch Induktion über die Herleitung von $\sigma \hat{=} \tau$.

Fall 1: Die Herleitung besteht aus der Anwendung eines Axioms. Aus $(\wedge\ E)$ und $(\wedge I)$ folgt offenbar:

$$\begin{array}{lcl} \Gamma \vdash_D s : \sigma & \iff & \Gamma \vdash_D s : \sigma \wedge \sigma \\ \Gamma \vdash_D s : \sigma \wedge \tau & \iff & \Gamma \vdash_D s : \tau \wedge \sigma \\ \Gamma \vdash_D s : \varrho \wedge (\sigma \wedge \tau) & \iff & \Gamma \vdash_D s : (\varrho \wedge \sigma) \wedge \tau. \end{array}$$

Fall 2: Der letzte Herleitungsschritt ist eine Regelanwendung. Für ($\hat{=}$ Sym) und ($\hat{=}$ Trans) ist die Behauptung offensichtlich. □

Beachte, daß $\hat{=}$ keine Kongruenzrelation auf Typausdrücken ist: aus $\sigma_1 \hat{=} \sigma_2$ und $\tau_1 \hat{=} \tau_2$ folgt nicht, daß $(\sigma_1 \to \tau_1) \hat{=} (\sigma_2 \to \tau_2)$ ist, oder daß $x : \sigma_1 \to \tau_1 \vdash_D x : \sigma_2 \to \tau_2$.

Definition 3.2.3 Für Typkontexte $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ und $\Delta = x_1 : \tau_1, \ldots, x_n : \tau_n$ mit derselben Variablenfolge sei

$$\Gamma \wedge \Delta \quad := \quad x_1 : (\sigma_1 \wedge \tau_1), \ldots, x_n : (\sigma_n \wedge \tau_n),$$

$$\Gamma \leq_D \Delta \quad :\Leftrightarrow \quad \text{für jedes } i = 1, \ldots, n \text{ ist } \sigma_i \leq_D \tau_i,$$

Durch $\leq_D$ wird eine partielle Ordnung auf Typausdrücken definiert, und offenbar ist $\Gamma \wedge \Delta \leq_D \Gamma$.

Nun sieht man leicht, daß D-Typisierungen unter Verstärkung von Annahmetypen oder Abschwächung des Ergebnistyps erhalten bleiben:

Lemma 3.2.4 $\Delta \vdash_D s : \sigma, \quad \Gamma \leq_D \Delta, \quad \sigma \leq_D \tau \qquad \Rightarrow \qquad \Gamma \vdash_D s : \tau.$

Beweis: Aus $\sigma \leq_D \tau$ folgt mit Proposition 3.2.2 und einer Anwendung von $(\wedge\ E)$, daß

$$\Gamma \vdash_D s : \sigma \qquad \Rightarrow \qquad \Gamma \vdash_D \sigma \wedge \tau \qquad \Rightarrow \qquad \Gamma \vdash_D s : \tau.$$

Es genügt also, durch Induktion über die Herleitung von $\Delta \triangleright s : \sigma$ zu zeigen:

$$\Delta \vdash_D s : \sigma, \quad \Gamma \leq_D \Delta \qquad \Rightarrow \qquad \Gamma \vdash_D s : \sigma.$$

Sei dazu $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$ und $\Delta \equiv x_1 : \tau_1, \ldots, x_n : \tau_n$.

Fall 1: Die Herleitung von $\Delta \triangleright s : \sigma$ endet in einer Anwendung von (*Var*). Dann ist $\Delta \triangleright x : \sigma \equiv \Delta \triangleright x_i : \tau_i$ für ein $i \in \{1, \ldots, n\}$. Aus $\Gamma \vdash_D x_i : \sigma_i$ und $\sigma_i \leq_D \tau_i$ folgt $\Gamma \vdash_D x_i : (\sigma_i \wedge \tau_i)$ nach Proposition 3.2.2, also mit einer Anwendung von $(\wedge\ E)$ auch $\Gamma \vdash_D x_i : \tau_i$.

Fall 2: Die Herleitung von $\Delta \rhd s : \sigma$ endet mit einer Anwendung von $(\to I)$. Es gibt zu $s : \sigma \equiv \lambda xt : \rho \to \tau$ eine Herleitung der Form

$$\frac{\begin{array}{c}\vdots\\ \Delta, x : \rho \rhd t : \tau\end{array}}{\Delta \rhd \lambda xt : \rho \to \tau}.$$

Da $\Gamma, x : \rho \ \leq_D \ \Delta, x : \rho$ ist, ist nach Induktionsannahme $\Gamma, x : \rho \rhd t : \tau$ schon D–beweisbar, also mit einer Anwendung von $(\to I)$ auch $\Gamma \rhd \lambda xt : \rho \to \tau$.

Fall 3: Die Herleitung endet mit einer Anwendung von $(\to E)$, $(\wedge I)$, oder $(\wedge E)$: Für diese Fälle zeigt man die Behauptung ebenso direkt aus der Induktionsannahme. □

Damit kommen wir zur Charakterisierung von Termmengen durch Typsisierbarkeit.

3.3 Stark normalisierbare Terme sind D–typisierbar

Definition 3.3.1 Ein Term in β–Normalform heißt *normal*. Ein Term t heißt *normalisierbar*, wenn es eine Reduktion von t zu einem normalen Term gibt. Er heißt *stark normalisierbar*, wenn jede mit t beginnende Reduktionsfolge (bis auf α–Reduktionen) endlich ist.

Lemma 3.3.2 Zu jedem normalen Term e mit $\mathit{frei}(e) \subseteq \{x_1, \ldots, x_n\}$ gibt es eine D–Typsierung $\Gamma \vdash_D e : \sigma$, bei der Γ aus Annahmen für die Folge $x_1, \ldots, x_n$ besteht. Falls $e \not\equiv \lambda xt$ ist, können wir dabei σ vorgeben.

Beweis: Durch Induktion über den Aufbau von e.

Fall $e \equiv x_i$: Nach (Var) ist $\Gamma \vdash_D x_i : \sigma$.

Fall $e \equiv \lambda x_{n+1}.t$: Nach Induktionsvoraussetzung gibt es einen Beweis für $\Gamma, x_{n+1} : \sigma \rhd t : \tau$ mit geeigneten Γ, σ, und τ. Mit $(\to I)$ setzt man ihn zu einem Beweis für $\Gamma \rhd \lambda x_{n+1}.t : \sigma \to \tau$ fort.

Fall $e \equiv (f \cdot s)$: Da e normal ist, ist $f \not\equiv \lambda xt$. Daher gibt es nach Induktion eine Typisierung $\Delta \vdash_D s : \sigma$, und zu vorgegebenem Typ $\sigma \to \tau$ eine Typisierung $\Gamma \vdash_D f : \sigma \to \tau$. Nach 3.2.4 erhalten wir daraus durch Abschwächungen einen Beweis

$$(\to E)\ \frac{\begin{array}{c}\vdots\\ \Gamma \wedge \Delta \rhd f : \sigma \to \tau\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \wedge \Delta \rhd s : \sigma,\end{array}}{\Gamma \wedge \Delta \rhd (f \cdot s) : \tau}.$$

Beachte, daß Γ und Δ nur aus Annahmen für die Folge $x_1, \ldots, x_n$ bestehen. □

Satz 3.3.3 Ist e stark normalisierbar, so ist e D–typisierbar.

Beweis: Durch Induktion über die maximale Länge $|e|$ von Reduktionsfolgen, die in e beginnen. Beachte, daß $|e|$ nach Zorns Lemma existiert, da jeder Term in einem Reduktionsschritt nur zu endlich vielen Termen führt (modulo α–Konversion).

Fall $|e| = 0$: Dann ist e normal, also D–typisierbar nach Lemma 3.3.2.

Fall $|e| > 0$: Sei $(\lambda xt \cdot s)$ ein Redex von e, und e' entstehe aus e durch dessen Kontraktion. Wegen $|e'| < |e|$ gibt es Γ und ρ mit $\Gamma \vdash_D e' : \varrho$. Sei nach 3.1.5

$$\begin{array}{c}\vdots\\ \Gamma, \Delta' \rhd [s/x]t : \tau\end{array} \tag{3.2}$$

eine Teilherleitung davon, wobei Δ' die zum Vorkommen von $[s/x]t$ in e' gehörende Kontexterweiterung ist.

(a) x kommt in t vor. Nach Lemma 3.1.11 ist dann auch $\Gamma, \Delta' \vdash_D (\lambda xt \cdot s) : \tau$. Setze die Herleitung davon in den Beweis von $\Gamma \rhd e' : \rho$ für die Teilherleitung (3.2) ein. Das ergibt einen D–Beweis für $\Gamma \rhd e : \varrho$.

(b) x kommt nicht in t vor. Wegen $|s| < |e|$ gibt es Δ und σ mit $\Delta \vdash_D s : \sigma$. Wir können mit 3.3.2 annehmen, daß Γ und Δ Annahmen für dieselbe Folge von Variablen enthalten. Durch Abschwächung nach 3.2.4 erhält man D-Beweise für $\Gamma \wedge \Delta \rhd s : \sigma$ und $\Gamma \wedge \Delta \rhd e' : \varrho$, und darin eine Teilherleitung

$$\begin{array}{c}\vdots\\ \Gamma \wedge \Delta, \Delta' \rhd [s/x]t : \tau\end{array} \quad \equiv \quad \begin{array}{c}\vdots\\ \Gamma \wedge \Delta, \Delta' \rhd t : \tau.\end{array} \tag{3.3}$$

Weitere Abschwächungen nach 3.1.3 liefern einen Beweis

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \wedge \Delta, \Delta', x : \sigma \rhd t : \tau\end{array}}{\Gamma \wedge \Delta, \Delta' \rhd \lambda xt : \sigma \to \tau} \qquad \begin{array}{c}\vdots\\ \Gamma \wedge \Delta, \Delta' \rhd s : \sigma\end{array}}{\Gamma \wedge \Delta, \Delta' \rhd \lambda xt \cdot s : \tau}.$$

Setzt man diesen für die Teilherleitung (3.3) in den Beweis von $\Gamma \wedge \Delta \rhd e' : \varrho$ ein, so hat man einen Beweis von $\Gamma \wedge \Delta \rhd e : \varrho$. □

Beispiel 3.3.4 Die Selbstanwendung, $\lambda x(xx)$, ist stark normalisierbar, da sie keinen β-Redex enthält. Eine D–Typisierung dafür ist

$$\dfrac{\dfrac{\dfrac{(\sigma \to \tau) \wedge x : \sigma \rhd x : (\sigma \to \tau) \wedge x : \sigma}{x : (\sigma \to \tau) \wedge x : \sigma \rhd x : \sigma \to \tau} \qquad \dfrac{(\sigma \to \tau) \wedge x : \sigma \rhd x : (\sigma \to \tau) \wedge x : \sigma}{x : (\sigma \to \tau) \wedge \sigma \rhd x : \sigma}}{x : (\sigma \to \tau) \wedge \sigma \rhd xx : \tau}}{\rhd \lambda x(xx) : ((\sigma \to \tau) \wedge \sigma) \to \tau}$$

Wie in 3.3.4 gezeigt wurde, ist $\lambda x(xx)$ nicht C-typisierbar.

3.4 D–typisierbare Terme sind stark normalisierbar

Wir interpretieren jetzt D-Typausdrücke durch geeignete Mengen von λ–Termen. Für Mengen A, B von λ–Termen, sei

$$(A \to B) := \{t \mid \text{ für alle } a \in A \text{ ist } ta \in B\}.$$

Beachte, daß $(A \to B)$ monoton in A und anti-monoton in B ist, d.h. für $A' \subseteq A$ und $B \subseteq B'$ ist $(A \to B) \subseteq (A' \to B')$.

Definition 3.4.1 Seien Φ und A Mengen von λ–Termen. A heißt Φ–*saturiert*, falls für alle λ–Terme $\lambda xt, t_1, \ldots, t_n$ und alle $s \in \Phi$ gilt:

$$([s/x]t)t_1 \cdots t_n \in A \qquad \Rightarrow \qquad (\lambda xt \cdot s)t_1 \cdots t_n \in A.$$

Eine D–*Interpretation relativ zu* Φ ist eine Abbildung I, die jedem Basistyp σ eine Φ–saturierte Menge $I(\sigma)$ von Termen zuordnet.

Nach dem folgenden Lemma 3.4.2 kann jede D-Interpretation I durch

$$I(\sigma \wedge \tau) := I(\sigma) \cap I(\tau) \qquad \text{und} \qquad I(\sigma \to \tau) := (I(\sigma) \to I(\tau))$$

so auf auf alle Typausdrücke ausgedehnt werden, daß $I(\sigma)$ für *jedes* σ eine Φ-saturierte Menge ist. Diese Mengen können daher zur Interpretation von Typausdrücken verwendet werden.

Lemma 3.4.2 (i) Sind A und B Φ–saturiert, so auch $A \cap B$.

(ii) Ist B Φ–saturiert, so auch $(A \to B)$.

Beweis: Behauptung i) ist klar. Für ii) seien $s \in \Phi$ und $([s/x]t)t_1 \cdots t_n \in (A \to B)$. Sei $a \in A$, also $([s/x]t)t_1 \cdots t_n a \in B$ nach Definition von $(A \to B)$. Da B Φ–saturiert ist, ist auch

$$(\lambda xt \cdot s)t_1 \cdots t_n a \in B;$$

da $a \in A$ beliebig war, ist $(\lambda xt \cdot s)t_1 \cdots t_n \in (A \to B)$, d.h. $(A \to B)$ ist Φ–saturiert. □

Satz 3.4.3 (Korrektheitssatz)
Sei I eine D–Interpretation relativ zu Φ, so daß $I(\tau) \subseteq \Phi$ für alle τ. Dann gilt:

$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash_D t : \tau \qquad \Rightarrow$
$[t_n/x_n] \ldots [t_1/x_1]t \in I(\tau)$ für alle $t_1 \in I(\sigma_1), \ldots, t_n \in I(\sigma_n)$ mit $x_{i+1}, \ldots, x_n$ nicht frei in t_i.

Beweis: Durch Induktion über die Herleitung von $x_1 : \sigma_1, \ldots, x_n : \sigma_n \rhd t : \tau$. Seien $t_1 \in I(\sigma_1)$, $\ldots, t_n \in I(\sigma_n)$, so daß kein t_j eine Variable aus $\{x_{j+1}, \ldots, x_n\}$ frei enthält. Wir unterscheiden nach der zuletzt angewendeten Regel:

Fall 1: Die Herleitung endet mit einer Anwendung von (*Var*). Dann ist die Herleitung von der Form $x_1 : \sigma_1, \ldots, x_n : \sigma_n \rhd x_i : \sigma_i$. Wegen $x_i : \sigma_i \equiv t : \tau$ ist dann

$$[t_n/x_n] \ldots [t_1/x_1]t \equiv [t_n/x_n] \ldots [t_{i+1}/x_{i+1}][t_i/x_i]x_i \equiv t_i \in I(\sigma_i) = I(\tau).$$

Fall 2: Die Herleitung endet in einer Anwendung von $(\to I)$ mit $t : \tau \equiv \lambda x e : (\sigma \to \varrho)$. Nach Induktion ist für $\Gamma, x : \sigma \vdash_D e : \varrho$ und $s \in I(\sigma)$ schon

$$[s/x]([t_n/x_n] \ldots [t_1/x_1]e) \in I(\varrho).$$

Da $s \in I(\sigma) \subseteq \Phi$, und $I(\varrho)$ Φ–saturiert ist, ist $\lambda x([t_n/x_n] \ldots [t_1/x_1]e) \cdot s \in I(\varrho)$. Da x nicht frei in den t_i vorkommt, folgt

$$[t_n/x_n] \ldots [t_1/x_1]\lambda x.e \in (I(\sigma) \to I(\varrho)) = I(\sigma \to \varrho) = I(\tau).$$

Fall 3: Die Herleitung endet in einer Anwendung von $(\to E)$, mit $t : \tau \equiv (f \cdot s) : \tau$. Nach Induktion ist für entsprechendes σ

$$[t_n/x_n] \ldots [t_1/x_1]f \in I(\sigma \to \tau) \quad \text{und} \quad [t_n/x_n] \ldots [t_1/x_1]s \in I(\sigma),$$

also

$$[t_n/x_n] \ldots [t_1/x_1](f \cdot s) \equiv ([t_n/x_n] \ldots [t_1/x_1]f \cdot [t_n/x_n] \ldots [t_1/x_1]s) \in I(\tau).$$

Fall 4: Die Herleitung endet mit einer Anwendung von $(\wedge E)$ oder $(\wedge I)$. Diese Fälle sind klar, da $I(\sigma_1 \wedge \sigma_2) = I(\sigma_1) \cap I(\sigma_2) \subseteq I(\sigma_i)$. □

Definition 3.4.4 Seien Φ,Θ,Ψ Mengen von λ–Termen. Das Paar (Ψ, Θ) heißt Φ–*adäquat*, falls (i) Ψ ist Φ–saturiert, (ii) $\Theta \subseteq \Psi$, (iii) $\Theta \subseteq (\Psi \to \Theta)$, und (iv) $(\Theta \to \Psi) \subseteq \Psi$.

Lemma 3.4.5 Sei (Ψ, Θ) Φ–adäquat, und $E(\Psi, \Theta) := \{A \mid \Theta \subseteq A \subseteq \Psi, A \text{ ist } \Phi\text{–saturiert}\}$.

$$A,\ B \in E(\Psi, \Theta) \qquad \Rightarrow \qquad A \cap B,\ (A \to B) \in E(\Psi, \Theta).$$

Beweis: Seien A und B aus $E(\Psi, \Theta)$. Nach Lemma 3.4.2 sind $A \cap B$ und $(A \to B)$ Φ–saturiert. Mit 3.4.4 (iii), (iv) und den Monotonieeigenschaften von $(A \to B)$ folgt

$$\Theta \subseteq (\Psi \to \Theta) \subseteq (A \to B) \subseteq (\Theta \to \Psi) \subseteq \Psi,$$

so daß $(A \to B) \in E(\Psi, \Theta)$. □

Korollar 3.4.6 Sei (Ψ, Θ) Φ–adäquat, und I eine D–Interpretation relativ zu Φ, so daß $I(\beta) \in E(\Psi, \Theta)$ für jeden Basistyp β. Dann ist $I(\tau) \in E(\Psi, \Theta)$ für jeden D–Typ τ.

Lemma 3.4.7 Sei $\Theta(\Psi) := \{(xt_1 \cdots t_n) \mid x \in \mathrm{Var},\ n \in I\!N, t_1, \ldots, t_n \in \Psi\}$. und I eine D–Interpretation relativ zu Φ mit $\Theta(\Psi) \subseteq I(\beta) \subseteq \Psi$ für jeden Basistyp β. Ist $(\Psi, \Theta(\Psi))$ Φ–adäquat und $\Psi \subseteq \Phi$, so gilt:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash_D t : \tau \qquad \Rightarrow \qquad t \in I(\tau) \subseteq \Psi.$$

Beweis: Nach Korollar 3.4.6 gilt $\mathrm{Var} \subseteq \Theta(\Psi) \subseteq I(\sigma) \subseteq \Psi$ für jeden D–Typ σ, insbesondere gilt wegen $\Psi \subseteq \Phi$ also $x_i \in I(\sigma_i) \subseteq \Phi$. Aus dem Korrektheitssatz 3.4.3 folgt daher, daß $t \equiv [x_n/x_n] \ldots [x_1/x_1]t \in I(\tau)$. □

Satz 3.4.8 Ist t D–typsierbar, so ist t auch stark normalisierbar.

Beweis: Sei SN die Menge der stark normalisierbaren Terme. Es ist klar, daß $\Theta(SN) \subseteq SN$. Für jeden Basistyp β sei $I(\beta) := \{\, t \mid \Gamma \vdash_D t : \beta \text{ für ein } \Gamma \,\} \cap SN$. Nach Lemma 3.4.7 genügt es zu zeigen, daß $\Theta(SN) \subseteq I(\beta) \subseteq SN$ und daß $(SN, \Theta(SN))$ SN–adäquat ist.

a) $\Theta(SN) \subseteq I(\beta)$: Zu $t_1, \ldots, t_n \in SN$ gibt es nach Satz 3.3.3 Typisierungen $\Delta_i \vdash_D t_i : \tau_i$. Wir können annehmen, daß $\Delta_1 = \ldots = \Delta_n$ ist und $x : \sigma$ darin vorkommt. Sei Γ die Verstärkung zu $x : \sigma \wedge (\tau_1 \to \ldots \to (\tau_n \to \beta) \ldots)$. Dann ist $\Gamma \vdash_D xt_1 \cdots t_n : \beta$, also $xt_1 \cdots t_n \in I(\beta)$.

b) $(SN, \Theta(SN))$ ist SN–adäquat: Wir zeigen die vier Eigenschaften der Definition.

i) SN ist SN–saturiert: Wir müssen für alle $s \in SN$ und alle λ–Terme $t, t_1, \ldots, t_n$ zeigen:

$$([s/x]t)t_1 \cdots t_n \in SN \quad \Rightarrow \quad (\lambda xt \cdot s)t_1 \cdots t_n \in SN.$$

Sei also $([s/x]t)t_1 \cdots t_n$ stark normalisierbar, und angenommen, von $(\lambda xt \cdot s)t_1 \cdots t_n$ gehe eine unendliche Reduktionsfolge aus. Solange darin der Redex am Anfang nicht kontrahiert wird, hat man einen Term der Form $(\lambda xt' \cdot s')t'_1 \cdots t'_n$ mit $t \to^* t'$, $s \to^* s'$ und $t_i \to^* t'_i$ für alle i. Also kann man $([s/x]t)t_1 \cdots t_n$ zu $([s'/x]t')t'_1 \cdots t'_n$ reduzieren. Da $([s/x]t)t_1 \cdots t_n \in SN$ ist, kann es nur endlich viele solcher $s', t', t'_1, \ldots, t'_n$ geben. Sobald man aber den Redex $(\lambda xt' \cdot s')t'_1 \cdots t'_n$ zu $([s'/x]t')t'_1 \cdots t'_n$ kontrahiert, hat man einen Term in SN erreicht, und die Reduktionsfolge muß nach endlich vielen weiteren Schritten abbrechen. Also ist $(\lambda xt \cdot s)t_1 \cdots t_n \in SN$.

ii) $\Theta(SN) \subseteq SN$: Sei $xt_1 \cdots t_n \in \Theta(SN)$. Da jede Reduktion auf einen Term $xt'_1 \cdots t'_n$ mit $t_i \to t'_i$ führt und $t_i \in SN$ ist, muß die Reduktion abbrechen.

iii) $\Theta(SN) \subseteq (SN \to \Theta(SN))$: Ist $(xt_1 \cdots t_n) \in \Theta(SN)$, so sind alle $t_i \in SN$, also ist mit $s \in SN$ auch $(xt_1 \cdots t_n s) \in \Theta(SN)$.

iv) $(\Theta(SN) \to SN) \subseteq SN$: Da $\mathrm{Var} \subseteq \Theta(SN)$, genügt zu zeigen, daß mit $t \in SN$ auch $(t \cdot x) \in SN$ gilt. Das ist aber klar. □

Da alle Typausdrücke und Typisierungsregeln der C-Typisierung auch bei der D-Typisierung erlaubt sind, folgt:

Korollar 3.4.9 Jeder C-typisierbare Term ist stark normalisierbar.

3.5 Normalisierbarkeit und E-Typisierbarkeit

Im vorigen Abschnitt haben wir die starke Normalisierbarkeit durch die Typisierbarkeit im D–Kalkül charakterisiert. Dasselbe kann man für die (schwache) Normalisierbarkeit tun. Wir geben nur die Grundidee dieser Charakterisierung an. (Da Normalisierbarkeit und die Existenz einer terminierenden Reduktionsfolge äquivalent sind, ist Typisierbarkeit von Termen in beiden Fällen unentscheidbar.)

Definition 3.5.1 Der *Typisierungskalkül* E erlaubt neben den Axiomen und Regeln des D-Kalküls noch das Axiom

$$(\omega\ I) \quad \Gamma \rhd t : \omega,$$

wobei ω ein ausgezeichneter Basistyp ist (ohne weitere Typsisierungsregeln).

Zuerst überlegt man sich, daß Lemma 3.1.11 auch für den E–Kalkül gilt, in einer leicht verallgemeinerten Version:

Lemma 3.5.2 $[s/x]t \subseteq_E \lambda xt \cdot s$.

Beweis: Der Beweis von Lemma 3.1.11 wird wie folgt modifiziert. Sei $\Gamma \vdash_E [s/x]t : \tau$, und seien

$$\begin{array}{ccc} \vdots & & \vdots \\ \Gamma, \Delta_1 \rhd s : \sigma_1, & \cdots, & \Gamma, \Delta_n \rhd s : \sigma_n \end{array}$$

die maximalen Teilbeweise der Typisierungen von s in einem Beweis für $\Gamma \rhd [s/x]t : \sigma$. Falls x in t vorkommt ($n > 0$), wählen wir wie in 3.1.11 $\sigma := \sigma_1 \wedge \ldots \wedge \sigma_n$; andernfalls wählen wir $\sigma := \omega$. In beiden Fällen erhalten wir einen E-Beweis der Form

$$\dfrac{\begin{array}{cc} \vdots & \vdots \\ \Gamma \rhd s : \sigma & \dfrac{\Gamma, x : \sigma \rhd t : \tau}{\Gamma \rhd \lambda xt : \sigma \to \tau} \end{array}}{\Gamma \rhd \lambda xt \cdot s : \tau};$$

der rechte Teilbeweis hierbei entsteht aus dem Beweis von $\Gamma \rhd [s/x]t : \tau$, indem man die ($n \geq 0$) Teilbeweise

$$\begin{array}{c} \vdots \\ \Gamma, \Delta_i \rhd s : \sigma_i \end{array}$$

durch die entsprechenden Beweise

$$\dfrac{\Gamma, x : \sigma, \Delta_i \rhd x : \sigma}{\Gamma, x : \sigma, \Delta_i \rhd x : \sigma_i}$$

ersetzt. □

Korollar 3.5.3 (i) $\lambda\beta \vdash s \to t \quad \Rightarrow \quad t \subseteq_E s$,

(ii) $\lambda\beta \vdash s \to \lambda x.t \quad \Rightarrow \quad s$ hat eine E-Typisierung mit einem Typ $\neq \omega$.

Korollar 3.5.4 Wenn t normalisierbar ist, gibt es eine E-beweisbare Typisierung $\Gamma \rhd t : \tau$, in der τ ein D-Typ ist (also ω nicht enthält).

Für die Umkehrung verwendet man eine Variante der Methode aus Abschnitt 3.4.

Satz 3.5.5 Ein Term t ist normalisierbar genau dann, wenn es eine E-beweisbare Typisierung $\Gamma \rhd t : \tau$ gibt, bei der alle in $\Gamma, t : \tau$ auftretenden Typausdrücke D–Typen sind.

Beweis: (Skizze) Sei N die Menge der normalisierbaren Terme. Man zeige zuerst, daß das Paar $(N, \Theta(N))$ Φ–adäquat ist, wenn man für Φ die Menge *aller* λ-Terme nimmt.

Sei I eine E–Interpretation, so daß $I(\beta) = N$ für jeden atomaren Typ β. Für jeden D–Typ τ erhält man

$$\Theta(N) \subseteq I(\tau) \subseteq N$$

nach dem Analogon zu Korollar 3.4.6. Hat man eine Typisierung $\Gamma \rhd t : \tau$ wie angegeben, so erhält man nach dem Beweis des Korrektheitssatzes für D–Interpretationen, daß $t \in I(\tau) \subseteq N$ ist. □

Die einheitliche Beweismethode zur Charakterisierung von starker und schwacher Normalisierbarkeit geht auf J.L. Krivine zurück. Man beachte, daß sie nur nichtleere Mengen als Bedeutung der Typausdrücke liefert, da stets $\mathrm{Var} \subseteq \Theta(\Psi) \subseteq I(\tau) \subseteq \Psi \subseteq \Phi$. Zur Interpretation von polymorphen Typen (siehe Kapitel 5) kann man sie daher nicht direkt verwenden.

Literatur

[1] Mario Coppo and Mariangola Dezani. An extension of the basic functionality theory for the λ–calculus. *Notre Dame Journal of Formal Logic*, 21:685–693, 1980.

[2] H. B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[3] Gerard Huet. Initiation au λ–calcul, 1989. Note de Cours du DEA 'Foncionnalité, Structure de Calcul et Programmation', donné à l'Université Paris VII eu (1987-88) et (1988–89).

Kapitel 4

Entscheidbarkeit der C-Typisierbarkeit

In den Kalkülen C und D kann man für einen typisierbaren λ–Term im allgemeinen unendlich viele Typisierungen herleiten; zum Beispiel gilt

$$\emptyset \vdash_C \lambda x.x : \sigma \to \sigma$$

für jedes σ. Alle im C–Kalkül herleitbaren Typisierungen sind aber Spezialisierungen eines einzigen Schemas, und dieses kann effizient aus dem Term berechnet werden.

4.1 Haupttypen im C–Kalkül

Um Schemata von Typen ausdrücken zu können, erweitern wir die Typen um *Typvariable* als weitere Form von atomaren Typausdrücken. Wir verwenden $\alpha_0, \alpha_1, \ldots$ als Typvariablen.

Definition 4.1.1 Eine Abbildung S: Typausdrücke $\to$ Typausdrücke ist ein *Typhomomorphismus*, falls für alle Basistypen β und alle C–Typen σ und τ gilt:

$$S(\iota) = \iota, \qquad S(\sigma \to \tau) = S(\sigma) \to S(\tau).$$

Ist Γ ein C–Kontext, so ist $S(\Gamma)$ die Liste der Typannahmen $x : S(\sigma)$ mit $x : \sigma$ in Γ.

Wir schreiben im Folgenden oft $S\sigma$ statt $S(\sigma)$ und $S\Gamma$ statt $S(\Gamma)$. Jede *Typsubstitution* S: Typvariable $\to$ Typausdrücke kann in eindeutiger Weise zu einem Typhomomorphismus fortgesetzt werden.

Proposition 4.1.2 Für jeden Typhomomorphismus S gilt:

$$\Gamma \vdash_C t : \sigma \qquad \Rightarrow \qquad S\Gamma \vdash_C e : S\sigma.$$

Beweis: Durch Induktion über die Ableitung. □

Definition 4.1.3 Eine Typisierung $\Gamma \triangleright e : \sigma$ heißt *allgemeiner* als $\Delta \triangleright e : \tau$, falls es eine Typsubstitution S gibt mit $\tau = S(\sigma)$ und $\Delta \| var(\Gamma) = S(\Gamma)$. $\Delta \triangleright e : \tau$ ist eine *C–Typisierung* von e *modulo* Γ, falls $\Delta \vdash_C e : \tau$ und Δ eine Instanz von Γ ist, d.h. $\Delta \| var(\Gamma) = S(\Gamma)$ für eine Substitution S. Eine *C–Haupttypisierung* von e *modulo* Γ ist eine C–Typisierung von e modulo Γ, die allgemeiner als jede andere C–Typisierung von e modulo Γ ist. Ist $\emptyset \triangleright e : \sigma$ eine Haupttypisierung von e (modulo $\emptyset$), so heißt σ ein *(C–) Haupttyp von e*.

Beispiel 4.1.4 (i) $x : \alpha \to \gamma \triangleright \lambda y.xy : \alpha \to \gamma$ ist eine Haupttypisierung von $\lambda y.xy$ modulo $\Gamma = x : \beta$, wobei α, β, γ Typvariablen seien.

(ii) $\emptyset \triangleright \lambda x.x : (\alpha \to \alpha) \to (\alpha \to \alpha)$ ist eine C-Typisierung von $\lambda x.x$ modulo $\emptyset$, aber es ist keine Haupttypisierung. Ein Haupttyp von $\lambda x.x$ ist $\alpha \to \alpha$.

4.2 Typsynthese für den C–Kalkül

Für jeden (ungetypten) λ–Term kann effektiv festgestellt werden, ob er eine C–beweisbare Typisierung hat. Falls er C–typisierbar ist, so hat er auch eine Hauptypisierung. Diese kann aus dem λ–Term automatisch „synthetisiert“ oder „inferiert“ werden. „Typinferenz“- oder „Synthese“-algorithmen, die dies leisten, werden bei der Syntaxanalyse beziehungsweise der Übersetzung von Programmen verwendet, um dem Programmierer das Aufschreiben der Typinformation zu ersparen.

Definition 4.2.1 Sei Γ ein Typkontext und e ein λ–Term, so daß Γ für jede frei in e vorkommende Variable x eine Annahme $x : \sigma$ Γ enthält. Der *Typsynthesealgorithmus W für C* ist wie folgt durch Induktion über e definiert:

$W(\Gamma, x)$	$=$	(Id, σ),	falls $x : \sigma$ die am weitesten rechts in Γ vorkommende Annahme über x ist
$W(\Gamma, e_1 \cdot e_2)$	$=$	$(US_2S_1, U\alpha)$,	falls $W(\Gamma, e_1) = (S_1, \sigma_1)$, $W(S_1\Gamma, e_2) = (S_2, \sigma_2)$, und $U = mgu(S_2\sigma_1, \sigma_2 \to \alpha)$ der allgemeinste Unifikator von $S_2\sigma_1$ und $\sigma_2 \to \alpha$ ist, wobei α eine unverbrauchte Typvariable ist,
$W(\Gamma, \lambda x.e)$	$=$	$(S, S\alpha \to \tau)$,	falls $W(\Gamma, x : \alpha, e) = (S, \tau)$, wobei α eine unverbrauch-, te Typvariable ist
$W(\Gamma, e)$	$=$	*fail*,	in allen anderen Fällen.

Satz 4.2.2 (Hindley[1]) (i) Jeder C–typisierbare λ–Term hat einen Haupttyp.

(ii) Die Menge der C-typisierbaren λ–Terme ist entscheidbar.

Beweis: Wir zeigen etwas allgemeiner durch Induktion über e für alle Γ:

(i) $W(\Gamma, e) = (S, \sigma) \quad \Rightarrow \quad \Gamma \rhd e : \sigma$ ist Haupttypisierung von e modulo Γ

(ii) $W(\Gamma, e) = \mathit{fail} \quad \Rightarrow \quad$ Es gibt keine C–Typisierung von e modulo Γ

(iii) $W(\Gamma, e)$ terminiert.

Sei also Γ ein C–Kontext mit $\mathit{frei}(e) \subseteq \mathit{dom}(\Gamma)$.

Fall $e \equiv x$: Γ kann zerlegt werden in $\Gamma \equiv \Gamma', x : \sigma, \Delta$, wobei Δ keine Typannahme über x enthält. Dann ist $W(\Gamma, x) = (Id, \sigma)$, und offenbar $\Gamma \rhd x : \sigma$ eine Haupttypisierung von x modulo Γ, also gelten i)–iii).

Fall $e \equiv e_1 e_2$: Nach Induktion terminiert $W(\Gamma, e_1)$. Falls $W(\Gamma, e_1) = \mathit{fail}$, so gibt es keine C–Typisierung von e_1 modulo Γ, also auch keine von $e_1 e_2$ modulo Γ. Nach der Definition ist auch $W(\Gamma, e_1 e_2) = \mathit{fail}$. Sei nun $W(\Gamma, e_1) = (S_1, \sigma_1)$, also $S_1\Gamma \rhd e_1 : \sigma_1$ eine Haupttypisierung modulo Γ. Nach Induktion terminiert $W(S_1\Gamma, e_2)$. Falls $W(S_1\Gamma, e_2) = \mathit{fail}$, so ist auch $W(\Gamma, e_1 e_2) = \mathit{fail}$, und es gibt keine C–Typisierung von e_2 modulo $S_1\Gamma$. Wäre $R\Gamma \rhd e_1 e_2 : \sigma$ eine C–Typisierung von $e_1 e_2$ modulo Γ, so wäre für geeignetes τ auch $R\Gamma \rhd e_1 : \tau \to \sigma$ eine C-Typisierung von e_1 modulo Γ, also (auf den Typvariablen in Γ)

$$R = R_1 \circ S_1, \qquad \tau = R_1(\sigma_1),$$

und $R\Gamma \rhd e_2 : \tau$ eine C–Typisierung von e_2 modulo Γ, also wegen $R\Gamma = R_1 S_1 \Gamma$ auch modulo $S_1\Gamma$, was unmöglich war. Sei daher $W(S_1\Gamma, e_2) = (S_2, \sigma_2)$, also

$$S_2 S_1 \Gamma \rhd e_2 : \sigma_2$$

eine Haupttypisierung modulo $S_1\Gamma$.

Dann erfolgt jede Typisierung von $e_1 e_2$ modulo Γ über Instanzen von

$$S_2 S_1 \Gamma \rhd e_1 : S_2(\sigma_1), \qquad S_2 S_1 \Gamma \rhd e_2 : \sigma_2$$

gemäß $(\to E)$. Sei nämlich

$$\frac{\begin{array}{cc}\vdots & \vdots \\ R\Gamma \rhd e_1 : \rho_1 \equiv (\rho_2 \to \rho) & R\Gamma \rhd e_2 : \rho_2\end{array}}{R\Gamma \rhd e_1 e_1 : \rho}$$

die Herleitung einer Typisierung von $e_1 e_2$ modulo Γ. Dann ist

$$R\Gamma \rhd e_1 : \rho_2 \to \rho \quad \equiv \quad R_1 S_1 \Gamma \rhd e_1 : R_1 \sigma_1$$

für eine geeignete Spezialisierung R_1 der Haupttypisierung $S_1\Gamma \rhd e_1 : \sigma_1$ modulo Γ. Damit ist aber auch

$$R\Gamma \rhd e_2 : \rho_2 \quad \equiv \quad R_1 S_1 \Gamma \rhd e_2 : \rho_2$$

eine Typisierung modulo $S_1\Gamma$. Da $S_2S_1\Gamma \rhd e_2 : \sigma_2$ eine Haupttypisierung modulo $S_1\Gamma$ war, gibt es eine weitere Substitution R_2 mit

$$\begin{array}{lll} (i) & R\Gamma \rhd e_2 : \rho_2 & \equiv R_1S_1\Gamma \rhd e_2 : \rho_2 \\ & & \equiv R_2S_2S_1\Gamma \rhd e_2 : \sigma_2 \\ (ii) & R\Gamma \rhd e_1 : \rho_2 \to \rho & \equiv R_2S_2S_1\Gamma \rhd e_1 : R_1\sigma_1. \end{array}$$

Also bleibt noch zu zeigen, daß $R_1\sigma_1 \equiv \rho_2 \to \rho \equiv R_2S_2\sigma_1$ ist. Dazu genügt, für jedes in σ_1 vorkommende α

$$R_1\alpha \equiv R_2S_2\alpha \tag{4.1}$$

zu zeigen. Falls α frei in $S_1\Gamma$ vorkommt, folgt (4.1) aus $R_1S_1\Gamma \equiv R_2S_2S_1\Gamma$ nach Eigenschaft (i). Sei α nicht frei in $S_1\Gamma$. Dann können wir $S_2(\alpha)$ frei wählen, da S_2 o.B.d.A. nur auf den freien Variablen von $S_1\Gamma$ definiert ist. Wegen $R_1\sigma_1 \equiv \rho_2 \to \rho$ ist $R_1\alpha$ ein Teilterm von $\rho_2 \to \rho$. Wähle eine frische Variable $\tilde{\alpha}$ und ändere S_2 und R_2 so, daß

$$S_2\alpha := \tilde{\alpha}, \qquad R_2\tilde{\alpha} := R_1\alpha.$$

Dann gilt wieder $R_1\alpha \equiv R_2S_2\alpha$, und die übrigen Eigenschaften von R_2 und S_2 bleiben erhalten.

Falls $S_2\sigma_1$ und $(\sigma_2 \to \alpha)$, mit frischem α, nicht unifizierbar sind, gibt es also keine C–Typisierung von e_1e_2 modulo Γ und $W(\Gamma, e_1e_2) = fail$. Anderenfalls sei $U = mgu(S_2\sigma_1, \sigma_2 \to \alpha)$. Dann erhalten wir

$$(\to E)\ \frac{US_1S_1\Gamma \rhd e_1 : US_2\sigma_1, \qquad US_2S_1\Gamma \rhd e_2 : U\sigma_2}{US_2S_1\Gamma \rhd e_1e_2 : U\alpha}$$

über $US_2\sigma_1 = U(\sigma_2 \to \alpha) = (U\sigma_2 \to U\alpha)$. Da U der allgemeinste Unifikator ist, ist diese C–Typisierung von e_1e_2 modulo Γ eine Haupttypisierung.

Fall $e = \lambda x.t$: Nach Induktion terminiert $W(\Gamma, x : \alpha, t)$. Falls $W(\Gamma, x : \alpha, t) = fail$, so ist auch $W(\Gamma, \lambda xt) = fail$. Wäre $R\Gamma \rhd \lambda x.t : \sigma \to \tau$ eine C–Typisierung modulo Γ, so wäre auch

$$R\Gamma, x : \sigma \rhd t : \tau$$

eine C–Typisierung modulo $\Gamma, x : \alpha$, was unmöglich ist wegen $W(\Gamma, x : \alpha, t) = fail$. Sei $W(\Gamma, x : \alpha, t) = (T, \tau)$, also

$$T\Gamma, x : T\alpha \rhd t : \tau$$

eine Haupttypisierung modulo $\Gamma, x : \alpha$. Dann ist $T\Gamma \rhd \lambda xt : T\alpha \to \tau$ eine C–Typisierung modulo Γ, und $W(\Gamma, \lambda xt) = (T, T\alpha \to \tau)$. Offenbar ist $T\Gamma \rhd \lambda xt : T\alpha \to \tau$ eine Hauptypisierung modulo Γ. □

Ist e ein abgeschlossener λ–Term, d.h. $frei(e) = \emptyset$, so ist mit $W(\emptyset, e) = (S, \sigma)$ der Typ σ ein Hauptyp von e. Beachte, daß Haupttypen nur bis auf Bijektionen von Typvariablen eindeutig sind.

Beispiel 4.2.3 Bezüglich $\Gamma = \{0 : int, f : \alpha\}$ hat $f \cdot 0$ den Haupttyp β, für frische β. Die Berechnung durch W ergibt:

$$\begin{array}{rcl} W(\Gamma, f \cdot 0) & := & (US_2S_1, U\beta), \text{ wobei } \beta \text{ neu und} \\ & & (S_1, \sigma_1) := W(\Gamma, f) = (Id, \alpha), \\ & & (S_2, \sigma_2) := W(S_1\Gamma, 0) = (Id, int), \\ & & U = mgu(S_2\sigma_1, \sigma_2 \to \beta) = mgu(\alpha, int \to \beta) = [int \to \beta/\alpha]. \end{array}$$

Also ist $W(\Gamma, f \cdot 0) = ([int \to \beta/\alpha], \beta)$, und $0 : int, f : int \to \beta \rhd f0 : \beta$ eine Haupttypisierung modulo Γ.

Beispiel 4.2.4 Haupttypisierungen bleiben nicht unter β–Reduktion erhalten:
Angenommen, wir haben geordnete Paare $\langle t_1, t_2\rangle$ und Paartypen $\tau_1 \times \tau_2$, mit (geeigneten Reduktionsregeln und) den Typisierungsregeln

$$(\times\ I)\ \ \frac{\Gamma \rhd s : \sigma, \qquad \Gamma \rhd t : \tau}{\Gamma \rhd \langle s, t\rangle : \sigma \times \tau}, \qquad (\times\ E_l)\ \ \frac{\Gamma \rhd \langle s, t\rangle : \sigma \times \tau}{\Gamma \rhd s : \sigma}, \qquad (\times\ E_r)\ \ \frac{\Gamma \rhd \langle s, t\rangle : \sigma \times \tau}{\Gamma \rhd t : \tau}.$$

(Man könnte $\langle t_1, t_2\rangle$ auch durch $\lambda z.zt_1t_2$ kodieren.) Für $(\lambda xt \cdot s) \equiv (\lambda x\langle x, x\rangle \cdot \lambda y.y)$ erhalten wir folgende Haupttypisierung

$$\dfrac{\dfrac{\dfrac{x : \gamma \rhd x : \gamma, \qquad x : \gamma \rhd x : \gamma}{x : \gamma \rhd \langle x, x\rangle : \gamma \times \gamma}\,(\times\ I)}{\rhd \lambda x\langle x, x\rangle : \gamma \to \gamma \times \gamma} \qquad \dfrac{y : \alpha \rhd y : \alpha}{\rhd \lambda y.y : \alpha \to \alpha}}{\rhd (\lambda x\langle x, x\rangle \cdot \lambda y.y) : (\alpha \to \alpha) \times (\alpha \to \alpha)}\,(\text{für } \gamma \equiv (\alpha \to \alpha))$$

Für den reduzierten Term $[s/x]t = \langle s, s\rangle$ haben wir aber eine allgemeinere Haupttypisierung:

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \rhd \lambda y.y : \alpha \to \alpha & \rhd \lambda y.y : \beta \to \beta\end{array}}{\rhd \langle \lambda y.y, \lambda y.y\rangle : (\alpha \to \alpha) \times (\beta \to \beta)}$$

Beispiel 4.2.5 Haupttypisierungen bleiben auch unter η–Reduktion i.a. nicht erhalten:
Die Haupttypisierung von $\lambda y.xy$ modulo $\Gamma = \{x : \alpha\}$ ist $[\beta \to \gamma/\alpha]\Gamma \rhd \lambda y.xy : \beta \to \gamma$, während $\Gamma \rhd x : \alpha$ der Haupttyp des η–reduzierten Terms ist. Für geschlossenes t bleibt aber der Haupttyp unter der Reduktion $\lambda y.ty \to t$ erhalten.

Aufgabe 4.2.1 Man zeige an einem Beispiel, daß Haupttypisierungen im D–Kalkül im allgemeinen nicht existieren.

In einem abgeschwächten Sinn hat auch die D-Typisierung Haupttypen, wie S. Ronchi della Rocca[3] zeigt.

Literatur

[1] Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[2] Roger Hindley. The completeness theorem for typing λ–terms. *Theoretical Computer Science*, 22, 1983.

[3] Simona Ronchi della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

Kapitel 5

Polymorphismus in der Programmiersprache *ML*

Die funktionale Programmiersprache Standard *ML*[4] erweitert das Typisierungskonzept des *C*–Kalküls in zweierlei Hinsicht, ohne die Entscheidbarkeit der Typisierbarkeit oder die Existenz von Haupttypisierungen einzubüßen:

(i) Typschemata, wie sie im *C*–Kalkül in herleitbaren Typen auftreten, werden in *ML* auch als Typannahmen erlaubt. Dazu wird der Typbegriff erweitert.

(ii) Für λ–Terme der Form $(\lambda xt \cdot s)$ wird eine Typisierungsregel eingeführt, die das Typschema von s zur Typisierung von $(\lambda xt \cdot s)$ ausnutzt; der Ausdruck $(\lambda xt \cdot s)$ kann nun typisierbar sein, auch wenn λxt es nicht ist.

Ein Ausdruck e heißt (typ)-*polymorph*, wenn er in verschiedenen Kontexten mit unterschiedlichem Typ verwendet werden kann. Ist etwa σ ein herleitbarer Typ von e mit $\Gamma = \emptyset$, so ist auch $\emptyset \triangleright e : S\sigma$ eine herleitbare Typisierung, und e kann in verschiedenen Kontexten mit verschiedenem Typ $S\sigma$ verwendet werden. Allgemeiner: ist

$$\Gamma \triangleright e : \sigma$$

eine *C*–Typisierung, und sind $\alpha_1, \ldots, \alpha_n$ Typvariable, die in Γ nicht vorkommen, so ist quasi auch

$$\Gamma \triangleright e : \forall\alpha_1 \ldots \forall\alpha_n.\sigma$$

herleitbar, und diese $\forall$–Quantoren können bei verschiedenen Verwendungen von e unterschiedlich belegt werden, gemäß der $\forall$-Beseitigungsregel

$$\frac{\Gamma \triangleright e : \forall\alpha.\sigma}{\Gamma \triangleright e : [\tau/\alpha]\sigma}.$$

Die *Motivation für (i)* ist nun, daß in den meisten Programmiersprachen viele Konstanten vorkommen, die ebenfalls polymorph verwendbar sind, auch wenn man sie nicht als λ–Term mit herleitbarem polymorphem Typ $\forall\alpha.\sigma$ auffassen will, zum Beispiel:

π_i :	$\forall\alpha_1, \alpha_2(\alpha_1 \times \alpha_2 \to \alpha_i)$	(Projektion auf die i-te Komponente)
o :	$\forall\alpha, \beta, \gamma(\alpha \to \beta) \times (\beta \to \gamma) \to (\alpha \to \gamma)$	(Komposition von Funktionen)
map :	$\forall\alpha, \beta.(\alpha \to \beta) \times \alpha$ `-list` $\to$ β `-list`	$(map\ f\ [a_1, \ldots, a_n] = [fa_1, \ldots, fa_n])$

Solche Typaussagen über Konstante einer Sprache wollen wir also als Annahmen in Γ zulassen.

Die *Motivation für (ii)* ist, daß man Funktionale programmieren möchte, die polymorphe Funktionen als Argumente akzeptieren, etwa

$$F := \lambda f.(f[0,1,2,3], f[\text{"}ab\text{"}, \text{"}cde\text{"}]),$$

so daß für polymorphe Funktionen wie

$$\texttt{reverse} : \forall\alpha(\alpha\ \texttt{-list} \to \alpha\ \texttt{-list}) \qquad \text{oder} \qquad \texttt{first} : \forall\alpha(\alpha\ \texttt{-list} \to \alpha)$$

die Ausdrücke $(F \cdot \texttt{reverse})$ oder $(F \cdot \texttt{first})$ sinnvolle und syntaktisch korrekte Programme sind, während für monomorphe Funktionen wie

$$sum : \texttt{int list} \to \texttt{int} \qquad \text{mit } sum\ [n_1, \ldots, n_k] = \Sigma_{i=1}^{k} n_i$$

der Ausdruck $(F \cdot sum)$ syntaktisch unkorrekt wäre. Die Idee hierbei ist, daß man zur Typisierung von $(F \cdot f)$ Typinformation aus dem *Argument* f (d.h. dem Kontext von F) gewinnt, die zur Syntaxanalyse von F verwendet werden kann, während diese Information aus F alleine nicht rekonstruierbar ist. Der Typ von f in $\lambda f.(f[0,1,2], f[\text{„}ab\text{“}, \text{„}cde\text{“}])$ ist aus den beiden Verwendungen –zumindest mit den bisher verwendeten Mitteln– nicht herleitbar: nach diesen ist

$$\begin{array}{ll} f : \texttt{int list} \to \sigma & \text{in } f[0,1,2], \text{ und} \\ f : \texttt{string list} \to \tau & \text{in } f[\text{„}ab\text{“}, \text{„}cde\text{“}], \end{array}$$

und wollte man diese Typen verallgemeinern zu einem Typ der Form

$$f : \forall\alpha.(\alpha\ \texttt{-list} \to \rho(\alpha)), \qquad \text{mit } \sigma = [\texttt{int}/\alpha]\rho, \text{ und } \tau = [\texttt{string}/\alpha]\rho,$$

so müsste man mit Unbekannten $\rho(\alpha)$ für Typmuster arbeiten, in denen gewisse Typvariable frei vorkommen. Es ist meines Wissens nicht bekannt, ob es Typsynthesalgorithmen für solche Fälle gibt.

5.1 *ML*–Programme und *ML*–Reduktionstheorie

Die funktionale Programmiersprache *ML* besteht im Kern aus einer Erweiterung des λ–Kalküls um neue syntaktische Konstrukte sowie Konstanten.

Definition 5.1.1 (*ML*–Terme)

$$\begin{array}{lcccccccc} c & := & true & | & false & | & \pi_1 & | & \pi_2 \\ e & := & x & | & c & | & (e_1 \cdot e_2) & | & \lambda x e \\ & & \langle e_1, e_2 \rangle & | & (if\ e_0\ then\ e_1\ else\ e_2) & | & (let\ x = e_1\ in\ e_2) & | & (rec\ x.e) \end{array}$$

Die Gleichungs– und Reduktionstheorien für *ML*-Terme ist grob gesagt eine Erweiterung der Theorie $\lambda\beta$ um die folgenden Reduktionsaxiome:

$$\begin{array}{rcl@{\qquad}rcl} (\mathit{if\ true\ then}\ e_1\ \mathit{else}\ e_2) & \to & e_1 & (\mathit{if\ false\ then}\ e_1\ \mathit{else}\ e_2) & \to & e_2 \\ (\mathit{let}\ x = s\ \mathit{in}\ t) & \to & [s/x]t & (\mathit{rec}\ x.e) & \to & [(\mathit{rec}\ x.e)/x]e \\ (\pi_1 \cdot \langle e_1, e_2\rangle) & \to & e_1 & (\pi_2 \cdot \langle e_1, e_2\rangle) & \to & e_2 \end{array}$$

Dazu kommen noch entsprechende Verträglichkeitsregeln, so daß man Ersetzungen innerhalb eines Ausdrucks vornehmen kann. Allerdings wird die Anwendung der Reduktionen in *SML* eingeschränkt, sodaß die erweiterte β-Reduktion die wirkliche (call-by-value) Reduktion von *SML* nicht korrekt beschreibt. Darauf gehen wir im folgenden Kapitel näher ein.

Bemerkung 5.1.2 Die Terme $(\lambda xt \cdot s)$, $(\mathit{let}\ x = s\ \mathit{in}\ t)$ und $[s/x]t$ sind *ML*-gleich, falls $x \in \mathit{frei}(t)$. Der Grund für die Einführung von *let* besteht in der Ausnutzung folgender Unterschiede:

- $(\mathit{let}\ x = s\ \mathit{in}\ t)$ ist einfacher als $(\lambda xt \cdot s)$, da als Teilterme nur x, s, t auftreten, aber nicht λxt.
- $(\mathit{let}\ x = s\ \mathit{in}\ t)$ ist einfacher als $[s/x]t$, da i.a. mehrere Vorkommen von s in $[s/x]t$ zu einem Vorkommen in $(\mathit{let}\ x = s\ \mathit{in}\ t)$ zusammengefaßt werden.

Die durch `let` eingeführte Abkürzung ermöglicht also der Erweiterung der typisierbaren λ–Terme und eine effizientere Syntaxanalyse.

Beispiel 5.1.3 Zur Listenverarbeitung seien Grundfunktionen *nil*, *cons*, *head*, *tail* und *null*? vorhanden. Dann kann man mit $(\mathit{letrec}\ x = e\ \mathit{in}\ t) := (\mathit{let}\ x = (\mathit{rec}\ x.e)\ \mathit{in}\ t)$ etwa definieren:

$$\begin{array}{lll} \mathit{letrec} & \mathit{map} = \lambda f \lambda l.\ (\mathit{if} & (\mathit{null?}\ l) \\ & \quad \mathit{then} & \mathit{nil} \\ & \quad \mathit{else} & (\mathit{cons}\ (f\ (\mathit{head}\ l))\ (\mathit{map}\ f\ (\mathit{tail}\ l)))) \\ \mathit{in} & (\mathit{map}\ g\ (\mathit{map}\ h\ [0,1,2])). & \end{array}$$

Man sieht, daß die beiden Vorkommen von *map* im allgemeinen verschiedene Typen erfordern.

5.2 *ML*–Typisierungen

In *ML* unterscheiden wir zwischen zwei Universen $U_1 \subset U_2$ von Typen, den *monomorphen Typen* und den *polymorphen Typen*, oder kurz den Monotypen und Polytypen. Die Polytypen werden auch oft *Typschemata* genannt. Wir beschränken uns hier bei den Basistypen auf die Wahrheitswerte, obwohl natürlich auch Zahlen, Listen u.a. zu den Basistypen von *SML* gehören.

Definition 5.2.1 (Monotypen und Polytypen)

$$\begin{array}{lrcl} \text{(Basistypen)} & \iota & := & \texttt{bool} \\ \text{(Monotypen)} & \tau & := & \alpha \mid \iota \mid (\tau_1 \to \tau_2) \mid (\tau_1 * \tau_2) \\ \text{(Polytypen)} & \overline{\sigma} & := & \tau \mid \forall\alpha.\overline{\sigma} \end{array}$$

Den Begriff der Substitution S : Typvariablen $\to$ Monotypen setzen wir auf Polytypen fort durch

$$S\forall\alpha.\overline{\sigma} = \forall\tilde{\alpha}.S[\tilde{\alpha}/\alpha]\overline{\sigma}, \qquad \text{mit einer frischen Variable } \tilde{\alpha}.$$

Typquantoren treten nur im Präfix auf, nicht im Bereich eines Typkonstruktors $\to, *$.

Definition 5.2.2 Eine Monotyp τ ist eine *generische Instanz* eines Polytyps $\overline{\sigma} = \forall\alpha_1 \ldots \alpha_n.\sigma$, geschrieben $\overline{\sigma} \succ \tau$, falls $S\sigma = \tau$ für eine nur auf $\{\alpha_1, \ldots, \alpha_n\}$ definierte Substitution S. Für Polytypen sei $\overline{\sigma}_1 \succ \overline{\sigma}_2$, falls $\overline{\sigma}_1 \succ \tau$ für alle Monotypen τ mit $\overline{\sigma}_2 \succ \tau$.

Man sieht leicht, daß $\overline{\sigma}_1 \succ \overline{\sigma}_2$ genau dann der Fall ist, wenn falls für einen Monotyp τ

$$\overline{\sigma}_1 \succ \tau \text{ und } \overline{\sigma}_2 \equiv \forall\alpha_1 \ldots \alpha_n.\tau \text{ mit } \alpha_1, \ldots, \alpha_n \notin \mathit{frei}(\overline{\sigma}_1).$$

Offenbar ist für $\overline{\sigma}_1 \succ \overline{\sigma}_2$ stets $\mathit{frei}(\sigma_1) \subseteq \mathit{frei}(\sigma_2)$. Beispielsweise hat man:

$$\begin{array}{llll} \forall\alpha.\alpha & \succ & \forall\beta(\beta \to \beta), & \forall\beta(\beta \to \beta) \not\succ \forall\alpha.\alpha, \\ \forall\beta(\beta \to \beta) & \succ & \texttt{boole} \to \texttt{boole}, & \forall\beta(\beta \to \beta) \succ \forall\alpha((\alpha \to \beta) \to (\alpha \to \beta)). \end{array}$$

Definition 5.2.3 (*ML*–Typisierungskalkül) Für die Konstanten nehmen wir folgende Menge Γ_0 von Grundannahmen an:

$$\begin{array}{llllll} \mathit{true} & : & \texttt{boole}, & \pi_1 & : & \forall\alpha\forall\beta(\alpha * \beta \to \alpha), \\ \mathit{false} & : & \texttt{boole}, & \pi_2 & : & \forall\alpha\forall\beta(\alpha * \beta \to \beta). \end{array}$$

Der Kalkül hat folgende Axiome und Regeln, wobei Γ Typannahmen mit Polytypen enthalten kann. Für Monotypen σ sei $\overline{\sigma}^{\Gamma} := \forall\alpha_1 \ldots \alpha_n.\sigma$ mit $\mathit{frei}(\sigma) - \mathit{frei}(\Gamma) = \{\alpha_1, \ldots, \alpha_n\}$.

$$(\mathit{Var}) \quad \Gamma \rhd x : \tau, \quad \text{falls } \Gamma(x) \succ \tau \qquad\qquad (\mathit{const}) \quad \Gamma \rhd c : \tau, \quad \text{falls } \Gamma_0(c) \succ \tau$$

$$(\to I) \quad \frac{\Gamma_x, x : \sigma \rhd t : \tau}{\Gamma_x \rhd \lambda x t : \sigma \to \tau} \qquad\qquad (\to E) \quad \frac{\Gamma \rhd f : \sigma \to \tau, \quad \Gamma \rhd s : \sigma}{\Gamma \rhd (f \cdot s) : \tau}$$

$$(\mathit{if}) \quad \frac{\Gamma \rhd e_0 : \mathit{boole}, \quad \Gamma \rhd e_1 : \tau, \quad \Gamma \rhd e_2 : \tau}{\Gamma \rhd (\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2) : \tau} \qquad (* I) \quad \frac{\Gamma \rhd s : \sigma \qquad \Gamma \rhd t : \tau}{\Gamma \rhd \langle s, t\rangle : (\sigma * \tau)}$$

$$(\mathit{let}) \quad \frac{\Gamma_x \rhd e : \sigma, \quad \Gamma_x, x : \overline{\sigma}^{\Gamma} \rhd t : \tau}{\Gamma_x \rhd (\mathit{let}\ x = e\ \mathit{in}\ t) : \tau} \qquad (\mathit{rec}) \quad \frac{\Gamma_x, x : \tau \rhd e : \tau}{\Gamma_x \rhd (\mathit{rec}\ x.e) : \tau}$$

Für geordnete Paare sind die Beseitigungsregeln

$$(* E)_1 \quad \frac{\Gamma \rhd t : (\tau_1 * \tau_2)}{\Gamma \rhd (\pi_1 \cdot t) : \tau_1}, \qquad (* E)_2 \quad \frac{\Gamma \rhd t : (\tau_1 * \tau_2)}{\Gamma \rhd (\pi_2 \cdot t) : \tau_2}.$$

aus $(\to E)$ und den Typannahmen über π_1, π_2 herleitbar.

Beachte, daß Poltypen nur in (*Var*), (*const*) und (*let*) auftreten. Die durch *rec* und λ gebundenen Variablen sind monomorph, d.h. haben überall im Wirkungsbereich denselben Typ (*monomorph = von einer einzigen Form*). Durch *let* gebundene Variablen sind polymorph, d.h. treten im Wirkungsbereich i.a. mit unterschiedlichen Typ auf (*polymorph = von vielerlei Form*). Da die verschiedenen Typen sich durch Einsetzen in die (gebundenen) Typparameter des Schemas ergeben, spricht man auch von *parametrischem Polymorphismus* – im Unterschied zu Überlagerungs- oder ad-hoc-Polymorphismus und Subtyppolymorphismus.

Bemerkung 5.2.4 (i) Durch polymorphe Annahmen kann man (in endlicher Weise) ausdrük-ken, daß Grundfunktionen einer Programmiersprache uniform in den Typen ihrer Argumente operieren. Für Listenfunktionen hat man etwa die Annahmen

$$\begin{array}{lcl} \texttt{Nil} & : & \forall\alpha(\alpha\texttt{-list}), \\ \texttt{Cons} & : & \forall\alpha(\alpha \times (\alpha\texttt{-list}) \to \alpha\texttt{-list}), \\ \texttt{Car} & : & \forall\alpha(\alpha\texttt{-list} \to \alpha). \end{array}$$

Beim Aufbau oder der Zerlegung der Listen spielt der Typ der Elemente keine Rolle.

(ii) Die Verwendung von Polytypen in den Annahmen erlaubt, ähnlich wie im *D*–Kalkül, Terme zu typisieren, die nicht *C*–typisierbar sind:

$$\frac{x : \forall\alpha.\alpha \triangleright x : \beta \to \gamma \quad , \quad x : \forall\alpha.\alpha \triangleright x : \beta}{x : \forall\alpha.\alpha \triangleright x \cdot x : \gamma}$$

Im Unterschied zum *D*–Kalkül kann man im *ML*–Kalkül polymorphe Annahmevariable aber nicht durch Abstraktion beseitigen: weder $\lambda x.xx$ noch $(rec\, x.xx)$ sind *ML*–typisierbar.

(iii) Statt der *rec*-Ausdrücke kann man eine polymorphe Konstante Y, den *Fixpunktoperator* (vgl. 2.3.2), einführen, mit dem Typ

$$Y : \forall\alpha\,((\alpha \to \alpha) \to \alpha)$$

und der Reduktionsregel

$$(Y) \qquad Yf \to f(Yf).$$

Durch $(rec\, x.e) := (Y \cdot \lambda x.e)$ lassen sich die `rec`-Ausdrücke darauf zurückführen. Die Typisierungsregel für *rec* ist jetzt herleitbar: Für $\tilde{\Gamma}_x = \Gamma_x, Y : \forall\alpha.(\alpha \to \alpha) \to \alpha$ haben wir nämlich:

$$\frac{\dfrac{\vdots}{\tilde{\Gamma}_x, x : \sigma \triangleright e : \sigma}}{\tilde{\Gamma}_x \triangleright \lambda x.e : \sigma \to \sigma} \qquad \tilde{\Gamma}_x \triangleright Y : (\sigma \to \sigma) \to \sigma}{\tilde{\Gamma}_x \triangleright (Y \cdot \lambda xe) : \sigma.}$$

In *SML* ist der Gebrauch der Rekursion allerdings so eingeschränkt, daß e die Form λyt haben muß: man kann nur rekursive Funktionen, keine rekursiven Daten definieren.

Beispiel 5.2.5 Wie in 5.2.4 (i) haben wir eine *ML*–Ableitung

$$\frac{\begin{array}{cc}\vdots & \vdots\\ id : \forall\alpha(\alpha\to\alpha) \triangleright id : (\beta\to\beta)\to(\beta\to\beta), & id : \forall\alpha(\alpha\to\alpha) \triangleright id : (\beta\to\beta)\end{array}}{id : \forall\alpha(\alpha\to\alpha) \triangleright (id\cdot id) : \beta\to\beta}$$

Diese läßt sich ergänzen zu

$$\frac{\dfrac{x : \alpha \triangleright x : \alpha}{\triangleright \lambda x.x : \alpha\to\alpha} \qquad \begin{array}{c}\vdots\\ id : \forall\alpha.\alpha\to\alpha \triangleright (id\cdot id) : \beta\to\beta\end{array}}{\triangleright (let\ id = \lambda x.x\ in\ (id\cdot id)) : \beta\to\beta}$$

Beachte, daß id in $(id\cdot id)$ mit zwei verschiedenen Instanzen des für $\lambda x.x$ hergeleiteten Typschemas $\alpha\to\alpha$ verwendet wird. Deswegen ist auch $(\lambda id.(id\cdot id))\cdot(\lambda x.x)$ *nicht ML*–typisierbar.

Definition 5.2.6 Eine Substitution S *betrifft* die Variable α, falls $S\alpha \neq \alpha$ oder $\alpha \in frei(S\beta)$ für ein $\beta \not\equiv \alpha$.

Proposition 5.2.7 Für Umgebungen Δ, Γ, Substitutionen S und Monotypen τ gilt:

(i) Ist $frei(\Gamma) \subseteq frei(\Delta)$, so ist $\overline{\tau}^{\Gamma} \succ \overline{\tau}^{\Delta}$.

(ii) $S(\overline{\tau}^{\Gamma}) \succ \overline{S\tau}^{S\Gamma}$.

(iii) Falls S die Variablen in $frei(\tau) - frei(\Gamma)$ nicht betrifft, ist $S(\overline{\tau}^{\Gamma}) = \overline{S\tau}^{S\Gamma}$.

Beweis: (iii) Sei $frei(\tau) - frei(\Gamma) = \{\alpha_1, \ldots, \alpha_n\}$. Dann ist

$$S\overline{\tau}^{\Gamma} = S\forall\alpha_1 \ldots \alpha_n.\tau = \forall\alpha_1 \ldots \alpha_n.S\tau = \overline{S\tau}^{S\Gamma},$$

da auch $frei(S\tau) - frei(S\Gamma) = \{\alpha_1, \ldots, \alpha_n\}$. □

Ist zum Beispiel $\tau = \alpha\to\alpha$ und $S\alpha = (\beta\to\beta)$ eine Substitution, die die Bedingung in (iii) nicht erfüllt, so gilt für jedes Γ ohne freie Typvariable

$$S(\overline{\tau}^{\Gamma}) = \forall\alpha(\alpha\to\alpha) \succ \forall\beta((\beta\to\beta)\to(\beta\to\beta)) = \overline{S\tau}^{S\Gamma},$$

wie in (ii) behauptet, aber $S(\overline{\tau}^{\Gamma}) \neq \overline{S\tau}^{S\Gamma}$.

Lemma 5.2.8 (Substitutionslemma)

(i) Falls $\Gamma \vdash_{ML} e : \tau$, so ist auch $S\Gamma \vdash_{ML} e : S\tau$.

(ii) Falls $\Gamma, x : \overline{\sigma}^{\Gamma} \vdash_{ML} e : \tau$ und $\Gamma \vdash_{ML} v : \sigma$, so ist auch $\Gamma \vdash_{ML} [v/x]e : \tau$.

(iii) Falls $\Gamma, x : \overline{\tau} \vdash_{ML} e : \rho$ und $\overline{\sigma} \succ \overline{\tau}$, so ist auch $\Gamma, x : \overline{\sigma} \vdash_{ML} e : \rho$.

Beweis: (i) Durch Induktion über die Herleitung von $\Gamma \vdash_{ML} e : \tau$.
Der letzte Schritt ist eine Anwendung von (*Var*): Sei $\overline{\sigma} \succ \tau$ mit $x : \overline{\sigma}$ in Γ,

$$\overline{\sigma} = \forall \alpha_1 \ldots \alpha_n.\tau_0 \quad \text{und} \quad \tau = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau_0$$

Ist S eine Typsubstitution, die o.B.d.A die Variablen $\alpha_1, \ldots, \alpha_n$ nicht betrifft, so ist

$$S\overline{\sigma} = \forall \alpha_1 \ldots \alpha_n.S\tau_0 \quad \text{und} \quad S\tau = [S\tau_1/\alpha_1, \ldots, S\tau_n/\alpha_n]S\tau_0,$$

also $x : S\overline{\sigma}$ in $S\Gamma$ und $S\overline{\sigma} \succ S\tau$ und daher $S\Gamma \vdash_{ML} x : S\tau$. Entsprechend argumentiert man, wenn der letzte Schritt eine Anwendung von (*const*) ist.

Der letzte Schritt ist eine Anwendung von (*if*), $(* \ I)$ oder (*rec*): Die Behauptung folgt wie in Lemma 4.1.2 direkt aus der Induktionsannahme.

Der letze Schritt ist eine Anwendung von (*let*): Beachte, daß mit herleitbaren

$$\Gamma_x \rhd e : \sigma \quad \text{und} \quad \Gamma_x, x : \overline{\sigma}^{\Gamma} \rhd t : \tau$$

nach Induktion auch

$$S\Gamma_x \rhd e : S\sigma \quad \text{und} \quad S\Gamma_x, S\overline{\sigma}^{\Gamma} \rhd t : S\tau$$

herleitbar sind. Durch Umbenennung können wir annehmen, daß S die Variablen in $\mathit{frei}(\sigma) - \mathit{frei}(\Gamma)$ nicht betrifft. Dann ist aber $S\overline{\sigma}^{\Gamma} = \overline{S\sigma}^{S\Gamma}$, und mit einer Anwendung von (*let*) folgt

$$S\Gamma_x \vdash_{ML} (\mathit{let}\ x = e\ \mathit{in}\ t) : S\tau.$$

(ii) Durch Induktion über die Struktur von e in $\Gamma, x : \overline{\sigma}^{\Gamma} \vdash_{ML} e : \tau$. Sei $\overline{\sigma}^{\Gamma} \equiv \forall \vec{\alpha}.\sigma$.

Fall $e \equiv c$ oder $e \equiv y \not\equiv x$: Wegen $[v/x]e \equiv e$ folgt $\Gamma \vdash_{ML} [v/x]e : \tau$ aus der Annahme.

Fall $e \equiv x$: Dann ist $\tau = [\vec{\rho}/\vec{\alpha}]\sigma$ für geeignete Typen $\vec{\rho}$, so daß $[\vec{\rho}/\vec{\alpha}]\Gamma \vdash_{ML} v : [\vec{\rho}/\vec{\alpha}]\sigma$, also $\Gamma \vdash_{ML} v : \tau$. Daraus folgt $\Gamma, x : \overline{\sigma}^{\Gamma} \vdash_{ML} [v/x]x : \tau$, da wegen $x \notin dom(\Gamma)$ auch $x \notin \mathit{frei}(v)$.

Fall $e \equiv (e_1 \cdot e_2)$, $e \equiv \lambda y.e'$: Diese Fälle bleiben dem Leser als Übung überlassen.

Fall $e \equiv (\mathit{let}\ y = e_1\ \mathit{in}\ e_2)$: Die Herleitung endet in

$$(\mathit{let})\ \frac{\Gamma, x : \overline{\sigma}^{\Gamma} \rhd e_1 : \rho \qquad \Gamma, x : \overline{\sigma}^{\Gamma}, y : \overline{\rho}^{\Gamma, x:\overline{\sigma}^{\Gamma}} \rhd e_2 : \tau}{\Gamma, x : \overline{\sigma}^{\Gamma} \rhd (\mathit{let}\ y = e_1\ \mathit{in}\ e_2) : \tau}$$

Aus der Voraussetung $\Gamma \vdash_{ML} v : \sigma$ folgt durch Abschwächen (und oBdA $y \notin \mathit{frei}(v)$)

$$\Gamma, y : \overline{\rho}^{\Gamma, x:\overline{\sigma}^{\Gamma}} \vdash_{ML} v : \sigma.$$

Beachte, daß $\overline{\sigma}^{\Gamma} = \overline{\sigma}^{\Gamma, y:\overline{\rho}^{\Gamma, x:\overline{\sigma}^{\Gamma}}}$. Also sind nach Induktionsannahme

$$\Gamma \vdash_{ML} [v/x]e_1 : \rho \quad \text{und} \quad \Gamma, y : \overline{\rho}^{\Gamma, x:\overline{\sigma}^{\Gamma}} \vdash_{ML} [v/x]e_2 : \tau.$$

Nach 5.2.7 (i) ist $\overline{\rho}^{\Gamma} \succ \overline{\rho}^{\Gamma, x:\overline{\sigma}^{\Gamma}}$, und daher mit Behauptung (iii) auch $\Gamma, y : \overline{\rho}^{\Gamma} \vdash_{ML} [v/x]e_2 : \tau$. Mit der Typregel (*let*) folgt die Behauptung.

(iii) Durch Induktion über die Ableitung von $\Gamma, x : \overline{\tau} \triangleright e : \rho$.

Der letzte Schritt ist eine Anwendung von (*Var*): Dann ist entweder $e \not\equiv x$ und nichts zu zeigen, oder es ist $e \equiv x$. In diesem Fall ist nach Voraussetzung $\overline{\tau} \succ \rho$. Nach Definition von $\overline{\sigma} \succ \overline{\tau}$ ist aber auch $\overline{\sigma} \succ \rho$.

Der letzte Schritt ist eine Anwendung von (*let*): Die Herleitung ende in

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma, x : \overline{\tau} \triangleright e : \eta & \Gamma, x : \overline{\tau}, y : \overline{\eta}^{\Gamma, x:\overline{\tau}} \triangleright t : \rho\end{array}}{\Gamma, x : \overline{\tau} \triangleright (let\ x = e\ in\ r) : \rho}.$$

Nach Induktion ist $\Gamma, x : \overline{\sigma} \vdash_{ML} e : \eta$ und $\Gamma, x : \overline{\sigma}, y : \overline{\eta}^{\Gamma, x:\overline{\tau}} \vdash_{ML} t : \rho$. Da wegen $\overline{\sigma} \succ \overline{\tau}$ aber $\mathit{frei}(\overline{\sigma}) \subseteq \mathit{frei}(\overline{\tau})$ gilt, ist $\overline{\eta}^{\Gamma, x:\overline{\sigma}} \succ \overline{\eta}^{\Gamma, x:\overline{\tau}}$ nach Proposition 5.2.7 (i). Also folgt aus der Induktionsannahme, daß $\Gamma, x : \overline{\sigma}, y : \overline{\eta}^{\Gamma, x:\overline{\sigma}} \vdash_{ML} t : \rho$. Eine Anwendung von (*let*) ergibt

$$\Gamma, x : \overline{\sigma} \vdash_{ML} (let\ x = e\ in\ t) : \rho.$$

Wird im letzten Schritt eine der anderen Typregeln angewendet, so folgt die Behauptung aus der Induktionsannahme. □

Proposition 5.2.9 (i) $(\lambda x t \cdot s) \subseteq_{ML} (let\ x = s\ in\ t) \subseteq_{ML} [s/x]t$.

(ii) Falls $x \in \mathit{frei}(t)$, so ist $[s/x]t \subseteq_{ML} (let\ x = s\ in\ t)$.

Der Beweis bleibt dem Leser zur Übung überlassen. Die Voraussetzung in (ii) ist nötig, da s in $(let\ x = s\ in\ t)$ ein untypisierbarer Term sein kann, etwa $\lambda y(yy)$.

Bemerkung 5.2.10 (i) Alle *rec*–freien Terme sind stark normalisierbar, während rekursive Definitionen zu unendlichen Reduktionsfolgen führen, etwa

$$\begin{array}{rll}(rec\ f.\lambda x.fx) & \to_{rec} & \lambda x.(rec\ f.\lambda x.fx)x \\ & \to_{rec} & \lambda x.(\lambda x.rec\ f.\lambda x.fx)x)x \\ & \to \ldots & \end{array}$$

(ii) Jede Folge von *let*–Reduktionen ausgehend von einem *ML*–Term t ist endlich; dies folgt aus dem Satz über die Endlichkeit von „Developments“ (vgl. Barendregt, Kap. 11,2): betrachte *let*–Ausdrücke als markierte β–Redexe.

5.3 Typsynthese für *ML*

Die Entscheidbarkeit der Typisierbarkeit und des Typsynthesealgorithmus des C–Kalküls lassen sich auf *ML* fortsetzen. Die Verwendung dieses Algorithmus' im *ML*–Compiler hat sich als große praktische Hilfe beim Programmieren bewährt, da sich viele Programmierfehler als Typfehler bemerkbar machen (Folgefehler) und *vor* der Verwendung des Programms beseitigt werden können.

Definition 5.3.1 (Typsynthesealgorithmus für *ML*) Sei Γ ein Typkontext, in dem Polytypen auftreten können, und e ein *ML*–Term, für dessen freie Variable Typannahmen in Γ vorhanden sind. Die Definition von $W(\Gamma, e)$ aus 4.2.1 wird wie folgt geändert und erweitert:

$W(\Gamma, x) = (Id, \tau[\alpha'_1/\alpha_1, \ldots, \alpha'_n/\alpha_n])$,

falls $\Gamma(x) = \forall\alpha_1 \ldots \alpha_n.\tau$ und $\alpha_1, \ldots, \alpha_n$ die nächsten n frischen Typvariablen sind.

$W(\Gamma, c) = (Id, \tau[\alpha'_1/\alpha_1, \ldots, \alpha'_n/\alpha_n])$,

falls $\Gamma_0(c) = \forall\alpha_1 \ldots \alpha_n.\tau$ und $\alpha_1, \ldots, \alpha_n$ die nächsten n frischen Typvariablen sind.

$W(\Gamma, (\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2)) = (US_2S_1S_0\Gamma, U\sigma_2)$,

falls
$$\begin{aligned} W(\Gamma, e_0) &= (S, \sigma_0), \\ S_0 &= mgu(\sigma_0, boole)S, \\ W(S_0\Gamma, e_1) &= (S_1, \sigma_1), \\ W(S_1S_0\Gamma, e_2) &= (S_2, \sigma_2), \text{ und} \\ U &= mgu(S_2\sigma_1, \sigma_2) \end{aligned}$$

$W(\Gamma, \langle e_1, e_2\rangle) = (S_2S_1, (S_2\tau_1 * \tau_2))$,

falls $W(\Gamma, e_1) = (S_1, \tau_1)$, $W(S_1\Gamma, e_2) = (S_2, \tau_2)$

$W(\Gamma_x, (\mathit{let}\ x = s\ \mathit{in}\ t)) = (TS, \tau)$,

falls
$$\begin{aligned} W(\Gamma_x, s) &= (S, \sigma) \text{ und} \\ W(S\Gamma_x, x : \overline{\sigma}^{S\Gamma_x}, t) &= (T, \tau) \end{aligned}$$

$W(\Gamma_x, \mathit{rec}\ x.e) = (US, U\sigma)$,

falls
$$\begin{aligned} W(\Gamma_x, x : \alpha, e) &= (S, \sigma) \text{ und} \\ U &= mgu(S\alpha, \sigma) \end{aligned}$$

$W(\Gamma, e) = fail$

in allen anderen Fällen, soweit sie nicht schon in 4.2.1 definiert sind.

Satz 5.3.2 (Damas/Milner 1982) (i) Jeder *ML*–typisierbare *ML*–Term hat einen Haupttyp.

(ii) Die Menge der *ML*–typisierbaren *ML*–Terme ist entscheidbar.

Beweis: Es genügt zu zeigen, daß die Eigenschaften (i)–(iii) aus dem Beweis von 4.2.2 gelten. Dies geht für Terme der Form $(rec\ x.e)$ wie für $\lambda x.e$ in 4.2.2, und für $(if\ e_0\ then\ e_1\ else\ e_2)$ und $\langle e_1, e_2\rangle$ ähnlich wie für $(e_1 \cdot e_2)$ in 4.2.2.

Fall $W(\Gamma, x)$*:* Nach Voraussetzung enthält Γ eine Annahme für x, etwa $\Gamma(x) = \forall\alpha_1 \ldots \forall\alpha_n.\tau$. Daher ist $W(\Gamma, x) = (Id, \tau')$ mit $\tau' = [\alpha'_1/\alpha_1, \ldots, \alpha'_n/\alpha_n]\tau$ für frische Variablen α'_i. Damit sind (ii) und (iii) klar, und offenbar ist $\Gamma \rhd x : \tau'$ eine *ML*–Typisierung. Für (i) bleibt zu zeigen, daß es eine Haupttypisierung modulo Γ ist.

Sei $S\Gamma \rhd x : \sigma'$ eine *ML*–Typisierung modulo Γ. Dann gibt es $x : \forall\beta_1 \ldots \forall\beta_k.\sigma \equiv S(\forall\alpha_1 \ldots \forall\alpha_n.\tau)$ in $S\Gamma$ und $\sigma' = [\beta'_1/\beta_1, \ldots, \beta'_k/\beta_k]\sigma$ für frische β'_i. OBdA ist $S(\forall\alpha_1 \ldots \forall\alpha_n.\tau) = \forall\alpha_1 \ldots \forall\alpha_n.S\tau$ mit $S\alpha_i = \alpha_i$, also $\forall\beta_1 \ldots \forall\beta_k.\sigma = \forall\alpha_1 \ldots \forall\alpha_n.S\tau$. Damit ist $\sigma' = [\beta'_1/\beta_1, \ldots, \beta'_k/\beta_k]\sigma = [\beta'_1/\alpha_1, \ldots, \beta'_k/\alpha_k]\tau =_{o.E.} S\tau'$, also $S\Gamma \rhd x : \sigma' = S\Gamma \rhd x : S\tau'$ eine Instanz von $\Gamma \rhd x : \tau'$.

Fall $W(\Gamma, c)$*:* Das geht wie für Variable, nur mit den Grundannahmen Γ_0.

Fall $W(\Gamma_x, (let\ x = s\ in\ t))$: Nach Induktion ist iii) klar, und ii) ist ähnlich wie in 4.2.2 zu zeigen. Für i) seien $\Gamma = \Gamma_x$,

$$\begin{array}{lcl} W(\Gamma, (let\ x = s\ in\ t)) & = & (TS, \tau), \\ W(\Gamma, s) & = & (S, \sigma), \\ W(S\Gamma, x : \overline{\sigma}^{S\Gamma}, t) & = & (T, \tau). \end{array}$$

(i) $TS\Gamma \rhd (let\ x = s\ in\ t) : \tau$ ist eine *ML*–Typisierung: Nach Induktion ist $S\Gamma \vdash_{ML} s : \sigma$ und $TS\Gamma, x : T(\overline{\sigma}^{S\Gamma}) \vdash_{ML} t : \tau$, und mit 5.2.8 (i) auch $TS\Gamma \vdash_{ML} s : T\sigma$. Um (let) anzuwenden, muß noch $\overline{T\sigma}^{TS\Gamma} = T(\overline{\sigma}^{S\Gamma})$ gezeigt werden. Für $\sigma = \sigma(\alpha, \beta)$ mit $\alpha \in frei(S\Gamma), \beta \notin frei(S\Gamma)$ ist aber

$$\begin{array}{lcll} T(\overline{\sigma}^{S\Gamma}) & = & T(\forall\beta.\sigma(\alpha, \beta)) & \\ & =_{o.E.} & \forall\beta.\sigma(T\alpha, \beta) & \\ & = & \overline{T\sigma}^{TS\Gamma} & (\text{bei } \{\beta\} = frei(S\Gamma), T\beta = \beta) \end{array}$$

(ii) $TS\Gamma \rhd (let\ x = s\ in\ t) : \tau$ ist *Haupt*typisierung modulo Γ. Sei $R\Gamma \rhd (let\ x = s\ in\ t) : \rho$ eine vorgegebene *ML*-Typisierung modulo Γ. Nach der Regel (let) haben wir

$$R\Gamma \vdash_{ML} s : \sigma_s, \qquad \text{und} \qquad R\Gamma, x : \overline{\sigma_s}^{R\Gamma} \vdash_{ML} t : \varrho.$$

Nach Induktion ist $S\Gamma \rhd s : \sigma$ eine Haupttypisierung modulo Γ, also für geeignetes R_1

$$R\Gamma \rhd s : \sigma_s \quad \equiv \quad R_1S\Gamma \rhd e : R_1\sigma.$$

Wegen $R_1(\overline{\sigma}^{S\Gamma}) \succ \overline{R_1\sigma}^{R_1S\Gamma} \equiv \overline{\sigma_s}^{R\Gamma}$ folgt

$$R_1S\Gamma, x : R_1(\overline{\sigma}^{S\Gamma}) \vdash_{ML} t : \varrho.$$

Da $TS\Gamma, x : T(\overline{\sigma}^{S\Gamma}) \rhd t : \tau$ eine Haupttypisierung modulo $S\Gamma, x : \overline{\sigma}^{S\Gamma}$ ist, ist analog

$$R_1S\Gamma, x : R_1(\overline{\sigma}^{S\Gamma}) \rhd t : \varrho \qquad \equiv \qquad R_2TS\Gamma, x : R_2T(\overline{\sigma}^{S\Gamma}) \rhd t : R_2\tau$$

für eine geeignete Substitution R_2. Also ist $R\Gamma, x : \overline{\sigma_s}^{S\Gamma} \triangleright t : \varrho$ eine Instanz von $TS\Gamma, x : T(\overline{\sigma}^{S\Gamma}) \triangleright t : \tau$, und $R\Gamma \triangleright s : \sigma_s \equiv R_2TS\Gamma \triangleright s : R_2T\sigma$ eine Instanz von $TS\Gamma \triangleright s : T\sigma$. Daher ist auch

$$R\Gamma \triangleright (let\ x = s\ in\ t) : \varrho \quad \equiv \quad R_2TS\Gamma \triangleright (let\ x = s\ in\ t) : R_2\tau$$

eine Instanz von $TS\Gamma \triangleright (let\ x = s\ in\ t) : \tau$, und dies eine Haupttypisierung modulo Γ.

□

Beispiel 5.3.3 Als Haupttypisierung von $(let\ x = \lambda y.y\ \text{in}\ xx)$ modulo $\Gamma = \emptyset$ errechnet man

$$\Gamma \triangleright (let\ x = \lambda y.y\ in\ xx) : \alpha \to \alpha$$

wie folgt:

(i) $W(\emptyset, \lambda y.y) = (Id, \alpha \to \alpha)$

(ii) $W(\emptyset \cup \{x : \forall\alpha.\alpha \to \alpha\}, xx) = (T, \tau)$, wobei (T, τ) aus der Typisierung der Applikation über

$$\begin{array}{lcl} W(x : \forall\alpha.\alpha \to \alpha, x) & = & (Id, \beta \to \beta), \\ W(x : \forall\alpha.\alpha \to \alpha, x) & = & (Id, \alpha \to \alpha) \\ (T, \tau) & = & (U\ Id\ Id, U\beta), \\ & & \text{für } U \ = \ mgu(\beta \to \beta, (\alpha \to \alpha) \to \gamma^{neu}) \\ & & \quad = \ [\alpha \to \alpha/\beta, \alpha \to \alpha/\gamma] \end{array}$$

bestimmt werden, d.h. o.E. $(T, \tau) = (U, \alpha \to \alpha)$.

Bemerkung 5.3.4 Polytypen erlauben $\forall$–Quantoren nur in Präfixposition. Durch explizite Abquantifizierung freier Typvariablen gemäß

$$\frac{\Gamma \triangleright e : \sigma}{\Gamma \triangleright e : \forall\alpha.\sigma}, \quad \alpha \notin frei(\Gamma)$$

könnte man $\forall$–Quantoren auch „innerhalb“ von Typausdrücken zulassen, so daß z.B.

$$\forall\alpha(\beta \to (\alpha \to \beta)) \quad \equiv \quad \beta \to \forall\alpha(\alpha \to \beta).$$

Die Typen von *ML*–typisierbaren Termen bilden dann eine *echte* Teilklasse der Typausdrücke, in denen $\forall$ nur *positiv* vorkommt:

$$(\forall\alpha.\alpha \to \sigma) \to \tau$$

wäre z.B. kein *ML*–herleitbarer Typ.

Bemerkung 5.3.5 Partiell typisierte Terme

$$(\to I)_{poly} \qquad \frac{\Gamma_x, x : \overline{\sigma} \triangleright e : \tau}{\Gamma_x \triangleright \lambda x : \overline{\sigma}.e : \overline{\sigma} \to \tau}$$

erlaubt Typsynthese mit „polymorphem λ“. Dann kann man zwar immer noch keinen Haupttyp für Funktionale wie

$$F = \lambda f.(f[0,1,2], f[„ab“, „cde“])$$

(aus der Einleitung zu Kap.5) herleiten, aber immerhin für alle vom Programmierer vorgegebenen Spezialisierungen der Form

$$F_\sigma = \lambda f : \overline{\sigma}.(f[0,1,2], f[„ab“, „cde“]).$$

Mit $(\rightarrow I)_{poly}$ würde man beispielsweise herleiten:

$\triangleright\ F_{\forall\alpha(\alpha\ \texttt{list}\rightarrow\alpha\ \texttt{list})}\ :\ \forall\alpha(\alpha\ \texttt{-list}\rightarrow\alpha\ \texttt{-list})\ \rightarrow\ \texttt{int -list}\ \times\ \texttt{string list};$
$\triangleright\ F_{\forall\alpha(\alpha\ \texttt{list}\rightarrow\alpha)}\ :\ \forall\alpha(\alpha\ \texttt{-list}\rightarrow\alpha)\ \rightarrow\ \texttt{int}\ \times\ \texttt{string}.$

Während (*let*) dazu dient, den Argument(poly-)typ $\overline{\sigma}$ von F aus dem Kontext, d.h. f in $(F \cdot f)$, zu synthetisieren, wird er bei $(\rightarrow I)_{poly}$ vom Programmierer vorgegeben. Der Vorteil ist, daß die F_σ unabhängig vom Kontext ihrer Verwendung definiert werden können.

Literatur

[1] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[2] Roger Hindley. The completeness theorem for typing λ–terms. *Theoretical Computer Science*, 22:1–17, 1983.

[3] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[4] Robin Milner, Robert Harper, and Mads Tofte. *The Definition of Standard ML*. MIT Press, 1990.

Kapitel 6

Typsicherheit von *ML*

Wir wollen nun zeigen, daß *ML*-Programme typsicher sind, also zur Laufzeit des Programms keine Typfehler auftreten. Dazu gehört einerseits, daß bei der Auswertung von Termen die Typen erhalten bleiben (Subject Reduction). Andererseits soll aber die Auswertung eines getypten Terms nicht „steckenbleiben“, weil eine Funktion auf ein Argument angewendet wird, das nicht in ihrem Definitionsbereich liegt.

Die zweite Forderung bedeutet insbesondere, daß die Typisierung der Grundfunkionen (Konstanten) der Sprache sicherstellen muß, daß diese auf allen Objekten, die ihren Argumenttyp haben, definiert ist, während sie auf anderen Objekten nicht definiert zu sein braucht.

(Wenn wir wie bisher eine totale Applikation $\cdot$ annehmen, werden alle Funktionen total; in diesem Fall müssen z.B. arithmetische Funktionen wie + als Werte bei nicht-arithmetischen Argumenten ein Fehlerobjekt liefern. Es ist realistischer, die Grundfunktionen der Programmiersprache als partiell anzunehmen. Typsicherheit heißt, daß die Auswertung wohlgetypter Ausdrücke im ersten Fall “kein Fehlerobjekt erzeugt”, im zweiten Fall “nicht steckenbleibt”.)

6.1 Operationale Semantik

Wir betrachten nun Auswertungsbegriffe, die sich von der β-Reduktion des λ-Kalküls etwas unterscheiden und näher an der operationalen Semantik von Programmiersprachen liegen. Man definiert sie durch Reduktionsregeln, die auf Teilterme in bestimmten Kontexten angewendet werden dürfen.

Durch einen Kontext wird ein Vorkommen eines Teilausdrücken in einem Ausdruck bestimmt. Dazu betrachtet man eine Variante der Ausdrücke, in denen ein Symbol $\Box$ ein bestimmtes Vorkommen eines Teilausdrucks repräsentiert - dieses Symbol darf nur einmal auftreten:

Definition 6.1.1 (Kontexte von *ML*-Ausdrücken)

$$\begin{array}{lllll} (Context) & C & ::= & \Box \mid (C \cdot e) \mid (e \cdot C) \mid \langle C, e\rangle \mid \langle e, C\rangle \mid \\ & & & \lambda x C \mid (let\ x = C\ in\ e) \mid (let\ x = e\ in\ C) \mid \\ & & & (if\ C\ then\ e\ else\ e) \mid (if\ e\ then\ C\ else\ e) \mid (if\ e\ then\ e\ else\ C) \end{array}$$

Wir schreiben $C[e]$ für den Ausdruck, der aus C durch Ersetzen von $\Box$ durch e entsteht, wobei die Ersetzung im Sinne von Zeichenreihen gemeint ist, d.h. es erfolgt *keine* Umbennung von Variablen wie bei $C[e/\Box]$!

Definition 6.1.2 Sei $\rightarrow$ eine zweistellige Relation auf der Menge der Ausdrücke. Mit $\twoheadrightarrow_C$ oder kurz $\twoheadrightarrow$ sei die kleinste reflexive, transitive Relation gemeint, die $\rightarrow$ umfaßt und mit Kontexten C verträglich ist:

$$(Axiome) \qquad e \twoheadrightarrow e$$

$$(Regeln) \qquad \frac{e_1 \rightarrow e_2}{e_1 \twoheadrightarrow e_2} \qquad \frac{e_1 \twoheadrightarrow e_2, \quad e_2 \twoheadrightarrow e_3}{e_1 \twoheadrightarrow e_3} \qquad \frac{e_1 \twoheadrightarrow e_2}{C[e_1] \twoheadrightarrow C[e_2]}$$

Entsprechend wird $\twoheadrightarrow_C$ für andere Grundrelationen $\rightarrow$ und Kontextbegriffe C definiert.

6.2 Typsicherheit des funktionalen Kerns von *ML*

Wir definieren eine Auswertung von Ausdrücken, bei der die Werte eine Teilklasse der Ausdrücke sind. Als Werte, d.h. schon ausgewertete Ausdrücke, gelten Konstante, Variable, Paare und Funktionsausdrücke; einfachheitshalber benutzen wir statt der Ausdrücke $(rec\ x.e)$ eine Konstante Y für den Fixpunktoperator. Noch weiter auszuwertende Ausdrücke (Redexe) sind Anwendungen, Abkürzungen und Fallunterscheidungen. Als (Auswertungs-) Kontexte werden nur Kontexte einer besonderen Form zugelassen:

Definition 6.2.1

(Konstante)	c	$::=$	$true \mid false \mid \pi_1 \mid \pi_2 \mid Y$
(Expression)	e	$::=$	$c \mid x \mid \lambda x e \mid \langle e, e\rangle \mid$
			$(e \cdot e) \mid (let\ x = e\ in\ e) \mid (if\ e\ then\ e\ else\ e)$
(Value)	v	$::=$	$c \mid x \mid \lambda x e \mid \langle v, v\rangle$
(Redex)	r	$::=$	$(v \cdot v) \mid (let\ x = v\ in\ e) \mid (if\ v\ then\ e\ else\ e)$
(Kontext)	E	$::=$	$\Box \mid (E \cdot e) \mid (v \cdot E) \mid \langle E, e\rangle \mid \langle v, E\rangle \mid$
			$(let\ x = E\ in\ e) \mid (if\ E\ then\ e\ else\ e)$

Da $\Box$ in Auswertungskontexten nicht im Wirkungsbereich von Bindungsoperatoren vorkommt, gibt es bei $E[e]$ keine Variablenkonflikte.

Lemma 6.2.2 Jeder Ausdruck e ist entweder ein Wert oder von der Form $E[r]$, wobei der Auswertungskontext E und der Auswertungsredex r eindeutig bestimmt sind.

Beweis: Durch Induktion über E sieht man, daß für jeden Redex r der Ausdruck $E[r]$ kein Wert ist. Für Kontexte $E' \not\equiv \Box$ ist folglich ein Ausdruck $E'[r']$ kein Redex. Ist $\Box[r] \equiv E'[r']$, so muß auch $E' \equiv \Box$ und $r' \equiv r$ sein, da $E'[r'] \equiv r$ ein Redex ist. Da $E[r]$ kein Wert ist, ist $(E[r] \cdot e) \not\equiv (v \cdot E'[r'])$ und $\langle E[r], e\rangle \not\equiv \langle v, E'[r']\rangle$. Daher folgt die Eindeutigkeit der Zerlegung in den übrigen Fällen aus der Form der Kontexte und der Induktionsannahme.

Es genügt also, für jedes $e \not\equiv v$ eine Zerlegung $e \equiv E[r]$ in Kontext und Redex anzugeben:

Fall $e \equiv (e_1 \cdot e_2)$ und e_1 ist kein Wert: Dann gibt es eine eindeutige Zerlegung $e_1 \equiv E_1[r_1]$ und daher ist $e \equiv (E_1 \cdot e_2)[r_1]$ die gesuchte Zerlegung.

Fall $e \equiv (v_1 \cdot e_2)$ und e_2 ist kein Wert: Dann gibt es eine eindeutige Zerlegung $e_2 \equiv E_2[r_2]$ und daher ist $e \equiv (v_1 \cdot E_2)[r_2]$ die gesuchte Zerlegung.

Fall $e \equiv (v_1 \cdot v_2)$: Dann ist e ein Redex und $e \equiv \Box[e]$.

Fall $e \equiv \langle e_1, e_2 \rangle$: Das geht entsprechend.

Fall $e \equiv (\mathit{let}\ x = e_1\ \mathit{in}\ e_2)$ und e_1 ist kein Wert: Es gibt eine eindeutige Zerlegung $e_1 \equiv E_1[r_1]$, und daher ist $e \equiv (\mathit{let}\ x = E_1\ \mathit{in}\ e_2)[r_1]$ die gesuchte Zerlegung.

Fall $e \equiv (\mathit{let}\ x = v_1\ \mathit{in}\ e_2)$: Dann ist e ein Redex und $e \equiv \Box[e]$ die gesuchte Zerlegung.

Fall $e \equiv (\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$ und e_0 ist kein Wert: Dann gibt es eine eindeutige Zerlegung $e_0 \equiv E_0[r_0]$, und $e \equiv (\mathit{if}\ E_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2)[r_0]$ ist die gesuchte Zerlegung.

Fall $e \equiv (\mathit{if}\ v_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$: Dann ist e ein Redex und $e \equiv \Box[e]$ die gesuchte Zerlegung. $\Box$

Wegen der Eindeutigkeit der Zerlegung ergibt sich aus der Form der Auswertungskontexte, daß –in deterministischer Weise– von außen nach innen und von links nach rechts ausgewertet wird („leftmost-outermost reduction“):

Bei einer Anwendung muß zuerst der in der Funktionsposition stehende Teilausdruck ausgewertet werden; erst wenn dort ein Wert steht, darf in der Argumentposition ausgewertet werden. Bei einem geordneten Paar wird zuerst der linke, dann der rechte Teilausdruck ausgewertet, und bei einer Abkürzung zuerst der definierende Teilausdruck. Bei einer Fallunterscheidung muß die Bedingung zu *true* oder *false* ausgewertet sein, bevor der Teilausdruck für den entsprechenden Fall ausgewertet werden darf.

Definition 6.2.3 (Operationale Semantik von *ML*-Ausdrücken) Seien *Const* die Menge der Konstanten c und *Value* die Menge der Werte v. Sei $\delta : \mathit{Const} \times \mathit{Value} \to \mathit{Value}$ eine partielle Anwendungsfunktion mit $\mathit{frei}(\delta(c, v)) \subseteq \mathit{frei}(v)$. Definiere die Relation $\to$ wie folgt:

$(if)_1$	$(\mathit{if\ true\ then}\ e_1\ \mathit{else}\ e_2)$	$\to$	e_1	
$(if)_2$	$(\mathit{if\ false\ then}\ e_1\ \mathit{else}\ e_2)$	$\to$	e_2	
(π_1)	$(\pi_1 \cdot \langle v_1, v_2 \rangle)$	$\to$	v_1	
(π_2)	$(\pi_2 \cdot \langle v_1, v_2 \rangle)$	$\to$	v_2	
(Y)	$(Y \cdot v)$	$\to$	$(v \cdot (\lambda x.Yvx))$	mit frischem $x \notin \mathit{frei}(v)$
(δ)	$(c \cdot v)$	$\to$	$\delta(c, v)$,	falls $\delta(c, v)$ definiert ist
(β_v)	$(\lambda x e \cdot v)$	$\to$	$[v/x]e$	
(let)	$(\mathit{let}\ x = v\ \mathit{in}\ e)$	$\to$	$[v/x]e$	

Bei (δ) sei $c \notin \{Y, \pi_1, \pi_2\}$. Auf den *ML*-Ausdrücken definieren wir durch

$$E[e] \mapsto E[e'] : \Longleftrightarrow\ e \to e'.$$

eine Relation $\mapsto$. Beachte, daß $\mathit{frei}(E[e]) \subseteq \mathit{frei}(E[e'])$. Mit $\mapsto\!\!\!\!\to$ ist die durch $\mapsto$ und Auswertungskontexte nach Definition 6.1.2 erzeugte Relation gemeint.

Ein Redex $(Y \cdot v)$ wird statt zu $(v \cdot (Yv))$ zum Ausdruck $(v \cdot \lambda x(Yvx))$ reduziert, in dem v auf einen Wert angewandt wird, weshalb ein Redex entsteht. Man kann also anschließend nicht direkt

den eingebetteten Redex (Yv) kontrahieren und dadurch in eine Schleife geraten. Beachte auch, daß eine Funktionsanwendung nur ausgeführt werden darf, wenn das Argument ausgewertet ist („call-by-value“).

Nicht für alle Redexe wurde eine Reduktionsregel angegeben: Redexe der Form $(x \cdot v)$ mit Variable x, $(c \cdot v)$ wo $\delta(c, v)$ undefiniert ist, $(\pi_i \cdot v)$ mit $v \neq \langle v_1, v_2 \rangle$, $(\langle v_1, v_2 \rangle \cdot v)$ und $(\mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$ mit $v \notin \{\mathit{true}, \mathit{false}\}$ sind *nicht ausführbare Redexe.*

Definition 6.2.4 Der Ausdruck e *klemmt*, falls e kein Wert ist, es aber auch kein e' mit $e \to e'$ gibt. Ein Redex ist *fehlerhaft*, falls er nicht ausführbar und nicht von der Form $(x \cdot v)$, $(\pi_1 \cdot x)$, $(\pi_2 \cdot x)$, oder $(\mathit{if}\ x\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$ mit einer Variablen x ist. Ein *ML*-Ausdruck e heißt *fehlerhaft*, falls er einen fehlerhaften Redex enthält.

Es ist nicht entscheidbar, ob e zu einem klemmenden Ausdruck reduzierbar ist. Deshalb interessieren wir uns für die Obermenge der fehlerhaften Ausdrücke, obwohl auch fehlerhafte Ausdrücke auswertbar sein können:

Beispiel 6.2.5 $e := (\lambda x.2) \cdot \lambda x(1 + \texttt{true})$ ist fehlerhaft, da $(1 + \texttt{true}) \equiv (+ \cdot \langle 1, \mathit{true} \rangle)$ undefiniert ist, wobei + die Addition auf ganzen Zahlen sei. Die Auswertung ergibt aber 2.

Wir lassen im Folgenden neben den in Definition 5.1.1 angegebenen noch weitere Konstanten zu. Die Typannahmen für die Konstanten unterliegen aber gewissen Einschränkungen:

Definition 6.2.6 Der Grundkontext Γ_0 mit den Typannahmen für die Konstanten der Sprache erfülle folgende Eigenschaften:

(i) Γ_0 enthält $Y : \forall\alpha\forall\beta\ ([(\alpha \to \beta) \to (\alpha \to \beta)] \to (\alpha \to \beta))$ und

$$\mathit{true} : \texttt{boole}, \quad \mathit{false} : \texttt{boole}, \quad \pi_1 : \forall\alpha\forall\beta(\alpha * \beta \to \alpha), \quad \pi_2 : \forall\alpha\forall\beta(\alpha * \beta \to \beta).$$

(ii) Für jede Konstante c ist $\mathit{frei}(\Gamma_0(c)) = \emptyset$.

(iii) Für kein c und Typen σ, τ ist $\Gamma_0(c) \succ \sigma * \tau$.

(iv) Für jedes c mit $\Gamma_0(c) \succ \texttt{boole}$ ist $c \in \{\mathit{true}, \mathit{false}\}$.

(v) Für jedes Konstante c und Typen σ, τ mit $\Gamma_0(c) \succ \sigma \to \tau$ gilt: ist v ein Wert und Γ eine Umgebung mit $\Gamma \vdash_{ML} v : \sigma$, so ist $\delta(c, v)$ definiert und $\Gamma \vdash_{ML} \delta(c, v) : \tau$.

Proposition 6.2.7 (i) Ist $\Gamma \vdash_{ML} E[e] : \tau$, so gibt es einen Typ τ' mit $\Gamma \vdash_{ML} e : \tau'$

(ii) Ist $\Gamma \vdash_{ML} C[e] : \tau$, so gibt es $\Gamma' \supseteq \Gamma$ und einen Typ τ' mit $\Gamma' \vdash_{ML} e : \tau'$.

Beweis: (ii) durch Induktion über die Kontexte C nach 6.1.1.

(i) Da in Anwendungskontexten E die Stelle $\Box$ nicht im Wirkungsbereich eines Bindungsoperators vorkommt, braucht man bei der Typisierung des Teilausdrucks e die Umgebung Γ nicht zu erweitern. $\Box$

Lemma 6.2.8 Falls e fehlerhaft ist, gibt es kein Γ und keinen Typ τ mit $\Gamma \vdash_{ML} e : \tau$.

Beweis: Angenommen, es sei $\Gamma \vdash_{ML} e : \tau$ und r ein nicht ausführbarer Redex in e. Nach 6.2.7 gibt es Γ' und τ' mit $\Gamma' \vdash_{ML} r : \tau'$. Wir zeigen, daß r und damit e nicht fehlerhaft ist.

Fall $r \equiv (x \cdot v)$ und x ist eine Variable: Dann ist r nicht fehlerhaft.

Fall $r \equiv (c \cdot v)$ und $\delta(c, v)$ ist undefiniert: Die Herleitung der Typisierung $\Gamma' \rhd r : \tau'$ endet in

$$\frac{\begin{array}{c}\vdots\\ \Gamma' \rhd c : \sigma' \to \tau'\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma' \rhd v : \sigma'\end{array}}{\Gamma' \rhd (c \cdot v) : \tau'}$$

Nach der Annahme (v) über die Typisierung von Konstanten muß dann $\delta(c, v)$ definiert sein, ein Widerspruch.

Fall $r \equiv (\pi_i \cdot v)$ und $v \not\equiv \langle v_1, v_2 \rangle$: Die Herleitung der Typisierung $\Gamma' \rhd r : \tau'$ endet in

$$\frac{\begin{array}{c}\vdots\\ \Gamma' \rhd \pi_i : \sigma_1 * \sigma_2 \to \sigma_i\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma' \rhd v : \sigma_1 * \sigma_2\end{array}}{\Gamma' \rhd \pi_i \cdot v : \sigma_i.}$$

Nach Annahme (ii) über den Grundkontext ist v keine Konstante, und nach den Typregeln kann v auch kein Ausdruck der Form λxt sein. Also ist v eine Variable x, und r ist nicht fehlerhaft.

Fall $r \equiv (\langle v_1, v_2 \rangle \cdot v)$: Die Herleitung von $\Gamma' \rhd r : \tau'$ muß in

$$(\to I)\ \frac{\begin{array}{c}\vdots\\ \Gamma' \rhd \langle v_1, v_2 \rangle : \sigma \to \tau'\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma' \rhd v : \sigma\end{array}}{\Gamma' \rhd (\langle v_1, v_2 \rangle \cdot v) : \tau'}$$

enden. Das ist aber nicht möglich, da $\langle v_1, v_2 \rangle$ nicht den in der linken Teilherleitung angegebenen Typ haben kann.

Fall $r \equiv (\mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$ mit $v \notin \{\mathit{true}, \mathit{false}\}$: Die Herleitung der Typisierung $\Gamma' \rhd r : \tau'$ endet in

$$\frac{\begin{array}{c}\vdots\\ \Gamma' \rhd v : \texttt{boole}\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma' \rhd e_1 : \tau'\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma' \rhd e_2 : \tau'\end{array}}{\Gamma' \rhd (\mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2) : \tau'.}$$

Nach Annahme (iv) über den Grundkontext ist v keine Konstante, und nach den Typregeln kein Ausdruck der Form λxt. Also ist v eine Variable x, und r ist nicht fehlerhaft. □

Lemma 6.2.9 (Subject Reduction) Falls $\Gamma \vdash_{ML} e : \tau$ und $e \to e'$, so gilt auch $\Gamma \vdash_{ML} e' : \tau$.

Beweis: durch Induktion über die Reduktion $e \to e'$.

Fall $(c \cdot v) \to \delta(c, v)$: Die Typisierung endet in

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \rhd c : \sigma \to \tau, & \Gamma \rhd v : \sigma\end{array}}{\Gamma \rhd (c \cdot v) : \tau}$$

Nach der Voraussetzung über die Typisierung von Konstanten ist $\delta(c, v)$ definiert und man hat $\Gamma \vdash_{ML} \delta(c, v) : \tau$.

Fall $(Y \cdot v) \to v \cdot (\lambda x.Yvx)$: Die Typisierung endet in

$$(\to E)\ \frac{\begin{array}{cc}\vdots & \vdots \text{ a)} \\ \Gamma \rhd Y : [(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)] \to (\tau_1 \to \tau_2), & \Gamma \rhd v : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)\end{array}}{\Gamma \rhd Yv : \tau_1 \to \tau_2},$$

nach der Typannahme zu Y. Durch Abschwächungen erhält man daher auch eine Ableitung der Form

$$(\to E)\ \frac{\begin{array}{cc}\vdots \text{ a)} & \\ \Gamma \rhd v : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2) & \dfrac{\dfrac{\begin{array}{cc}\vdots & \\ \Gamma, x : \tau_1 \rhd Yv : \tau_1 \to \tau_2, & \Gamma, x : \tau_1 \rhd x : \tau_1\end{array}}{\Gamma, x : \tau_1 \rhd (Yv)x : \tau_2}}{\Gamma \rhd \lambda x(Yvx) : \tau_1 \to \tau_2}\end{array}}{\Gamma \rhd v \cdot \lambda x(Yvx) : \tau_1 \to \tau_2}$$

Fall $(let\ x = v\ in\ e) \to [v/x]e$: Die Typisierung endet in

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \rhd v : \sigma, & \Gamma, x : \overline{\sigma}^{\Gamma} \rhd e : \tau\end{array}}{\Gamma \rhd (let\ x = v\ in\ e) : \tau}.$$

Aus dem Substitutionslemma 5.2.8 folgt $\Gamma \vdash_{ML} [v/x]e : \tau$.

Die übrigen Fälle bleiben dem Leser überlassen. □

Die Auswertung eines geschlossenen Ausdrucks führt entweder zu einem fehlerhaften (ggf. weiter auswertbaren) Ausdruck oder zu einem Wert ohne freie Variable, oder sie divergiert:

Lemma 6.2.10 Sei $frei(e) = \emptyset$. Dann gilt eine der folgenden Aussagen: (i) es gibt ein fehlerhaftes e' mit $e \mapsto\!\!\!\!\to e'$, oder (ii) es gibt einen Wert v mit $e \mapsto\!\!\!\!\to v$, oder (iii) die Auswertung divergiert, $e \Uparrow$, d.h. es gibt eine unendliche Reduktionsfolge $e \mapsto e_1 \mapsto \ldots \mapsto e_n \mapsto e_{n+1} \mapsto \ldots$.

Beweis: Angenommen, (iii) trifft nicht zu. Wir zeigen, daß e entweder fehlerhaft oder ein Wert ist, oder daß es ein geschlossenes e' mit $e \mapsto e'$ gibt. Durch Induktion über die Länge der von e' ausgehenden Reduktionsfolge folgt dann die Behauptung.

Fall $e \equiv c, Y, \lambda x e'$: Dann ist e ein Wert.

Fall $e \equiv x$: Der Fall tritt nicht auf, da e geschlossen ist.

Fall $e \equiv (\mathit{let}\ x = e_1\ \mathit{in}\ e_2)$ und e_1 ist kein Wert: Nach 6.2.7 ist $e_1 \equiv E_1[r_1]$ für einen Kontext E_1 und einen Redex r_1. Ist r_1 fehlerhaft, so auch e. Im anderen Fall ist $\mathit{frei}(r_1) \subseteq \mathit{frei}(e) = \emptyset$, da $\Box$ in E_1 nicht im Bereich eines Bindungsoperators vorkommt. Daher gibt es e'' mit $r_1 \to e''$. Also ist $e \equiv (\mathit{let}\ x = E_1\ \mathit{in}\ e_2)[r_1] \mapsto (\mathit{let}\ x = E_1\ \mathit{in}\ e_2)[e'']$, und dies ist ein geschlosser Ausdruck.

Fall $e \equiv (\mathit{let}\ x = e_1\ \mathit{in}\ e_2)$ und e_1 ist ein Wert: Dann ist $e \mapsto e'$ für $e' \equiv [e_1/x]e_2$ geschlossen.

Fall $e \equiv (\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$ und e_0 ist kein Wert: das geht analog.

Fall $e \equiv (\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$ und e_0 ist ein Wert: Falls $e_0 \notin \{\mathit{true}, \mathit{false}\}$ ist, ist e wegen $\mathit{frei}(e_0) = \emptyset$ fehlerhaft. Im anderen Fall ist $e \mapsto e_i$ für ein i.

Fall $e \equiv (e_1 \cdot e_2)$ und e_1 ist kein Wert: Nach 6.2.7 ist $e_1 \equiv E_1[r_1]$ mit Redex r_1. Ist r_1 fehlerhaft, so ist es auch e. Im anderen Fall ist wieder $\mathit{frei}(r_1) = \emptyset$, und es gibt ein e'' mit $r_1 \to e''$, mit $\mathit{frei}(e'') \subseteq \mathit{frei}(r_1)$. Also ist $e \equiv (E_1 \cdot e_2)[r_1] \mapsto (E_1 \cdot e_2)[e'']$, ein geschlossener Ausdruck.

Fall $e \equiv (e_1 \cdot e_2)$ und e_1 ist ein Wert: Falls e_2 kein Wert ist, gibt es nach 6.2.2 eine Zerlegung $e_2 \equiv E_2[r_2]$. Ist darin der Redex r_2 fehlerhaft, so ist auch e fehlerhaft. Im anderen Fall enthält r_2 wie oben keine freie Variable, und daher gibt es ein e'' mit $r_2 \to e''$. Dann ist $e \equiv (e_1 \cdot E_2)[r_2] \mapsto (e_1 \cdot E_2)[e'']$ geschlossen. Falls schließlich auch e_2 ein Wert ist, so ist $e \equiv (e_1 \cdot e_2) \equiv \Box[e]$ ein Redex, und entweder fehlerhaft oder reduzierbar, da $e_i \not\equiv x$ ist.

Entsprechend behandelt man den Fall $\langle e_1, e_2 \rangle$. $\Box$

Nun können wir zeigen, daß typisierbare geschlossene Ausdrücke zu einem Wert vom gleichen Typ reduzierbar sind, oder divergieren (was nur eintreten kann, wenn sie Y enthalten). Es tritt kein Laufzeittypfehler auf, d.h. die Auswertung führt nicht über fehlerhafte Ausdrücke:

Satz 6.2.11 Falls $\vdash_{ML} e : \tau$, so ist entweder $e \Uparrow$ oder es gibt einen Wert v mit $e \mapsto\!\!\!\!\to v$ und $\vdash_{ML} v : \tau$

Beweis: Nach Lemma 6.2.10 gibt es entweder ein fehlerhaftes e' mit $e \mapsto\!\!\!\!\to e'$, oder einen Wert v mit $e \mapsto\!\!\!\!\to v$, oder $e \Uparrow$, die Auswertung divergiert. Nach Lemma 6.2.9 muß in den ersten beiden Fällen auch $\vdash_{ML} e' : \tau$ beziehungsweise $\vdash_{ML} v : \tau$ gelten. Da das für fehlerhafte e' aber nach Lemma 6.2.8 ausgeschlossen ist, gilt die Behauptung. $\Box$

6.3 Haupttypen und Typsicherheit für *ML* mit Referenzen

Die Sprache *SML* hat neben dem funktionalen Kern eine Reihe weiterer Ausdrucksmöglichkeiten, von denen hier noch die Möglichkeit betrachtet werden soll, den Speicher zu verändern.

Dazu braucht man Namen für Speicheradressen und die Möglichkeit, Werte an der benannten Stelle des Speichers abzulegen, dort abzulesen, oder durch andere Werte zu ersetzen. Man ordnet einer benannten Stelle des Speichers einen Typ zu und möchte zur Compilezeit eines Programms

sicherstellen, daß dort zur Laufzeit nur Werte des entsprechenden Typs gespeichert und gelesen werden.

Zunächst sind die Definitionen von *ML*-Ausdrücken und Typen zu erweitern:

Definition 6.3.1 (Ausdrücke und Typen)

$$\begin{array}{lrcl}
\text{(Konstante)} & c & ::= & \texttt{true} \mid \texttt{false} \mid \pi_1 \mid \pi_2 \mid Y \mid \\
 & & & \texttt{ref} \mid \texttt{!} \mid := \mid () \\
\text{(Ausdrücke)} & e & ::= & c \mid x \mid \langle e, e\rangle \mid (e \cdot e) \mid \lambda x e \mid \\
 & & & (\mathit{let}\ x = e\ \mathit{in}\ e) \mid (\mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e) \\
 & & & \\
\text{(Basistypen)} & \iota & ::= & \texttt{boole} \mid \texttt{unit} \\
\text{(Monotypen)} & \tau & := & \alpha \mid \iota \mid (\tau \to \tau) \mid (\tau * \tau) \mid \tau\ \texttt{ref} \\
\text{(Polytypen)} & \overline{\sigma} & := & \tau \mid \forall\alpha.\overline{\sigma}
\end{array}$$

Dabei `ref` sowohl eine Individuenkonstante als auch ein einstelliger Typkonstruktor, der *hinter* seinem Argument steht. Im Folgenden schreiben wir einfacher $(\texttt{ref}\, v)$, $!\, x$, $(x := v)$ statt der Ausdrücke $(\texttt{ref} \cdot v)$, $(! \cdot x)$, $(:= \cdot v)$ mit expliziter Anwendung.

Definition 6.3.2 Zu der Grundumgebung Γ_0 nehmen wir zusätzlich zu den Bedingungen aus 6.2.6 noch folgende Typannahmen:

$$() : \texttt{unit}, \quad \texttt{ref} : \forall\alpha(\alpha \to \alpha\,\texttt{ref}), \quad ! : \forall\alpha(\alpha\,\texttt{ref} \to \alpha), \quad := : \forall\alpha((\alpha\,\texttt{ref} * \alpha) \to \texttt{unit}).$$

Für keine Konstante c und keinen Typ τ ist $\Gamma(c) \succ \tau\,\texttt{ref}$.

Definition 6.3.3 Ein *Speicher* μ (für Memory) ist ein endliches Gleichungssystem

$$\begin{array}{lcl}
x_1 & \simeq & v_1(x_1, \ldots, x_m) \\
\vdots & \vdots & \vdots \\
x_m & \simeq & v_m(x_1, \ldots, x_m)
\end{array} \tag{6.1}$$

wobei die $x_1, \ldots, x_m$ paarweise verschiedene Variable und die $v_1, \ldots, v_m$ Werte sind, deren freie Variable unter $x_1, \ldots, x_n$ vorkommen.

Wir betrachten μ auch als eine endliche Funktion und schreiben daher oft $\mu(x_i)$ statt v_i, und $x_i \in \mathit{dom}(\mu)$. Für Variable x und Werte v mit $\mathit{frei}(v) \subseteq \mathit{dom}(\mu) \cup \{x\}$ sei

$$\mu[x \simeq v] := \begin{cases} \mu \cup \{x \simeq v\}, & \text{falls } x \simeq v' \text{ für kein } v' \text{ in } \mu \text{ vorkommt} \\ (\mu - \{x \simeq v'\}) \cup \{x \simeq v\}, & \text{sonst.} \end{cases}$$

In *SML* ergibt die Auswertung von `val n = ref7` den Wert `7 : int ref` und die neue Typannahme `n : int ref`. Danach kann man durch `n := 4` den gespeicherten Wert durch 4 überschreiben, und durch `val y = !x` lesen. Dagegen wird bei der Anweisung `n := true` ein Typfehler erkannt.

Es ergibt sich aber ein Problem mit gespeicherten Werten von polymorphem Typ.

Beispiel 6.3.4 Mit den Typregeln von *SML* für Listen kann man folgenden Ausdruck typisieren:

```
let val L = ref [] in L := [4]; L := true :: (! L) end
```

Nach der Deklaration `val L = [] ref` würde der Typkontext um $\mathtt{L} : \forall\alpha(\alpha\,\mathtt{list})$ erweitert, so daß `L` bei der ersten Zuweisung den Typ `int list` annehmen kann, und dann bei `!L` und der zweiten Zuweisung den Typ `bool list`; das wäre dann auch der Typ des gesamten Ausdrucks. Zur Laufzeit wird aber bei `L` die inhomogene, also untypisierbare Liste `[true,4]` gespeichert und später vielleicht wieder gelesen. Das bisherige Typsystem mit den naheliegenden Typannahmen für Referenzen ist also nicht sicher.

In *SML*'93 wird durch ein etwas kompliziertes Vorgehen dafür gesorgt, daß das vorige Beispiel nicht typisierbar ist. Die neue Definition *SML*'97 (siehe Milner e.a.[1]) verwirklicht einen einfacheren Vorschlag von A.K. Wright[2], die Typisierung von Abkürzungen in zwei Regeln aufzuspalten:

$$(let)_{val} \quad \frac{\Gamma \rhd v : \sigma, \qquad \Gamma, x : \overline{\sigma}^{\Gamma} \rhd e : \tau}{\Gamma \rhd (let\ x = v\ in\ e) : \tau}$$

$$(let)_{exp} \quad \frac{\Gamma \rhd e' : \sigma, \qquad \Gamma, x : \sigma \rhd e : \tau}{\Gamma \rhd (let\ x = e'\ in\ e) : \tau}, \quad \text{falls } e' \text{ kein Wert ist.}$$

Die Auswertung von Ausdrücken, die `ref`, `!` oder `:=` enthalten, muß die Veränderung des Speichers berücksichtigen.

Definition 6.3.5 Definiere eine Relation $\to$ zwischen Paaren von Ausdrücken und Speichern:

(β_v)	$(\lambda x e \cdot v),\ \mu$	$\to$	$[v/x]e,\ \mu$	
(Y)	$(Y \cdot v),\ \mu$	$\to$	$(v \cdot (\lambda x.Yvx)),\ \mu$	mit neuem x
(let)	$(let\ x = v\ in\ e),\ \mu$	$\to$	$[v/x]e,\ \mu$	
$(if)_1$	$(if\ \mathtt{true}\ then\ e_1\ else\ e_2),\ \mu$	$\to$	$e_1,\ \mu$	
$(if)_2$	$(if\ \mathtt{false}\ then\ e_1\ else\ e_2),\ \mu$	$\to$	$e_2,\ \mu$	
(π_1)	$(\pi_1 \cdot \langle v_1, v_2\rangle),\ \mu$	$\to$	$v_1,\ \mu$	
(π_2)	$(\pi_2 \cdot \langle v_1, v_2\rangle),\ \mu$	$\to$	$v_2,\ \mu$	
(ref)	$(\mathtt{ref} \cdot v),\ \mu$	$\to$	$x,\ \mu[x \simeq v],$	mit neuem x
(!)	$(! \cdot x),\ \mu$	$\to$	$v,\ \mu,$	falls $x \simeq v \in \mu$
(:=)	$(:= \cdot \langle x, v\rangle),\ \mu$	$\to$	$(),\ \mu[x \simeq v]$	

Die Auswertung wird jetzt ebenfalls für solche Paare definiert:

$$E[e], \mu \ \mapsto\ E[e'], \nu \quad :\Longleftrightarrow \quad e, \mu \ \to\ e', \nu.$$

Mit $\mapsto\!\!\!\!\to$ ist die durch $\mapsto$ und Auswertungskontexte nach Definition 6.1.2 erzeugte Relation gemeint.

Definition 6.3.6 Der Speicher μ *ist typisierbar in der Umgebung* Γ, kurz $\Gamma \vdash_{ML} \mu$, falls es für jedes $x \simeq v \in \mu$ einen Typ τ mit $\Gamma(x) = \tau\,\mathtt{ref}$ und $\Gamma \vdash_{ML} v : \tau$ gibt. Wir schreiben $\Gamma \vdash_{ML} e, \mu : \tau$, falls $\Gamma \vdash_{ML} \mu$, $\mathit{frei}(e) \subseteq \mathit{dom}(\mu)$, und $\Gamma \vdash_{ML} e : \tau$.

Lemma 6.3.7 Ist $\Gamma \vdash_{ML} r, \mu : \tau$ und r ein Redex mit $r, \mu \to e, \nu$, so gibt es eine Erweiterung $\Gamma' \supseteq \Gamma$ um Annahmen über Individuenvariable in $\mathit{dom}(\nu) - \mathit{dom}(\mu)$, so daß $\Gamma' \vdash_{ML} e, \nu : \tau$.

Beweis: Durch Fallunterscheidung nach der Form des Redexes.

Außer bei den Reduktionen (`ref`), (!) und (:=) gilt für $r, \mu \to e, \nu$ stets $\mathit{frei}(e) \subseteq \mathit{frei}(r)$ und $\nu = \mu$. Daher gilt nach Induktion auch $\Gamma \vdash_{ML} \nu$, und mit Satz 6.2.9 auch $\Gamma \vdash_{ML} e : \tau$.

Fall $(\mathtt{ref}\, v), \mu \to x, \mu[x \simeq v]$ mit frischem x: Nach Induktion ist $\mathit{frei}(\mathtt{ref}\, v) \subseteq \mathit{dom}(\mu)$, $\Gamma \vdash_{ML} \mu$ und es gibt eine Typherleitung für $(\mathtt{ref}\, v)$, die in

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \rhd \mathtt{ref} : \tau \to \tau\,\mathtt{ref}, & \Gamma \rhd v : \tau\end{array}}{\Gamma \rhd (\mathtt{ref}\, v) : \tau\,\mathtt{ref}}$$

endet. Da $x \notin \mathit{dom}(\mu)$ ist, folgt $\Gamma, x : \tau\,\mathtt{ref} \vdash_{ML} \mu[x \simeq v]$ mit der rechten Teilherleitung. Da $x \notin \mathit{dom}(\Gamma)$ ist, gelten auch $\Gamma, x : \tau\,\mathtt{ref} \vdash_{ML} x : \tau\,\mathtt{ref}$ und offenbar $x \in \mathit{dom}(\mu[x \simeq v])$. Zusammen zeigt das $\Gamma, x : \tau\,\mathtt{ref} \vdash_{ML} x, \mu[x \simeq v] : \tau\,\mathtt{ref}$.

Fall $!x, \mu \to v, \mu$ für $x \simeq v \in \mu$: Da μ ein Speicher ist, ist $\mathit{frei}(v) \subseteq \mathit{dom}(\mu)$. Nach Induktion ist $\Gamma \vdash_{ML} \mu$, und es gibt eine Typherleitung, die in

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \rhd\, ! : \tau\,\mathtt{ref} \to \tau, & \Gamma \rhd x : \tau\,\mathtt{ref}\end{array}}{\Gamma \rhd\, ! x : \tau}$$

endet. Wegen $x \simeq v \in \mu$, $\Gamma \vdash_{ML} \mu$ und der rechten Teilherleitung ist $\Gamma(x) = \tau\,\mathtt{ref}$ und $\Gamma \vdash_{ML} v : \tau$. Also gilt $\Gamma \vdash_{ML} v, \mu : \tau$.

Fall $(x := v), \mu \to (), \mu[x \simeq v]$: Nach Induktionssannahme ist $\mathit{frei}(x := v) \subseteq \mathit{dom}(\mu)$, $\Gamma \vdash_{ML} \mu$, und es gibt eine Typherleitung, die in

$$\frac{\begin{array}{cc}\vdots & \begin{array}{c}\vdots \\ \Gamma \rhd x : \sigma\,\mathtt{ref}\end{array}\quad\begin{array}{c}\vdots \\ \Gamma \rhd v : \sigma\end{array} \\ \Gamma \rhd :=\, : \sigma\,\mathtt{ref} * \sigma \to \mathtt{unit} & \dfrac{}{\Gamma \rhd \langle x, v\rangle : \sigma\,\mathtt{ref} * \sigma}\end{array}}{\Gamma \rhd (x{:=}v) : \mathtt{unit}}$$

endet. Wegen $x \in \mathit{dom}(\mu)$, $\Gamma \vdash_{ML} \mu$ und der mittleren Teilherleitung ist $\Gamma(x) = \sigma\,\mathtt{ref}$. Wegen $\mathit{frei}(v) \subseteq \mathit{dom}(\mu) = \mathit{dom}(\mu[x \simeq v])$ und der rechten Teilherleitung ist dann $\Gamma \vdash_{ML} \mu[x \simeq v]$. Da nach den Typannahmen für Konstante und Abschwächungen auch $\Gamma \vdash_{ML} () : \mathtt{unit}$ gilt, ist die Behauptung $\Gamma \vdash_{ML} (), \mu[x \simeq v] : \mathtt{unit}$ gezeigt. □

Damit können wir zeigen, daß auch bei der Verwendung von Referenzen die Typen bei der Auswertung von Ausdrücken erhalten bleiben:

Satz 6.3.8 (Subject Reduction) Ist $\Gamma \vdash_{ML} e, \mu : \tau$ und $e, \mu \ \mapsto \ e', \mu'$, so ist $\Gamma' \vdash_{ML} e', \mu' : \tau$ für eine Erweiterung $\Gamma' \supseteq \Gamma$ um Annahmen für Individuenvariable aus $dom(\mu') - dom(\mu)$.

Beweis: Nach Lemma 6.2.2 ist e ein Wert oder kann eindeutig in einen Auswertungskontext und einen Auswertungsredex zerlegt werden. Die Behauptung folgt daher aus der Aussage:

$$\Gamma \vdash_{ML} E[r], \mu : \tau \text{ und } r, \mu \ \to \ e', \mu' \quad \Rightarrow \quad \Gamma' \vdash_{ML} E[e'], \mu' : \tau \text{ für eine Erweiterung } \Gamma' \supseteq \Gamma.$$

Hierbei ist Γ' wieder eine Erweiterung um Annahmen für Individuenvariable aus $dom(\mu') - dom(\mu)$. Wir beweisen diese Aussage durch Induktion über E:

Fall $\Box[r], \mu \ \mapsto \ \Box[e'], \mu'$: Dieser Fall wurde in Lemma 6.3.7 behandelt.

Fall $\langle E[r], e\rangle, \mu \ \mapsto \ \langle E[e'], e\rangle, \mu'$: Dann ist $\Gamma \vdash_{ML} \mu$, $frei(E[r], e) \subseteq dom(\mu)$, und es gibt eine Herleitung

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \rhd E[r] : \tau_1 & \Gamma \rhd e : \tau_2\end{array}}{\Gamma \rhd \langle E[r], e\rangle : \tau_1 * \tau_2.}$$

mit $\tau \equiv \tau_1 * \tau_2$. Daher gilt $\Gamma \vdash_{ML} E[r] : \tau_1, \mu$, also nach Induktion auch $\Gamma' \vdash_{ML} E[e'] : \tau_1, \mu'$ für eine geeignete Erweiterung $\Gamma' \supseteq \Gamma$. Durch Abschwächungen erhält man aus der rechten Teilherleitung $\Gamma' \vdash_{ML} e : \tau_2$. Eine Anwendung von $(* \, I)$ ergibt $\Gamma' \vdash_{ML} \langle E[e'], e\rangle : \tau_1 * \tau_2$, woraus die Behauptung $\Gamma' \vdash_{ML} \langle E[e'], e\rangle : \tau_1 * \tau_2, \mu'$ folgt.

Fall $\langle v, E[r]\rangle, \mu \to \langle v, E[e']\rangle \mu'$: Analog.

Fall $(let \ x = E[r] \ in \ e), \mu \mapsto (let \ x = E[e'] \ in \ e), \mu'$: Wegen $\Gamma \vdash_{ML} (let \ x = E[r] \ in \ e), \mu : \tau$ ist $\Gamma \vdash_{ML} \mu$, und $\Gamma \vdash_{ML} (let \ x = E[r] \ in \ e) : \tau$. Da r ein Redex ist, ist $E[r]$ kein Wert (!), und deshalb endet die Herleitung hierfür in

$$(let)_{exp} \ \frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \rhd E[r] : \sigma & \Gamma, x : \sigma \rhd e : \tau\end{array}}{\Gamma \rhd (let \ x = E[r] \ in \ e) : \tau}.$$

Mit der linken Teilherleitung folgt $\Gamma \vdash_{ML} E[r], \mu : \sigma$, und daraus induktiv $\Gamma' \vdash_{ML} E[e'], \mu' : \sigma$ für eine geeignete Fortsetzung Γ' von Γ, insbesondere $\Gamma' \vdash_{ML} E[e'] : \sigma$. Aus der rechten Teilherleitung erhält man durch Abschwächungen $\Gamma', x : \sigma \vdash_{ML} e : \tau$ und daraus $\Gamma', x : \overline{\sigma}^{\Gamma'} \vdash_{ML} e : \tau$. Falls $E[e']$ kein Wert ist, so folgt $\Gamma' \vdash_{ML} (let \ x = E[e'] \ in \ e) : \tau$ durch eine Anwendung von $(let)_{exp}$. Falls $E[e']$ ein Wert v' ist, benutze man $(let)_{val}$. Daraus ergibt sich die Behauptung.

Die übrigen drei Fälle, $(E[r] \cdot e), \mu \to (E[e'] \cdot e), \mu'$, $(v \cdot E[r]), \mu \to (v \cdot E[e']), \mu'$, und schließlich $(if \ E[r] \ then \ e_1 \ else \ e_2), \mu \to (if \ E[e'] \ then \ e_1 \ else \ e_2), \mu'$, bleiben dem Leser überlassen. □

Jetzt können wir zeigen, daß *ML* typsicher ist, also zur Laufzeit keine Typfehler auftreten:

Satz 6.3.9 (*ML* ist typsicher) Falls $\Gamma \vdash_{ML} E[r], \mu : \tau$, so ist r ein ausführbarer Redex.

Beweis: Nach Voraussetzung ist $\Gamma \vdash_{ML} \mu$, $dom(\mu) = dom(\Gamma)$, und $\Gamma \vdash_{ML} E[r] : \tau$. Wir unterscheiden nach der Form des Redexes r: [1]

Fall $r \equiv (c \cdot v)$: Es gibt nach 6.2.7 einen Typ σ und eine Herleitung, die in

$$(\rightarrow I)\ \frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \rhd c : \rho \rightarrow \sigma & \Gamma \rhd v : \rho\end{array}}{\Gamma \rhd (c \cdot v) : \sigma}$$

endet. Wegen der rechten Teilherleitung ist $c \notin \{\texttt{true}, \texttt{false}, ()\}$.

Für $c \in \{\pi_1, \pi_2, :=\}$ ist $\rho = (\rho_1 * \rho_2)$ für geeignete Typen ρ_1, ρ_2. Der Wert v kann keine Konstante sein, da Konstante nach Voraussetzung nicht den Typ $(\rho_1 * \rho_2)$ annehmen können. Wegen $\Gamma \not\vdash Y : (\rho_1 * \rho_2)$ und $\Gamma \not\vdash \lambda xt : (\rho_1 * \rho_2)$ ist v weder Y noch ein λ-Ausdruck. Wäre $v \equiv x$ für eine Variable x, so wäre $\Gamma(x) = \tilde{\tau}\,\texttt{ref}$ für einen geeigneten Typ $\tilde{\tau}$, wegen $dom(\Gamma) \subseteq dom(\mu)$ und $\Gamma \vdash_{ML} \mu$, im Widerspruch zu $\Gamma \vdash_{ML} x : (\rho_1 * \rho_2)$. Also sind alle Werte v mit $\Gamma \vdash_{ML} v : (\rho_1 * \rho_2)$ von der Form $\langle v_1, v_2 \rangle$. Ist nun $c \equiv \pi_i$, $i = 1, 2$, so ist die Reduktionsregel (π_i) anwendbar. Ist $c \equiv :=$, so ist $\rho_1 \equiv \tilde{\tau}\,\texttt{ref}$ für einen Typ $\tilde{\tau}$, und es muß $\Gamma \vdash_{ML} v_1 : \tilde{\tau}\,\texttt{ref}$ gelten. Nach den Annahmen über die Typisierung der Konstanten kann v_1 dann keine Konstante sein, und nach den Typregeln ist es auch nicht von der Form $Y, \lambda x.t, \langle e_1, e_2 \rangle$. Also ist v_1 eine Variable x, so daß die Reduktionsregel $(:=)$ anwendbar ist.

Für $c \equiv\ !$ ist $\rho = \sigma\,\texttt{ref}$. Wegen $\Gamma \vdash_{ML} v : \sigma\,\texttt{ref}$ und der Annahme über die Typisierung der Konstanten kann v keine Konstante sein, und nach den Typregeln auch kein Wert der Form $Y, \lambda xt, \langle v_1, v_2 \rangle$. Ist v eine Variable x, so ist wegen $\Gamma \vdash_{ML} \mu$ schon $\Gamma(x) = \sigma\,\texttt{ref}$, und wegen $dom(\Gamma) \subseteq dom(\mu)$ ist die Reduktionsregel $(!)$ anwendbar.

Für $c \equiv \texttt{ref}$ ist $\sigma = \rho\,\texttt{ref}$, und man kann die Reduktionsregel $(\texttt{ref})$ anwenden.

Fall $r \equiv (x \cdot v)$: Benutze, daß $dom(\Gamma) \subseteq dom(\mu)$, um diese Redexe $(x \cdot v)$ auszuschließen.

Fall $r \equiv (Y \cdot v)$: Dann ist die Reduktionsregel (Y) anwendbar.

Fall $r \equiv (\lambda xe \cdot v)$: Man kann die Reduktionsregel (β_v) benutzen.

Fall $r \equiv (\mathit{let}\ x = v\ \mathit{in}\ e)$: Die Reduktionsregel (let) ist anwendbar.

Fall $r \equiv (\mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2)$: Nach 6.2.7 gibt es ein σ mit $\Gamma \vdash_{ML} (\mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2) : \sigma$. Nach den Typregeln muß dann auch $\Gamma \vdash_{ML} v : \texttt{boole}$ gelten. Da v ein Wert vom Typ `boole` ist, kann es nur eine Variable x oder eine der Konstanten `true,false` sein. Wäre $v \equiv x$, so wäre $\Gamma(x) \succ \texttt{boole}$, also $x \in dom(\Gamma) \subseteq dom(\mu)$. Dann müßte aber $\Gamma(x) = \tau\,\texttt{ref}$ sein, ein Widerspruch. Ist aber $v \in \{\texttt{true}, \texttt{false}\}$, so ist eine der Reduktionsregeln $(\mathit{if})_i$ anwendbar. □

Bemerkung: Felleisen/Wright[3] erlauben, daß bei der Auswertung von (ref e) eine namenlose Referenz erzeugt wird. Das entspricht genauer dem, was in SML passiert.

Aufgabe 6.3.1 Man erweitere den Typsynthesealgorithmus aus 5.3.1 für die in diesem Abschnitt hinzugenommenen Ausdrücke und spalte die Klausel für `let`-Ausdrücke in zwei Fälle, einen für das "monomorphe `let`" für Werte und einen für das bisherige "polymorphe `let`". Beweise, das man damit wieder Haupttypen berechnet.

[1] Man sollte hier einfach $\mathit{frei}(E[r]) \subseteq dom(\Gamma) \subseteq dom(\mu)$ voraussetzen; vorher braucht man das nicht wesentlich.

Literatur

[1] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[2] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Report tr93-200, Rice University, February 1993.

[3] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 1993(?).

Kapitel 7

Entscheidungskomplexität der *ML*–Typisierbarkeit

Die Typisierbarkeit im C–Kalkül kann effizient festgestellt und ein Haupttyp kann effizient gefunden werden, falls man zur Darstellung der Typausdrücke gerichtete kreisfreie Graphen verwendet. In der Praxis ist auch die *ML*–Typisierbarkeit und die Bestimmung eines *ML*–Haupttyps „effizient“ durchführbar, so daß mehrfach (ohne Beweise) behauptet wurde, dies sei mit linearem Zeitaufwand in der Größe des Eingabeprogramms möglich.

Eine genauere Untersuchung von Kanellakis/Mitchell[4] zeigte, daß das Typisierbarkeitsproblem für *ML* PSPACE-schwer ist, also nur im Fall P = PSPACE eine effiziente Lösung zuläßt, und daß DEXPTIME eine obere Komplexitätsschranke für die *ML*-Typisierbarkeit ist. Mairson[6] und Kfoury/Tiuryn/Urcyczyn[1990] zeigten schließlich, daß diese obere Schranke angenommen wird, d.h. daß *ML*–Typisierbarkeit DEXPTIME schwer ist.

Der Beweis von Mairson kodiert die Rechnung einer deterministischen Turing–maschine in die Berechnung der Typen geeigneter λ–Terme. Die Verwendung von „let“ wird nur dazu benötigt, exponentiell viel Platz auf dem Rand und exponentiell viele Anwendungen der Übergangsfunktion zu ermöglichen.

Die Technik wurde von F. Henglein[2] erweitert, und dazu verwendet, DEXPTIME als untere Schranke für das Typisierbarkeitsproblem im zweistufigen λ–Kalkül (Girard/Reynolds) nachzuweisen. Beim Beweis von Mairson's Satz stützen wir uns auf die Vereinfachung des Mairson'schen Beweises durch F. Henglein[3].

7.1 Entscheidungskomplexität von C–Typisierungen

Effiziente Verfahren zur Typisierung setzen voraus, daß die Typen nicht als Terme (Bäume), sondern als gerichtete kreisfreie Graphen (Dag's) dargestellt werden—diese entstehen aus den Bäumen, indem isomorphe Teilbäume miteinander identifiziert werden. Die Baumdarstellung des Haupttyps von t kann nämlich exponentiell groß in der Größe von t sein:

Beispiel 7.1.1 Sei $\langle e_1, \ldots, e_n \rangle := \lambda z.ze_1 \cdots e_n$ mit $z \notin \mathit{frei}(e_i)$. Sind $\Gamma \rhd e_i : \sigma_i$ Haupttypisierungen von e_i, so ist $\Gamma \rhd \langle e_1, \ldots, e_n \rangle : (\sigma_1 \to \ldots \to \sigma_n \to \alpha) \to \alpha$ eine Haupttypisierung von $e_1, \ldots, e_n$, mit „neuem“ α. Sei $\sigma_1 \times \ldots \times \sigma_n := (\sigma_1 \to \ldots \to \sigma_n \to \alpha) \to \alpha$, mit α nicht frei in $\sigma_1, \ldots, \sigma_n$. Da

$$\emptyset \rhd \lambda x.\langle x, x \rangle : \alpha \to \alpha \times \alpha$$

eine Haupttypisierung ist, folgt für $p := \lambda x.\langle x, x\rangle$: Ist $\Gamma \triangleright e : \sigma$ eine Haupttypisierung, so auch

$$\Gamma \triangleright (p \cdot e) : \sigma \times \sigma, \qquad \Gamma \triangleright p \cdot (p \cdot e) : (\sigma \times \sigma) \times (\sigma \times \sigma) \quad \text{usw.}$$

Es folgt

Proposition 7.1.2 Für jedes n gibt es einen geschlossenen λ–Term der Länge n, dessen Haupttypen die Länge $2^{\Omega(n)}$ haben.

Die Dag–Darstellung von Typausdrücken läßt dagegen zu, daß man Haupttypen in Linearzeit und mit in der Programmgröße linearen Dag-Größe berechnen kann.

Lemma 7.1.3 Es gibt einen Algorithmus, der die C-Typisierbarkeit von t modulo Γ in der Zeit $O(|t| + |\Gamma|_{dag})$ entscheidet, und im positiven Fall einen Haupttyp τ modulo Γ mit $|\tau|_{dag} = O(|t| + |\Gamma|_{dag})$ bestimmt.

Beweis: (Skizze) Es gibt Algorithmen (s. Paterson/Wegman[7]), die die Unifizierbarkeit von σ_1 und σ_2 in der Zeit $O(\mid \sigma_1 \mid_{dag} + \mid \sigma_2 \mid_{dag})$ auf einer deterministischen Turingmaschine entscheiden, und ggf. eine Dag–Darstellung der Lösung ausgeben, die ebenfalls linear in der Dag–Größe der Eingabe ist.

Der rekursive Algorithmus W aus Kapitel 4 macht jedoch mehr als $\mid t \mid$ viele Anwendungen der Unifikation, so daß man auch unter Verwendung von Dag–Darstellungen nur in polynomieller Zeit hinkommt. Daher modifiziere man ihn wie folgt:

(1) Sorge durch Umbenennungen dafür, daß λ–gebundene Variablen paarweise verschieden sind.

(2) Ordne jedem Teilterm (bzw. Dag) eine neue Typvariable zu.

(3) Stelle für jeden zusammemgesetzten Term eine Gleichung zwischen seiner Typvariablen und denen der direkten Teilterme auf; z.B.

$$(e_1^{\alpha} \cdot e_2^{\beta})^{\gamma} \;\mapsto\; \alpha = \beta \to \gamma$$

(4) Löse dieses Gleichungssystem (oBdA in eine Gleichung verpackt) durch *eine* Anwendung des linearen Unifikationsalgorithmus.

Man sieht leicht, daß (1)–(3) nur linearen Aufwand in $\mid t \mid$ erfordern, da der C–Kalkül „syntax–gerichtet“ ist. Das Gleichungssystem in (4) ist linear in $\mid t \mid$ und $\mid \Gamma \mid_{dag}$, so daß die Lösung durch die einmalige Anwendung der Unifikation in $O(\mid t \mid + \mid \Gamma \mid_{dag})$ erfolgt. □

7.2 Die Größe von *ML*–Haupttypen

Es ist nicht schwierig, den Linearzeit–Algorithmus aus 7.1 auch auf rec– und und if-then-else–Ausdrücke von *ML* auszudehnen. Die Schwierigkeit liegt erwartungsgemäß bei den let–Ausdrücken.

Beispiel 7.2.1 Sei $\Gamma \triangleright e : \sigma$ eine *ML*–Haupttypisierung und α die einzige Typvariable in σ, die in Γ nicht vorkommt. Dann ist mit neuen α_1, α_2

$$\Gamma \triangleright (\mathit{let}\ x = e\ \mathit{in}\ \langle x, x\rangle) : \sigma(\alpha_1) \times \sigma(\alpha_2)$$

eine *ML*–Haupttypisierung mod Γ.

Beachte, daß sich nicht nur die Länge des Typs, sondern auch die Anzahl seiner in Γ nicht vorkommenden Typvariablen verdoppelt hat. Dies erzwingt auch eine Vergrößerung der Dag–Darstellung (im Unterschied zum Haupttyp von $(\lambda x.\langle x, x\rangle) \cdot e$)

Beispiel 7.2.2 (Wand, Buneman) Sei

$$\begin{array}{lllll} s_n := & \mathit{let} & x_0 = \lambda y.y & & \\ & \mathit{in} & \mathit{let} & x_1 = \langle x_0, x_0\rangle & \\ & & \mathit{in} & & \\ & & & \ddots & \\ & & & & \mathit{let} \quad x_n = \langle x_{n-1}, x_{n-1}\rangle \\ & & & & \mathit{in} \quad x_n \end{array}$$

Die Dag–Dastellung des Haupttyps von s_n hat $2^{\Omega(n)}$ Knoten, da der Haupttyp so viele Variablen hat. Andere Typisierungen können weniger Knoten haben (z.B. alle Variablen identifizieren).

Beispiel 7.2.3 (Kanellakis/Mitchell) Sei

$$\begin{array}{lllll} t_n := & \mathit{let} & x_1 = \lambda y.\langle y, y\rangle & & \\ & \mathit{in} & \mathit{let} & x_2 = x_1 \circ x_1 & \\ & & \mathit{in} & & \\ & & & \ddots & \\ & & & & \mathit{let} \quad x_n = x_{n-1} \circ x_{n-1} \\ & & & & \mathit{in} \quad x_n \cdot \lambda y.y \end{array}$$

Der Haupttyp von t_n ist $(\alpha \to \alpha)^{[2^n]}$, mit $2^{2^{\Omega(n)}}$ Knoten in der Baum– und $2^{\Omega(n)}$ Knoten in der Dag–Darstellung. Da er nur eine Variable hat, ist *jeder* Typ von t_n von mindestens dieser Größe. (Beachte, daß mit $\sigma^{[1]} := \sigma, \sigma^{[n+1]} := \sigma^{[n]} \times \sigma^{[n]}$ hierbei $x_1 : \alpha \to \alpha^{[2]}, x_i : \alpha \to \alpha^{[2^i]}$.)

Bemerkung:

Die Typisierung von t_5 benötigt auf einer Sun 3/160 über 2 Minuten CPU-Zeit, 60 Megabyte Speicher und >170 Seiten Ausdruck für den Haupttyp.

Beachte, daß x_n in 7.2.3 der 2^n–fachen Komposition $\lambda y.x_0^{2^n}(y)$ von $x_0 = \lambda y.\langle y, y\rangle$ entspricht. Die Grundidee von Mairson's Beweis ist nun

- Ist x_0 das „Anhängen eines Felds“ an ein Turingband, so verschaftt x_n exponentiell viel Platz (in der Länge n von t_n)
- Ist x_0 die Übergangsfunktion einer Turingmaschine und y eine Konfiguration, so ist x_n die 2^n–fache Anwendung der Übergangsfunktion auf y.

7.3 Programmieren auf der Ebene der Typen

Wir wollen nun beliebige Berechnungen in die Ermittlung von Haupttypen geeigneter λ–Terme kodieren. Dazu wählen wir —nach Kanellakis/Mitchell[1989]— als „Programme“ λ-Terme der folgenden Form:

$$P = \lambda\overbrace{x_1 \ldots x_n}^{\text{Eingabe}}.\lambda\overbrace{y_1 \ldots y_m}^{\text{Ausgabe}}.\lambda z.Kz(\lambda\overbrace{z_1 \quad \ldots \quad z_k}^{\text{lokale Variable}} . \overbrace{e}^{\text{Befehle}}).$$

Dabei soll P ein geschlossener typisierbarer λ–Term sein. Wegen $K = \lambda xy.x$ ist

$$\lambda x_1 \ldots x_n.\lambda y_1 \ldots y_m.\lambda z.z =: nf(P)$$

die β–Normalform von P. Wir sind aber nicht am Wert von P, sondern am Haupttyp von P interessiert. Durch geeignete Wahl von e wird die Haupttypisierung

$$\emptyset \triangleright nf(P) : \alpha_1 \to \ldots \to \alpha_n \to \beta_1 \to \ldots \to \beta_m \to \gamma \to \gamma$$

in eine spezifischere Haupttypisierung

$$(*) \quad \emptyset \triangleright P : \tilde{\sigma}_1 \to \ldots \to \tilde{\sigma}_n \to \tilde{\tau}_1 \to \ldots \to \tilde{\tau}_m \to \gamma \to \gamma$$

überführt, wobei γ in $\tilde{\sigma}_i$ und $\tilde{\tau}_j$ nicht vorkommt. Die Haupttypisierung $(*)$ erlaubt es, P *als Funktion auf Typen* zu interpretieren:

$$(**) \quad P(\sigma_1, \ldots, \sigma_n) \quad := \quad \begin{cases} (\tau_1, \ldots, \tau_m), & \text{falls } S = mgu(\tilde{\sigma}_1 \to \ldots \to \tilde{\sigma}_n, \sigma_1 \to \ldots \to \sigma_n) \\ & \neq fail \text{ und } \tau_j = S\tilde{\tau}_j, j = 1, \ldots, m \\ \text{undefiniert}, & \text{falls } mgu(\tilde{\sigma}_1 \to \ldots \to \tilde{\sigma}_n, \sigma_1 \to \ldots \to \sigma_n) = fail \end{cases}$$

Hierbei ist *mgu* wieder der allgemeinste Unifikator. Um nun ein solches P während der Ermittlung des Haupttyps eines λ–Terms t „als Funktion auf Typen“ zu verwenden, brauchen wir einen Teilterm von t der Form $(Ps_1 \cdots s_n)$. Sind für $i = 1, \ldots, n$

$$\Gamma \triangleright s_i : \sigma_i$$

Haupttypisierungen von s_i, so ist durch die Typisierung von $(Ps_1, \ldots, s_n)$ modulo Γ, d.h. durch

$$W(\Gamma, Ps_1 \cdots s_n) = (S, \tau_1 \to \ldots \to \tau_m \to \gamma \to \gamma),$$

die Anwendung $P(\sigma_1, \ldots, \sigma_n) = (\tau_1, \ldots, \tau_m)$ von P als Typfunktion simuliert.
Bemerkung:

(i) *Nach* dieser Anwendung von P werden weitere Teilterme von t nicht mehr modulo Γ, sondern modulo $S\Gamma$ typisiert. Weitere Vorkommen der s_i stehen daher *nicht* mehr für Argumenttypen σ_i'.

(ii) Ob es zu σ einen (rec–freien) Term t gibt, so daß $\emptyset \triangleright t : \sigma$ eine Haupttypisierung ist, ist ein PSPACE–vollständiges Problem: es bedeutet zu zeigen, daß σ eine (intuitionistische) Tautologie ist (Statman[8]).

Die Terme, Anweisungen und Prozeduren einer Programmiersprache zum „Rechnen auf der Typebene“ definieren wir simultan wie folgt.

Definition 7.3.1

$$\begin{array}{lcl} t & := & x \quad | \quad P_i t_1 \cdots t_n \\ e & := & t_1 \hat{=} t_2 \quad | \quad t \to t_1 \hat{=} t_2 \quad | \quad \langle e_1, \ldots, e_n \rangle \\ P_n & := & \lambda x_1 \ldots x_n . \lambda y_1 \ldots y_m . \mathit{local}\ z_1 \ldots z_k . e, \end{array}$$

wobei folgende Abkürzungen verwendet werden:

$$\begin{array}{rcl} t_1 \hat{=} t_2 & := & \lambda z . \langle z t_1, z t_2 \rangle \\ t \to t_1 \hat{=} t_2 & := & t t_1 t_2 \\ \langle e_1, \ldots, e_n \rangle & := & \lambda z . z e_1 \cdots e_n \\ \mathit{local}\ z_1 \ldots z_k . e & := & \lambda z . K z (\lambda z_1 \ldots z_k . e) \end{array}$$

Proposition 7.3.2 Jede Typisierung von λ–Termen der Form $t_1 \hat{=} t_2, b \to t_1 \hat{=} t_2, \langle e_1, \ldots, e_n \rangle$ und $(\mathit{local}\ z_1 \ldots z_k . e)$ erfolgt nach folgenden herleitbaren Regeln:

$$\frac{\Gamma \triangleright t_1 : \sigma \qquad \Gamma \triangleright t_2 : \sigma}{\Gamma \triangleright t_1 \hat{=} t_2 : (\sigma \to \tau) \to (\tau \to \tau \to \varrho) \to \varrho} \qquad \frac{\Gamma \triangleright b : \sigma_1 \to \sigma_2 \to \tau, \quad \Gamma \triangleright t_i : \sigma_i}{\Gamma \triangleright b \to t_1 \hat{=} t_2 : \tau}$$

$$\frac{\Gamma \triangleright e_1 : \sigma_1 \quad \ldots \quad \Gamma \triangleright e_n : \sigma_n}{\Gamma \triangleright \langle e_1, \ldots, e_n \rangle : (\sigma_1 \to \ldots \to \sigma_n \to \tau) \to \tau} \qquad \frac{\Gamma \triangleright \lambda z_1 \ldots z_k . e : \sigma}{\Gamma \triangleright \mathit{local}\ z_1 \ldots z_k . e : \tau \to \tau}$$

Hierbei sind die Typen von $t_1 \hat{=} t_2$ usw. unwesentlich. Es kommt auf die „operationale“ Lesart der Regeln an:

Anweisung	*Bedeutung*
$t_1 \hat{=} t_2$	Unifiziere die Typen von t_1 und t_2.
$t \to t_1 \hat{=} t_2$	Unifiziere die Typen von t_1, t_2 mit den Argumenttypen von t
$\langle e_1, \ldots, e_n \rangle$	Führe die Anweisungen $e_1, \ldots, e_n$ aus.

7.4 Kodierung monotoner Boole'scher Algebra

Zur Interpretation der „bedingten Anweisungen" $b \to t_1 \hat{=} t_2$ brauchen wir eine geeignete Kodierung der Bedingung b als Boole'scher Wert auf der Typebene: Es genügt, die monotone Algebra $\mathcal{L} = (\{\texttt{true}, \texttt{false}\}, \wedge, \vee, \texttt{true}, \texttt{false})$ *ohne* Negation zu kodieren. Wir wählen folgende Programme:

Definition 7.4.1 (Kanellakis/Mitchell, Henglein)

$$\begin{aligned} \mathit{True} &:= \lambda y_1 y_2.\mathit{local}.\langle y_1 \hat{=} y_2 \rangle \\ \mathit{False} &:= \lambda y_1 y_2.\mathit{local}.\langle\,\rangle \\ \mathit{Or} &:= \lambda x_1 x_2.\lambda y.\mathit{local}.\langle y \hat{=} \mathit{False},\ x_1 \to y \hat{=} \mathit{True},\ x_2 \to y \hat{=} \mathit{True} \rangle \\ \mathit{And} &:= \lambda x_1 x_2.\lambda y.\mathit{local}\ z.\langle y \hat{=} \mathit{False},\ x_1 \to z \hat{=} \mathit{True},\ x_2 \to y \hat{=} z \rangle \end{aligned}$$

Proposition 7.4.2 Folgende Haupttypisierungen sind C–herleitbar:

$$\begin{aligned} &\emptyset \rhd \mathit{True} : \tau_{\mathit{True}} := \alpha \to \alpha \to \gamma \to \gamma \\ &\emptyset \rhd \mathit{False} : \tau_{\mathit{False}} := \alpha \to \beta \to \gamma \to \gamma \\ &\emptyset \rhd \mathit{Or} : (\tau_{\mathit{False}} \to \tau'_{\mathit{True}} \to \delta') \to (\tau_{\mathit{False}} \to \tau''_{\mathit{True}} \to \delta'') \to \tau_{\mathit{False}} \to \eta \to \eta \\ &\emptyset \rhd \mathit{And} : \eta \to \tau'_{\mathit{True}} \to \delta') \to (\tau_{\mathit{False}} \to \eta \to \delta'') \to \tau_{\mathit{False}} \to \eta' \to \eta' \end{aligned}$$

wobei $\delta', \delta'', \eta, \eta'$ Typvariable sind und τ'_{True} und τ''_{True} Kopien von τ_{True} mit frischen Typvariablen.

Folgerung 7.4.3 Als Funktionen auf Typen verhalten sich *True*, *False*, *Or* und *And* wie erwartet:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{True}() & = & (\alpha, \alpha) & \mathit{False}() & = & (\alpha, \beta) \\ \\ \mathit{Or}(\tau_{\mathit{True}}, \tau'_{\mathit{True}}) & = & \tau''_{\mathit{True}} & \mathit{And}(\tau_{\mathit{True}}, \tau'_{\mathit{True}}) & = & \tau''_{\mathit{True}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathit{Or}(\tau_{\mathit{False}}, \tau'_{\mathit{False}}) & = & \tau''_{\mathit{False}} & \mathit{And}(\tau_{\mathit{False}}, \tau'_{\mathit{False}}) & = & \tau''_{\mathit{False}} \end{array}$$

Beachte, daß die Wahl von *True*, *False* tatsächlich die angestrebte Bedeutung von $b \to y \hat{=} z$ als „bedingter Anweisung" erzeugt:

$$b \to y \hat{=} z \quad \equiv \quad byz$$

zu typisieren bedeutet, y und z zu typisieren und *falls* $b = \mathit{True}$, die Ergebnisse zu unifizieren. Bei der Übergabe von *True*, *False* an Prozeduren entstehen unerwünschte Seiteneffekte, falls die Argumente mehrfach verwendet werden:

Beispiel 7.4.4 Sei $P = \lambda x.\lambda y_1 y_2.\mathit{local}.\langle x \to y_1 \hat{=} \mathit{False}, x \to y_2 \hat{=} \mathit{True} \rangle$. Wir erwarten, daß der Ausdruck $(P\ \mathit{True})$ den Haupttyp

$$\tau_{\mathit{False}} \to \tau'_{\mathit{True}} \to \gamma \to \gamma$$

hat, d.h. $y_1 \hat{=} \mathit{False}$ und $y_2 \hat{=} \mathit{True}$ „ausführt". Da aber $\mathit{True} : \alpha \to \alpha \to \delta \to \delta$ Gleichheit der beiden Argumenttypen erzwingt und x in P überall denselben Typ hat (λ–Regel!), erhält man

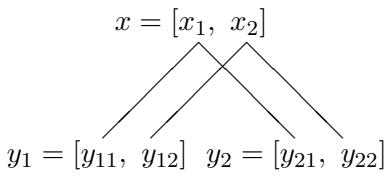
$$\emptyset \rhd (P\ \mathit{True}) : \tau_{\mathit{True}} \to \tau_{\mathit{True}} \to \gamma \to \gamma\ !$$

Die Kodierung ist also nicht korrekt, falls eine Prozedur ein boole'sches Argument mehrfach verwendet. Wir korrigieren das, indem wir in jeder Prozedur zunächst das Argument (bzw. seinen Typ) so oft kopieren, wie es verwendet wird. Dazu wird eine boole'sche Verzweigung verwendet.

Definition 7.4.5

$$\begin{array}{rcl} [x_1, \ldots, x_n] & := & \lambda y_1 \ldots y_n.\mathit{local}.\langle y_1 \hat{=} x_1, \ldots, y_n \hat{=} x_n \rangle \\ \mathit{fanout}\ x & := & \lambda y_1 y_2.\mathit{local}\ x_1\, x_2\, y_{11}\, y_{12}\, y_{21}\, y_{22}. \\ & & \langle\ [y_{11}, y_{12}] \hat{=} y_1,\ y_1 \hat{=} \mathit{False},\ [y_{21}, y_{22}] \hat{=} y_2,\ y_2 \hat{=} \mathit{False}, \\ & & \quad [x_1, x_2] \hat{=} x,\ [y_{11}, y_{21}] \hat{=} x_1,\ [y_{12}, y_{22}] \hat{=} x_2\ \rangle \end{array}$$

Anschaulich ist *fanout* ein Schaltkreis der Form



Proposition 7.4.6 Jede Typisierung von $[e_1, \ldots, e_n]$ und $(\mathit{fanout}\ e)$ erfolgt gemäß den folgenden herleitbaren Regeln:

$$\frac{\Gamma \rhd e_1 : \sigma_1,\ \ldots,\ \Gamma \rhd e_n : \sigma_n}{\Gamma \rhd [e_1, \ldots, e_n] : \sigma_1 \to \ldots \to \sigma_n \to \varrho \to \varrho}$$

$$\frac{\Gamma \rhd e : (\sigma_1 \to \sigma_2 \to \sigma_3) \to (\tau_1 \to \tau_2 \to \tau_3) \to \varrho}{\Gamma \rhd (\mathit{fanout}\ e) : (\sigma_1 \to \tau_1 \to \varrho_1 \to \varrho_1) \to (\sigma_2 \to \tau_2 \to \varrho_2 \to \varrho_2) \to \varrho_3 \to \varrho_3}$$

Folgerung: Folgende Haupttypisierungen sind herleitbar:

$$\begin{array}{l} \Gamma \rhd \mathit{fanout}\ \mathit{True} : (\alpha \to \alpha \to \gamma_1 \to \gamma_1) \to (\beta \to \beta \to \gamma_2 \to \gamma_2) \to \gamma \to \gamma \\ \Gamma \rhd \mathit{fanout}\ \mathit{False} : (\alpha \to \beta \to \gamma_1 \to \gamma_1) \to (\delta \to \eta \to \gamma_2 \to \gamma_2) \to \gamma \to \gamma \end{array}$$

bzw.

$$\begin{array}{l} \Gamma \rhd \mathit{fanout}\ \mathit{True} : \tau_{\mathit{True}} \to \tau'_{\mathit{True}} \to \gamma \to \gamma \\ \Gamma \rhd \mathit{fanout}\ \mathit{False} : \tau_{\mathit{False}} \to \tau'_{\mathit{False}} \to \gamma \to \gamma. \end{array}$$

Mit anderen Worten, *fanout* erzeugt aus den boole'schen Eingabetypen zwei voneinander unabhängige Kopien.

Beispiel 7.4.7 (Fortsetzung von 7.4.4) Ersetzen wir

$$P = \lambda x.\lambda y_1 y_2.local.\langle x \to y_1 \hat{=} False,\ x \to y_2 \hat{=} True\rangle$$

durch

$$P' = \lambda x.\lambda y_1 y_2.local\ z_1, z_2.\langle [z_1, z_2] \hat{=} (fanout\ x),\ z_1 \to y_1 \hat{=} False,\ z_2 \to y_2 \hat{=} True\rangle,$$

so erhalten wir die gewünschten Haupttypisierungen

$$\begin{array}{l} \emptyset \rhd (P'\ True) : \tau_{False} \to \tau'_{True} \to \eta \to \eta, \\ \emptyset \rhd (P'\ False) : \alpha \to \beta \to \gamma \to \gamma, \end{array}$$

d.h. der unerwünschte Seiteneffekt von P ist behoben.

7.5 Kodierung endlicher Mengen

Nach den Programmen müssen wir jetzt noch die Kodierung der Bandsymbole, Maschinenzustände usw. festlegen, damit wir die Ausführung der Programme auf einer Maschine durch die Typsynthese simulieren können.

Wir repräsentieren Mengen $D = \{d_0, \ldots, d_m\}$ der Mächtigkeit m durch Folgen boole'scher Werte,

$$\begin{array}{rcl} d_0 & := & [True, False, \ldots, False] \\ d_1 & := & [False, True, \ldots, False] \\ & \vdots & \\ d_m & := & [False, False, \ldots, True], \end{array}$$

wobei in der Kodierung von d_i nur das $i+1$-te Folgenglied $True$ ist. Fallunterscheidung über D kann damit durch folgendes Hilfsprogramm dargestellt werden:

$$\begin{array}{l} select_D(x, [x_0, \ldots, x_m]) := \\ \quad \lambda y.local\ z_0 \ldots z_m.\langle [z_0, \ldots, z_m] \hat{=} x,\ z_0 \to y \hat{=} x_0,\ \ldots,\ z_m \to y \hat{=} x_m\rangle \end{array}$$

Beachte, daß die Typisierung dieses Ausdrucks einen Seiteneffekt auf x ausübt, d.h. den Eingangstyp von x verändert. Um die (Typen der) Elemente von D mehrfach verwenden zu können, brauchen wir auch eine Möglichkeit, davon frische Kopien zu erzeugen:

$$\begin{array}{rcl} copy & := & \lambda x.\lambda y_1 y_2.local\ z_0 z'_0 z''_0 \ldots z_m z'_m z''_m. \\ & & \langle\ [z_0, \ldots, z_m] \hat{=} x, \\ & & \quad [z'_0, z''_0] \hat{=} (fanout\ z_0),\ \ldots,\ [z'_m, z''_m] \hat{=} (fanout\ z_m), \\ & & \quad y_1 \hat{=} [z'_0, \ldots, z'_m],\ y_2 \hat{=} [z''_0, \ldots, z''_m]\ \rangle \end{array}$$

Beachte wieder, daß auf x ein Seiteneffekt ausgeübt wird. Von den 2 frischen Kopien können wir aber eine für weiteres Kopieren verwenden, und damit e–faches kopieren definieren

$$\begin{array}{rcl} copy - e & := & \lambda x.\lambda y_1 \ldots y_e.local\ z_0 z'_0 z''_0 \ldots z'_m \ldots z''_m x_1 \ldots x_e. \\ & & \langle [x_1, y_1] \hat{=} (copy\ x),\ [x_2, y_2] \hat{=} (copy\ x_1),\ \ldots,\ [x_e, y_e] \hat{=} (copy\ x_{e-1})\rangle \end{array}$$

7.6 Kodierung von Turingmaschinen

Eine Turingmaschine $M = (Q, \Gamma, \delta, F, q_0)$ mit Zustandsmenge $Q = \{q_0, \ldots, q_k\}$, Bandsymbolen $\Gamma = \{B_0, \ldots, B_l\}$, Bewegungsrichtungen L, R, Transitionsfunktion $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, Startzustand $q_0 \in Q$, Endzustandsmenge $F \subseteq Q$ kodieren wir nun „in den Typen“ geeigneter λ–Terme.

Kodierung von $Q, \Gamma, \{L, R\}$: Dies erfolgt gemäß der Kodierung endlicher Mengen, 7.5.

Kodierung von δ:

$$
\begin{array}{lll}
P_\delta & := & \lambda q, b.\lambda q' b' d.\mathit{local}\ t_0 t_1 1_2 b_0 \ldots b_k. \\
 & & \langle\ [b_0, \ldots, b_k] \hat{=} (\mathit{copy}_\Gamma - (k+1)\ b), \\
 & & \quad t_0 \hat{=} (\mathit{select}_Q\ q \quad [\mathit{select}_\Gamma\ b_0\ [\delta(q_0, B_0), \ldots, \delta(q_0, B_e)], \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
 & & \qquad\qquad\qquad\qquad\quad \mathit{select}_\Gamma\ b_k\ [\delta(q_k, B_0), \ldots, \delta(q_k, B_e)]\ ]), \\
 & & \quad [t_1] \hat{=} t_0, \\
 & & \quad [t_2] \hat{=} t_1, \\
 & & \quad [q', b', d] \hat{=} t_2\ \rangle
\end{array}
$$

Kodierung einer Konfiguration $K \in \Gamma^* \times Q \times \Gamma^*$:

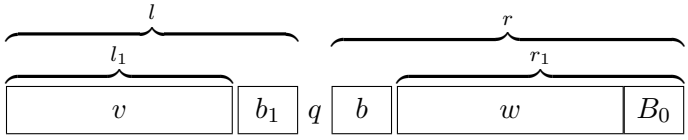
Wir können annehmen, daß ein ausgezeichnetes Bandsymbol, etwa B_0, stets und nur am Anfang und am Ende der Bandinschrift vorkommt. Außerdem soll M nicht über den Wortanfang nach links laufen (bzw. nur das rechte Wortende verschieben können). Ist $K = b_1 b_2 \ldots b_n q b_{n+1} \ldots b_m$, so kodieren wir K durch den Term

$$
\begin{array}{lllllll}
\overline{K} & = & [\overline{K_l}, q, \overline{K_r}], & \text{mit} & \overline{K_l} & = & [b_n, [b_{n-1}, [\ldots [b_1, \$] \ldots]]], \\
 & & & & \overline{K_r} & = & [b_{n+1}, [b_{n+2}, [\ldots [b_m, \$] \ldots]]], \\
 & & & & \$ & = & [B_0, \lambda z.z] \qquad (\lambda z.z = \langle\,\rangle)
\end{array}
$$

Eine *Anfangskonfiguration* mit Eingabe $b \ldots b_m$ hat also die Form

$$\overline{K_0} = [\$, q_0, [b_1, [\ldots, [b_m, \$] \ldots]]].$$

Kodierung eines Rechenschritts: Eine Konfiguration

$K =$ 

können wir nach ihrer Form zerlegen und mit Fallunterscheidungen analysieren. Die Nachfolgekonfiguration wird dann mit P_δ aus den Bestandteilen zusammengesetzt:

$$
\begin{array}{rcl}
P_{move} & = & \lambda K.\lambda K'.local\ldots \\
 & & \langle\ [l,q,r]\hat{=}K, \\
 & & \ [b_1,r_1]\hat{=}r, \\
 & & \ [b',r']\hat{=}(select\ b_1\ [r,\ldots,[B,\$],\ldots r]), \qquad ([B,\$]\text{: Test auf Bandende}) \\
 & & \ [b_l,l_1]\hat{=}l, \\
 & & \ [q',\tilde{b},dir]\hat{=}(P_\delta\ q\ b'), \\
 & & \ K'\hat{=}(select\ dir\ [\ \underbrace{[l_1,q',[b_l,[\tilde{b},r']]]}_{\text{Linksbewegung}},\quad \underbrace{[[\tilde{b},l],q',r']}_{\text{Rechtsbewegung}}\quad])\ \rangle
\end{array}
$$

Eine Rechtsbewegung wird in δ durch $vb_lqbw \to vb_l\tilde{b}q'w$ beschrieben; entsprechend für Linksbewegungen. Wir können annehmen, daß M bei jeder Eingabe der Länge n mindestens 2^n Schritte rechnet.

Kodierung von $2^{c\cdot n}$ Rechenschritten durch einen *ML*–Term der Länge n

Wir verwenden jetzt die kompakte Darstellung von Funktionskomposition aus 7.2.3, um durch einen kurzen *ML*–Term viele Hintereinanderausführungen von P_{move} auf den Typen zu simulieren:

$$
\begin{array}{rcl}
sim_{K_0} = & let & move_1 = P_{move} \circ P_{move} \\
 & in & \ let \quad move_2 = move_1 \circ move_1 \\
 & & \ \ in \\
 & & \qquad \ddots \\
 & & \qquad\quad let \quad move_n = move_{n-1} \circ move_{n-1} \\
 & & \qquad\ in \quad \lambda lqr.\lambda y.local.\langle\ [l,q,r]\hat{=}(move_n\ K_0), \\
 & & \qquad\qquad\qquad\qquad\ y\hat{=}(select\ q\ [False,\ldots,True,\ldots]), \\
 & & \qquad\qquad\qquad\qquad\ y \to \lambda x.x\hat{=}\lambda xy.x\ \rangle,
\end{array}
$$

wobei K_0 die Anfangskonfiguration zu gegebener Eingabe ist, und in $[False,\ldots,True,\ldots]$ steht *True* an den Positionen i mit $q_i \in F$, *False* an den anderen. Sei

$$
\begin{array}{rl}
sim'_{K_0} = \lambda l\,q\,r.\lambda y.local.\langle & [l,q,r]\hat{=}(\overbrace{P_{move}(P_{move}(\ldots(P_{move}}^{2^n\text{–mal}}\ K_0)\ldots)))), \\
 & y\hat{=}(select\ q\ [False,\ldots,True,\ldots]), \\
 & y \to \lambda x.x\hat{=}\lambda xy.x\ \rangle.
\end{array}
$$

das Ergebnis der `let`-Reduktion von sim_{K_0}.

Lemma 7.6.1 M akzeptiert die in K_0 kodierte Eingabe w in 2^n Schritten $\Leftrightarrow$ sim_{K_0} ist nicht *ML*–typisierbar.

Beweis: M befindet sich nach 2^n Schritten in einem Endzustand $\Leftrightarrow$ q und damit y in sim' müssen bei einer Wohltypisierung den Typ τ_{True} erhalten $\Leftrightarrow$ $y \to \lambda x.x\hat{=}\lambda xy.x$ ist untypisierbar wegen $y : \tau_{True}$, also auch sim'. □

Beachte, daß zur Typisierung von *sim* die polymorphe *let*–Regel erforderlich ist, d.h. daß die beiden Vorkommen der $move_i$ jeweils *verschiedene* Typen brauchen: an *sim'* sieht man, daß die Teiltterme $(P_{move}(\ldots(P_{move}\ K_0)\ldots))$ in ihren Typen i.a. verschiedene Konfigurationen von M kodieren, also P_{move} jeweils anderen Typ annimmt. Damit haben wir gezeigt:

Satz 7.6.2 (Mairson[6],Kfoury e.a.[5]) Das Typisierbarkeitsproblem für *ML*-Terme ist DEXPTIME schwer.

Daß man die Frage, ob ein λ–Term *ML*–typisierbar ist, mit einer deterministischen Turingmaschine in exponentieller Zeit entscheiden kann, zeigt

Satz 7.6.3 (P. Kanellakis/J.C. Mitchell[4]) *ML*–Typisierbarkeit liegt in DEXPTIME.

Für den Beweis wird auf die angegebene Literatur verwiesen.

Es bleibt zu erklären, wieso die hohe theoretische Komplexität der *ML*-Typsynthese sich wie eingangs gesagt in der Programmierpraxis nicht bemerkbar macht. Ein plausibler Grund ist, daß von `let`-Abkürzungen nicht sehr tief geschachtelt auftreten, so daß die Vervielfachung der Größe von Haupttypen kaum auftreten dürfte. Außer bei sehr kurzen Programmen, die man zur Lehrzwecken benutzt, ist der Haupttyp eines *ML*-Programms normalerweise nicht größer als das Programm. McAllester[1] hat gezeigt, daß die Typisierbarkeit eines ML-Programms e in fast-linearer Zeit (d.h. in $O(bn \cdot \alpha(bn))$ Schritten, wobei $n = |e|$, b eine Konstante und α die Inverse der Ackermannfunktion ist) entschieden werden kann, wenn folgende Voraussetzung erfüllt ist: ist e' durch Ausführen aller `let`-Redexe aus e entstanden und U das Unfikationsproblem für Typausdrucke, das man zur Synthese des C-Typs von e' lösen muß, so hat jeder in der Lösung von U auftretende Typ eine durch b beschränkte Größe.

Literatur

[1] David McAllester. Inferring recursive data types. 1996. unpublished.

[2] F. Henglein. A lower bound for full polymorphic type inference: Girard-Reynolds typability is DEXPTIME-hard. Technical Report RUU-CS-90-14, Utrecht University, April 1990.

[3] F. Henglein. A simplified proof of DEXPTIME-completeness of ML typing. Manuscript, March 1990.

[4] Paris B. Kanellakis and John Mitchell. Polymorphic unification and ML-typing. In *ACM Symposium on Principles of Programming Languages*, pages 5–15, 1989.

[5] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML-typability. *Journal of the ACM*, 199?

[6] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Prooceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 382–401. ACM, January 1990.

[7] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[8] R. Statman. Intuitionistic propositional logic is polynomial space complete. *Theoretical Computer Science*, 9:67–72, 1979.