

ML-Polymorphismus

Autor: Marc Hoffmann

Betreuer: Prof. Tobias Nipkow

im Rahmen des Proseminars „Programmiersprachen“
an der Technischen Universität München

29. November 2006

Inhaltsverzeichnis

1	Einleitung	4
1.1	Motivation	4
1.2	Lambda-Kalkül und Ocaml	4
1.2.1	Variablen & Applikation:	4
1.2.2	Abstraktion:	5
2	Explizite Typbestimmung	6
2.1	Typen im λ -Kalkül	6
2.1.1	Gebundene Variablen	6
2.1.2	Die Umgebung Γ	7
2.1.3	Notation	7
2.2	Typbestimmung durch Ableitungsbäume	7
2.2.1	Typinferenz	8
2.2.1.1	Gebundene und freie Variablen:	8
2.2.1.2	Ableitungsregeln [inference rules]	8
2.2.1.3	Äquivalenz der Ausdrücke	9
2.2.2	Bool als Typ	9
2.2.3	Beispielhafte Typherleitungen	10
2.3	Eigenschaften des einfach typisierten λ -Kalküls	10
2.3.1	Eindeutigkeit des Typs (bei gegebener Umgebung Γ)	10
2.3.2	Typerhalt bei Evaluierung	11
3	Implizite Typbestimmung	13
3.1	Substitution von Typvariablen	13
3.1.1	Substitution σ als Beschreibung oder Anwendung	13
3.1.2	Formale Beschreibung von σ	13
3.1.3	Typerhalt bei σ -Substitution	14
3.2	Typ als Lösung von (Γ, t)	15
3.2.1	Formale Lösung von (Γ, t)	15
3.2.2	Beispiel	15
3.3	Typinferenz mit Bedingungen [constraint based typing]	15
3.3.1	Bestimmung von Bedingungen \mathcal{C}	16
3.3.1.1	Definition und Erfüllung von Bedingungen:	16
3.3.1.2	Erzeugung von Bedingungen:	16
3.3.1.3	Lösung von (Γ, t)	16

3.3.2	Ableitungsregeln mit Bedingungen	16
3.3.3	Beispiel	18
3.3.4	Äquivalenz der Lösungen für (Γ, t) und (Γ, t, S, C)	19
3.3.4.1	Lemma „ \Leftarrow “ [soundness of constraint typing]	19
3.3.4.2	Lemma „ \Rightarrow “ [completeness of constraint typing]	19
3.4	Unifikation [unification]	20
3.4.1	Allgemeinere und allgemeinste Substitution	20
3.4.2	Unifikations-Algorithmus	21
3.4.3	Allgemeinster Typ eines Terms	22
3.4.4	Beispiel	22
3.5	Zusammenfassung der Typrekonstruktion	24
3.6	Let-Polymorphismus	24
3.6.1	Anforderungen an ein <code>let</code> -Konstrukt	25
3.6.2	Schrittweise Herleitung der <code>let</code> -Ableitung (CT-LetPoly)	26
3.6.3	Hinzunahme der impliziten Annotation	26
3.6.4	Verfeinerung von (T-LETPOLY)	27
4	Zusammenfassung	28

1 Einleitung

1.1 Motivation

Grundlage jeder funktionalen Programmiersprache ist der λ -Kalkül - ein formales System, welches ab den 1920er Jahren von Alonzo Church entwickelt wurde. Objekt unserer Betrachtungen soll der einfach getypte λ -Kalkül (*simply typed lambda-calculus*, Church 1940, Curry 1958) sein.

Typen dienen hauptsächlich dazu, ein voraussehbares Verhalten zu garantieren, Programmierfehler sowie Inkonsistenzen zu verhindern und Spezifikationen (beispielsweise für Schnittstellen) mit Typangaben auszudrücken. In der Welt der funktionalen Programmiersprachen und insbesondere in der ML-Sprachfamilie stellt der „ML-Style“-Polymorphismus die Grundlage für die konkreten Typmechanismen dar. Wegen der Verwendung von Typvariablen sind die Mechanismen dem parametrischen Polymorphismus zuzuschreiben.

Der ML-Polymorphismus basiert auf dem λ -Kalkül und erlaubt die Prüfung auf Typkorrektheit von geschlossenen Ausdrücken (auch Kombinatoren genannt). Besonders interessant ist die implizite Typinferenz, mit welcher der (allemeinste) Typ eines Ausdrucks rekonstruiert werden kann, ohne dass Typangaben vonnöten sind.

Ziel dieses Vortrages ist es, eine Einführung in die Typmechanismen des λ -Kalküls zu geben, die angesprochene implizite Typermittlung und die dabei verwendeten Algorithmen zu ergründen, um schließlich einen Blick auf `let`-Konstrukte zu werfen.

1.2 Lambda-Kalkül und Ocaml

Zur Veranschaulichung der Theorie können Terme des λ -Kalküls direkt in OCaml ausgedrückt werden. Dies sollte nicht verwundern - schließlich ist der λ -Kalkül die Grundlage für funktionale Sprachen. OCaml prüft die Terme auf Korrektheit, bevor das eigentliche Programm ausgeführt werden kann. Dazu werden die jeweiligen Typen der angegebenen Ausdrücke rekonstruiert. Mit welchen Mechanismen dies geschieht, werden wir in Abschnitt 3 kennenlernen.

1.2.1 Variablen & Applikation:

Variablen und Applikationen werden wie im λ -Kalkül ausgedrückt. „;“ bezeichnet dabei in OCaml bekanntermaßen das Ende eines Ausdrucks.

- Variable: `x x;;`
- Applikation: `f x f x;;`

1.2.2 Abstraktion:

1. `λx. x fun x -> x;;` - die Identität
2. `λx. λy. x y fun x -> fun y -> x y;;` - Funktion höherer Ordnung: es wird eine Funktion `x` und ein Ausdruck `y` übergeben; schließlich wird `x` auf `y` angewendet (Applikation).
3. `λx. x x : fun x -> x x;;` - Funktion, welche ihr Argument auf sich selbst anwendet.

Die Typüberprüfung durch OCaml wertet folgendermaßen aus:

1. `'a -> 'a = <fun>` - Die Funktion nimmt einen variablen Typ `A` als Argument an und gibt einen Term mit *selbigem* Typ `A` zurück.
2. `('a -> 'b) -> 'a -> 'b = <fun>` - entspricht $(A \rightarrow B) \rightarrow (A \rightarrow B)$ laut *Rechtsassoziativität*. Die äußere Funktion (höherer Ordnung) nimmt eine Funktion $(A \rightarrow B)$ als Parameter an (hier `x`), wonach die innere Funktion einen Term vom Typ `A` (hier `y`) in einen Term mit Typ `B` umwandelt; der Rückgabewert der Funktion ist also `B`.
3. `This expression has type 'a -> 'b but is here used with type 'a` - Der Typ des Ausdrucks kann nicht korrekt ermittelt werden.

Im reinen λ -Kalkül waren wir nur durch die Syntax des λ -Kalküls beschränkt - aber nicht alle syntaktisch korrekten Terme sind typisierbar! Typisierbare Terme garantieren allerdings, dass die Auswertung der Ausdrücke in keinen festgefahrenen Zustand [stuck state] gerät - typkorrekte Ausdrücke werten stets aus und sind folglich *typsicher* (vgl. [Pie02]).

Weiterhin ist zu beobachten, dass die Ausdrücke keinerlei Typangaben (*Typannotationen*) beinhalten. Im folgenden werden wir zunächst die Typherleitung mit Annotationen betrachten, um uns später dem Rekonstruktions- und Unifikationsalgorithmus zu widmen, bei welchem Typannotationen entbehrlich sind. Beim let-Polymorphismus ist das Aussparen der Typannotationen in manchen Fällen schließlich nötig.

2 Explizite Typbestimmung

Um die Mechanismen der Typinferenz zu verinnerlichen, möchten wir zunächst die „explizite Typbestimmung“ betrachten. Sie wird *explizit* genannt, weil für alle gebundenen Variablen ein fester Typ angegeben (annotiert) wird. Insbesondere werden nur Terme betrachtet, in denen alle Variablen gebunden sind, d.h. *geschlossene Terme*. Die jeweiligen Typen der Ausdrücke werden (falls möglich) durch *Ableitungsbäume* bestimmt, welche wir nach der Einführung einiger Formalien kennenlernen werden.

Wir gehen bei der expliziten Typbestimmung von einem monomorphen Typsystem aus, um später - darauf aufbauend - das polymorphe Typsystem vorzustellen.

2.1 Typen im λ -Kalkül

2.1.1 Gebundene Variablen

Bei einer Abstraktion der Form $\lambda x . t$ wird zukünftig für die Variable x ein Typ T gewählt. Im Term t heißt die Variable x *gebunden*. Für *alle* Vorkommnisse der Variable x in t nehmen wir dann diesen Typ T an. Können wir schließlich unter dieser Annahme zeigen, dass der Term t den Typ T' annimmt, so hat die Abstraktion insgesamt den Typ $T \rightarrow T'$. Wir werden dies später (2.2.1.2) formaler ausdrücken.

Anmerkungen:

- Der Typ T' hängt vom Typ T ab.
- Der Term t ist *nicht typkorrekt*, wenn kein Typ T' ableitbar ist.
- Variablenbezeichner werden höchstens einmal gebunden. Möchten wir einen Ausdruck aus zwei Termen aufbauen, bei denen der Schnitt der Menge der Variablenbezeichner nicht-leer ist, so nennen wir die betroffenen Variablenbezeichner in einem der Terme um (mit einem noch ungebrauchten Bezeichner).

Beispiel: Es sei $f = \lambda x . \text{iszero } x$ mit $x : \text{Nat}$. Dann hat f den Typ Bool .

2.1.2 Die Umgebung Γ

Um Annahmen über die gebundenen Variablen festzuhalten, bedienen wir uns einer *Umgebung*. Dadurch können die verschiedenen Variable-Typ Tupel $(x : T)$ bei komplexeren Ausdrücken (z.B. mehrere verschachtelte Abstraktionen) sinnvoll verwaltet werden.

Umgebung: Formal ist eine *Umgebung* Γ eine Sequenz (Menge) von Tupeln $x : T$, wobei x eine Variable und T der zugehörige Typ ist. Neue Einträge werden rechts eingefügt und werden mit einem Komma von den bisherigen getrennt, also z.B. $\Gamma = x_1 : T_1, x_2 : T_2$. Die leere Umgebung \emptyset wird oft nicht ausgeschrieben, sondern ausgelassen.

2.1.3 Notation

Variablen $\Gamma \vdash x : T$

„die Variable x hat unter den Annahmen Γ den Typ T “

Abstraktion $\Gamma \vdash \lambda x : T_1 . t : T_1 \rightarrow T_2$

„die Funktion $\lambda x . t$ hat unter den Annahmen Γ den Typ $T_1 \rightarrow T_2$ “

Applikation $\Gamma \vdash t_1 t_2 : T$

„die Applikation $t_1 t_2$ hat unter den Annahmen Γ den Typ T “.

Betrachten wir den Gesamtterm, so schreiben wir

$$\vdash t : T$$

und meinen „der *geschlossene* Term t hat Typ T bei leerer Umgebung“. Dies ist gerade, was wir bestimmen möchten (und nennen den Term t auch *Gesamtterm* / *Gesamtausdruck*).

2.2 Typbestimmung durch Ableitungsbäume

Wir bestimmen im Folgenden *Ableitungsregeln*, welche auf Terme der Form $\Gamma \vdash t$ angewandt werden (bei noch unbekanntem Typ). Diese *Inferenzregeln* erzeugen einen oder mehrere neue Terme der Form $\Gamma' \vdash t'$. Wiederholen wir diese Vorgehensweise, bis wir auf Terme stoßen, die nicht weiter ableitbar sind, so erhalten wir einen *Ableitungsbaum*. In diesen Blättern finden wir die Variablen, aus welchen die Terme zusammengesetzt sind.

Geht alles gut, so steht in den Blättern jeweils ein Ausdruck, dessen Typ wir bestimmen können. Nun können die Typen der jeweiligen Teilausdrücke „rückwärts“ - gemäß den Ableitungsregeln - bestimmt werden, bis wir den Typ des Gesamtausdrucks abgeleitet haben. Falls an mindestens einer Stelle ein Schritt nicht möglich ist, so ist der Term insgesamt nicht typisierbar - und somit nicht typkorrekt.

2.2.1 Typinferenz

2.2.1.1 Gebundene und freie Variablen:

Wir gehen von einem Gesamtterm aus, in welchem Variablenbezeichner höchstens einmal gebunden sind (siehe auch 2.1.1). Hiermit wird die Darstellung der Ableitungsregeln vereinfacht, da die Unterscheidung zwischen freien und gebundenen Variablen entfällt.

Beispielsweise ist der diese Bedingung nicht erfüllende Term

$$\lambda x : T_1 . \lambda y : T_2 . (\lambda x : T_3 . x y) x$$

äquivalent in folgenden „unproblematischen“ Ausdruck umwandelbar:

$$\lambda x : T_1 . \lambda y : T_2 . (\lambda z : T_3 . z y) x$$

Stellen wir diese Bedingung nicht, so müsste unsere Umgebung zwei Einträge für x sinnvoll verwalten. Dies ist möglich (z.B. mit Verschattung), aber hier nicht erwünscht.

2.2.1.2 Ableitungsregeln [inference rules]

Die erste Ableitungsregel haben wir im Wesentlichen bereits bei 2.1.1 kennengelernt:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

1. $\Gamma \vdash \lambda x : T_1 . t_2$ (unterhalb) ist eine Abstraktion mit Annahmen Γ , wobei für die Variable x der Typ T_1 im Term t_2 angenommen wird.
 $\Gamma, x : T_1 \vdash t_2$ (oberhalb) spiegelt genau diesen Sachverhalt wieder: alle Vorkommnisse von x in t_2 sollen den Typ T_1 haben.
2. Können wir nun zeigen, dass unter den Annahmen ($\Gamma' = \Gamma, x : T_1$) der Term t_2 den Typ T_2 annimmt,
3. so hat der Ausdruck $\Gamma \vdash \lambda x : T_1 . t_2$ den Typ $T_1 \rightarrow T_2$.

Der Ausdruck oberhalb des Trennstriches wird auch *Prämisse* genannt. Ist diese erfüllt (und ist insbesondere der Typ bestimmt), so folgt der Typ des unteren Ausdruckles.

Diese Ableitungsregel ist für alle Abstraktionen der Form $\Gamma \vdash \lambda x : T_1 . t_2$ anwendbar. Klar ist, dass die Bestimmung des Typs von t_2 weitere Ableitungsschritte benötigt. Für Variablen gestaltet sich dies recht einfach:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

Der Typ einer Variablen ist gerade der, welcher in der momentanen Umgebung vorkommt.

Dass eine Variable tatsächlich in der jeweiligen Umgebung vorkommt, ist gegeben: der Gesamtterm (mit leerer Umgebung) ist geschlossen, d.h. alle Variablen müssen durch Abstraktionen gebunden sein und werden folglich durch (T-ABS) in Γ vermerkt (laut 2.2.1.1 sogar genau einmal).

Ist eine Variable nicht durch eine Abstraktion gebunden, so ist ihr kein Typ zugeordnet; folglich ist der Gesamtterm nicht typisierbar¹.

Bemerkung: Die Ableitung (T-VAR) erzeugt ein Blatt im Ableitungsbaum, ab welchem die Typbestimmung zurück zum Ursprungsterm (rückwärtig) laufen kann.

Schließlich benötigen wir noch eine Ableitungsregel für die Applikation:

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

1. Gegeben: $\Gamma \vdash t_1 t_2$ (Applikation).
2. Können wir unter den Voraussetzungen Γ zeigen, dass t_1 den Typ $T_{11} \rightarrow T_{12}$ und t_2 den Typ T_{11} hat,
3. so ist das Resultat der Applikation $t_1 t_2$ gerade T_{12} .

Auch hier sind weitere Ableitungsschritte für die Bestimmung der Typen von t_1 und t_2 (in der Umgebung Γ) vonnöten.

2.2.1.3 Äquivalenz der Ausdrücke

Die Ableitungsregeln stellen eine Äquivalenz dar: die Ausdrücke oberhalb des Trennstriches folgen aus dem Ausdruck unterhalb, und umgekehrt². Beispielsweise gilt $\Gamma \vdash x : T \Leftrightarrow x : T \in \Gamma$.

2.2.2 Bool als Typ

Um den Typinferenzmechanismus nutzen zu können, benötigen wir noch konkrete Typen. Im folgenden stellen wir den Typ *Bool* mitsamt der zugehörigen Ableitungsregeln vor, um schließlich eine Typinferenz vorführen zu können. Weitere Typen wie z.B. *Nat* werden der Kürze wegen nicht vorgestellt.

$$\frac{}{\Gamma \vdash true : Bool} \quad (\text{T-TRUE})$$

$$\frac{}{\Gamma \vdash false : Bool} \quad (\text{T-FALSE})$$

¹OCaml lehnt einen solchen Fall mit „unbound variable“ ab.

²Siehe auch [Pie02, Inversion of the typing relation] S. 104.

In jeder Umgebung sind *true* und *false* vom Typ *Bool*. Diese beiden Ableitungsregeln erzeugen ebenfalls Blätter im Ableitungsbaum (insbesondere sind sie Prämissenfrei, und werden auch manchmal ohne Trennstrich geschrieben).

$$\frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

1. Gegeben: $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3$.
2. Können wir unter den Annahmen Γ zeigen, dass t_1 den Typ *Bool*, t_2 sowie t_3 den *gleichen* Typ T haben
3. dann ist der Typ der unteren Zeile ebenfalls T .

Hier wird vorausgesetzt, dass der Typ von t_2 mit dem Typ von t_3 übereinstimmt. Dies ist insofern sinnvoll, als dass die Auswertung des Ausdrucks t_1 nicht nötig ist, um den Typ des unteren Ausdrucks zu bestimmen (ob t_1 zu *true* oder *false* auswertet, ändert am Typ T nichts).

2.2.3 Beispielhafte Typherleitungen

- Was ist der Typ des Ausdrucks $(\lambda x : Bool . x) \text{ true}$?

$$\frac{\frac{\frac{x : Bool \in x : Bool}{x : Bool \vdash x : Bool} \text{ T-VAR}}{\vdash \lambda x : Bool . x : Bool \rightarrow Bool} \text{ T-ABS} \quad \frac{}{\vdash \text{true} : Bool} \text{ T-TRUE}}{\vdash (\lambda x : Bool . x) \text{ true} : Bool} \text{ T-ABS}$$

- Übung: Zeigen Sie, dass gilt:
 $f : Bool \rightarrow Bool \vdash \lambda x : Bool . f (\text{if } x \text{ then } \text{false} \text{ else } x) : Bool \rightarrow Bool$

2.3 Eigenschaften des einfach typisierten λ -Kalküls

2.3.1 Eindeutigkeit des Typs (bei gegebener Umgebung Γ)

Behauptung: Bei gegebener Umgebung Γ hat ein Term t (dessen freie Variablen alle in Γ vorkommen) höchstens einen Typ; falls ein Term also typisierbar ist, so ist der Typ eindeutig. Weiterhin ist der zugehörige Ableitungsbaum, welcher auf den Ableitungsregeln basiert, eindeutig.

Beweis: Wir nehmen an, ein Typ hätte bei gleicher Umgebung Γ sowohl den Typ S als auch T , d.h.

$$\Gamma \vdash t : T$$

$$\Gamma \vdash t : S$$

und zeigen durch Induktion über den Ableitungsbaum, dass $S = T$ gilt.

Die Schlüsse beruhen auf der Äquivalenz der oberen und unteren Ausdrücke in den Ableitungsregeln (siehe 2.2.1.3).

- Fall (T-VAR): Es sei $t = x$ mit $x : T \in \Gamma$.
Wir nehmen nun $\Gamma \vdash x : S$ an, laut (T-VAR) ist dann $x : S \in \Gamma$. Da jede Variable nur einmal gebunden wird, folgt $S = T$.
(T-VAR) erzeugt Blätter in Ableitungsbäumen, und stellt somit einen Induktionsanfang dar. Übrigens erfüllen (T-TRUE) und (T-FALSE) die Bedingung trivialerweise. ✓
- Fall (T-ABS): Gegeben seien der Term $t = \lambda x : R . t_1$ sowie $\Gamma, x : R \vdash t_1 : T_2$.
Aus (T-ABS) folgt, dass falls $\Gamma \vdash \lambda x : R . t_1 : T$ gilt, so ist $T = R \rightarrow T_2$ für ein T_2 mit $\Gamma, x : R \vdash t_2 : T_2$. Wir nehmen nun wieder an, dass der Term t ebenfalls den Typ S hat. Daraus folgt mit (T-ABS), dass $S = R \rightarrow S_2$ für ein S_2 mit $\Gamma, x : R \vdash t_2 : S_2$. Laut Induktionsannahme gilt $S_2 = T_2$, womit $S = R \rightarrow S_2 = R \rightarrow T_2 = T$ folgt. ✓
- Fall (T-APP): Es sei gegeben der Term $t = t_1 t_2$ sowie $\Gamma \vdash t_1 : R \rightarrow X$ und $\Gamma \vdash t_2 : R$.
Aus (T-APP) folgt, dass falls $\Gamma \vdash t_1 t_2 : T$ gilt, so ist $R \rightarrow X = R \rightarrow T$. Wir nehmen nun an, der Term t habe Ebenfalls den Typ S . Mit (T-APP) gilt wieder $R \rightarrow X = R \rightarrow S$. Laut Induktionsannahme gilt $T_1 = S_1$ für ein S_1 mit $\Gamma \vdash t_1 : R \rightarrow S_1$. Insgesamt folgt $R \rightarrow S = R \rightarrow X = R \rightarrow T$, d.h. $S = T$. ✓

Somit gilt die Behauptung unter Verwendung von (T-VAR), (T-ABS) und (T-APP). Ableitungsbäume sind somit eindeutig; weitere Fälle können analog gezeigt werden. \square

Bemerkung: Später werden für die Polymorphie Typvariablen statt konkreter Typen verwendet. Ein Ausdruck hat dann im Allgemeinen eine Menge von Instanziierungen, die typkorrekt sind.

2.3.2 Typerhalt bei Evaluierung

Wir möchten nun eine Minimalforderung an das Typsystem, nämlich dass der Typ eines Ausdrucks bei jedem Auswertungsschritt erhalten bleibt, beweisen.

Behauptung: Es seien $\Gamma \vdash t : T$ sowie $t \rightarrow t'$ (der Term t wertet - z.B. durch β -Reduktion - zu t' aus). Dann gilt $\Gamma \vdash t' : T$.

Beweis: Induktion über den Ableitungsbaum des Ausdrucks $\Gamma \vdash t : T$. Wir nehmen in jedem Schritt an, dass für Evaluierungen auf Teilableitungsbäumen die Behauptung gilt (also z.B., dass falls für eine Prämisse der betrachteten Ableitungsregel $\Gamma \vdash s : S$ und $s \rightarrow s'$ gilt, dann $\Gamma \vdash s' : S$ folgt). Die Evaluierungsregeln wurden mit gleichen Bezeichnungen aus [Pie02] übernommen.

- Fall (T-VAR): $t = x$. Es gibt keine Evaluierungsregeln, welche eine Variable umwandeln (erfüllt trivialerweise die Behauptung). ✓
- Fall (T-ABS): $t = \lambda x : T_1 . t_2$. Für einen Ausdruck dieser Form gibt es ebenfalls keine Evaluierungsregel. ✓
- Fall (T-APP): $t = t_1 t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$.
Es gibt drei Evaluierungsregeln, welche separat in Betracht gezogen werden:

- E-APP1: $t_1 \rightarrow t'_1 \quad t' = t'_1 t_2$. Laut Induktionsannahme ist $\Gamma \vdash t'_1 : T_{11} \rightarrow T_{12}$, womit insgesamt folgt, dass $\Gamma \vdash t' : T$ gilt. ✓
- E-APP2: $t_1 = v_1$ (d.h. t_1 ist ein Wert), $t_2 \rightarrow t'_2 \quad t' = v_1 t'_2$. Analog zu E-APP1. ✓
- E-APPABS: $t_1 = \lambda x : T_{11} . t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]t_{12}$. Aus $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$ folgt mit (T-ABS) $\Gamma, x : T_{11} \vdash t_{12} : T_{12}$. Mit dem Substitutionslemma³ folgt dann wegen $\Gamma \vdash v_2 : T_{11}$, dass $\Gamma \vdash [x \mapsto v_2]t_{12} : T_{12}$, und somit $\Gamma \vdash t' : T_{12}$. ✓

Womit die Behauptung unter Verwendung von (T-VAR), (T-ABS) und (T-APP) gilt. Weitere Fälle analog. \square

³Das Substitutionslemma besagt, dass falls $\Gamma, x : S \vdash t : T$ sowie $\Gamma \vdash s : S$ gilt, so folgt $\Gamma \vdash [x \mapsto s]t : T$. Der Beweis kann durch Induktion über den Ableitungsbaum erfolgen.

3 Implizite Typbestimmung

Wir möchten die bisherigen Begriffe nun so erweitern, dass wir die monomorphen Typregeln nun auf eine polymorphe Typherleitung erweitern. Wie bereits erwähnt werden Typvariablen verwendet, welche als Platzhalter fungieren und bei einer Instanziierung durch einen konkreten Typ ersetzt werden. Wir werden für einen Ausdruck mit Typvariablen alle möglichen typkorrekten Instanziierungen beschreiben wollen, wozu wir ein algorithmisches Verfahren (Typinferenz mit Bedingungen) verwenden. Schließlich soll dies in Form eines allgemeinsten Typs ausgedrückt werden, so dass hieraus alle Lösungen generiert werden können.

Letztlich wollen wir den Einsatz von polymorphen Ausdrücken im Rahmen der `let`-Konstrukte und den damit verbundenen Fragestellungen anreißen.

3.1 Substitution von Typvariablen

3.1.1 Substitution σ als Beschreibung oder Anwendung

Das Symbol σ wird folgendermaßen verwendet:

- als Beschreibung einer Typsubstitution: $\sigma = [X \mapsto Bool]$ ist eine Substitutionsregel, die alle Vorkommnisse von X durch $Bool$ ersetzt.
- als Anwendung einer Typsubstitution: $\sigma(X)$ ersetzt die Typvariablen X laut der Beschreibung. Im Beispiel ist $\sigma(X \rightarrow X) = Bool \rightarrow Bool$. Wir schreiben beispielsweise $[X \mapsto T, Y \mapsto U]$ für die Substitution, welche X auf T und Y auf U abbildet. Alle Ersetzungen sollen *gleichzeitig* vorgenommen werden, d.h. $[X \mapsto Y, Y \mapsto Z](X \rightarrow Y) = (Y \rightarrow Z)$.

3.1.2 Formale Beschreibung von σ

Eine *Typsubstitution* ist eine endliche Abbildung von Typvariablen auf Typen. Wir schreiben $dom(\sigma)$ für die Menge der Typvariablen, welche auf den jeweiligen linken Seiten der Paare erscheinen. Entsprechend schreiben wir $range(\sigma)$ für die Menge von Typen, welche auf den jeweiligen rechten Seiten der Paare stehen.

Beispiel: Für $[X \mapsto T, Y \mapsto U]$ ist $dom(\sigma) = \{X, Y\}$ und $range(\sigma) = \{T, U\}$.

Anmerkung: Die Elemente von $\text{range}(\sigma)$ können ebenfalls Typvariablen sein.

Anwendung:

$$\begin{aligned}\sigma(X) &= \begin{cases} T & , \text{ falls } (X \mapsto T) \in \sigma \\ X & , \text{ falls } X \notin \text{dom}(\sigma) \end{cases} \\ \sigma(\text{Bool}) &= \text{Bool} \\ \sigma(T_1 \rightarrow T_2) &= \sigma T_1 \rightarrow \sigma T_2 \\ \sigma(x_1 : T_1, \dots, x_n : T_n) &= (x_1 : \sigma T_1, \dots, x_n : \sigma T_n) \\ \sigma \circ \gamma &= \left[\begin{array}{ll} X \mapsto \sigma(T) & \text{für alle } (X \mapsto T) \in \gamma \\ X \mapsto T & \text{für alle } (X \mapsto T) \in \sigma \text{ mit } X \notin \text{dom}(\gamma) \end{array} \right]\end{aligned}$$

Beachte: $(\sigma \circ \gamma)S = \sigma(\gamma S)$. Für die Anwendung von σ auf einen Term t gilt außerdem:

$$\begin{aligned}\sigma(x : X) &= x : \sigma(X) \text{ (Variable)} \\ \sigma(\lambda x : X . t : Y) &= \lambda x : \sigma(X) . \sigma(t) : \sigma(Y) \\ \sigma(t_1 t_2 : X) &= \sigma(t_1) \sigma(t_2) : \sigma(X) \\ \sigma(\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T) &= \text{if } \sigma(t_1) \text{ then } \sigma(t_2) \text{ else } \sigma(t_3) : \sigma(T)\end{aligned}$$

Beobachtung: Die Substitution ist eindeutig.

3.1.3 Typerhalt bei σ -Substitution

Eine essentielle Eigenschaft der Typsubstitution ist, dass sie die Typkorrektheit erhält:

Behauptung: Es sei σ eine beliebige Typ-Substitution. Aus $\Gamma \vdash t : T$ folgt $\sigma\Gamma \vdash \sigma t : \sigma T$.

Beweis: Induktion über den Ableitungsbaum des Ausdrucks $\Gamma \vdash t : T$.

- Fall (T-VAR): $t = x$. Für die Variable x gilt $\sigma x = x$. Aus $\Gamma \vdash x : T$ folgt mit (T-VAR) $x : T \in \Gamma$, d.h. $x : T \in \Gamma_1, x : T, \Gamma_2$.
Es gilt $x : \sigma T \in \Gamma_1, x : \sigma T, \Gamma_2$ (Eindeutigkeit der Substitution), sowie auch $x : \sigma T \in \sigma\Gamma_1, x : \sigma T, \sigma\Gamma_2$, d.h. $x : \sigma T \in \sigma\Gamma$. Aus (T-VAR) folgt schließlich $\sigma\Gamma \vdash x : \sigma T$, was $\sigma\Gamma \vdash \sigma t : \sigma T$ entspricht. ✓
- Fall (T-ABS): Gegeben $\Gamma \vdash \lambda x : T_1 . t : T_1 \rightarrow T_2$. Aus (T-ABS) folgt $\Gamma, x : T_1 \vdash t : T_2$. Laut Induktionsvoraussetzung gilt $\sigma\Gamma, \sigma x : \sigma T_1 \vdash \sigma t : \sigma T_2$ und mit (T-ABS) wiederum mit $\sigma x = x$ und $\sigma(\lambda x : T_1) = \lambda x : \sigma T_1$ schließlich $\sigma\Gamma \vdash \lambda x : \sigma T_1 . \sigma t : \sigma T_1 \rightarrow \sigma T_2$. ✓

- Fall (T-APP): Gegeben $\Gamma \vdash t_1 t_2 : T_2$. Aus (T-APP) folgt $\Gamma \vdash t_1 : T_1 \rightarrow T_2$ und $\Gamma \vdash t_2 : T_1$ für ein T_1 . Laut Induktionsvoraussetzung gilt $\sigma\Gamma \vdash \sigma t_1 : \sigma T_1 \rightarrow \sigma T_2$ und $\sigma\Gamma \vdash \sigma t_2 : \sigma T_1$ und mit (T-APP) wiederum $\sigma\Gamma \vdash \sigma t_1 \sigma t_2 : \sigma T_2$. ✓

Somit gilt die Behauptung unter Verwendung von (T-VAR), (T-ABS) und (T-APP). Weitere Fälle analog. □

3.2 Typ als Lösung von (Γ, t)

Um die Typrekonstruktion zu ergründen, benötigen wir noch eine formale Schreibweise für die möglichen Substitutionen von Typvariablen, welche einen typkorrekten Term erzeugen.

3.2.1 Formale Lösung von (Γ, t)

Es sei Γ eine Umgebung und t ein Term. Eine *Lösung* von (Γ, t) ist ein Paar (σ, T) , so dass $\sigma\Gamma \vdash \sigma t : T$ gilt.

Dabei gibt es im Allgemeinen mehrere Lösungen.

3.2.2 Beispiel

Es seien $\Gamma = f : X, a : Y$ sowie $t = f a$. Dann sind

- $([X \mapsto Y \rightarrow Bool], Bool)$
- $([X \mapsto Y \rightarrow Z, Z \mapsto Bool], Z)$ (wegen 3.1.1)
- $([X \mapsto Bool \rightarrow Bool, Y \mapsto Bool], Bool)$
- $([X \mapsto Y \rightarrow Bool \rightarrow Bool], Bool \rightarrow Bool)$

einige Lösungen von (Γ, t) .

3.3 Typinferenz mit Bedingungen [constraint based typing]

Die im vorigen Abschnitt beschriebenen Lösungen von (Γ, t) können wir im Moment nur durch Probieren finden. Stattdessen ist das Ziel, Aussagen über die Menge der Lösungen treffen zu können – diese sollen es uns dann ermöglichen, jeden Ausdruck daraufhin zu prüfen, ob er Element der Lösung ist.

3.3.1 Bestimmung von Bedingungen \mathcal{C}

Wir werden nun einen Algorithmus kennenlernen, der zu einem gegebenen Term t mit Umgebung Γ einen Typ S und eine Menge von Bedingungen \mathcal{C} berechnet.

3.3.1.1 Definition und Erfüllung von Bedingungen:

Mit \mathcal{C} bezeichnen wir eine Menge von Bedingungen $\{S_i = T_i\}_{i \in 1 \dots n}$. Die einzelnen Gleichungen können hierbei Typvariablen beinhalten. Eine Substitution σ *unifiziert* (erfüllt) eine Gleichung $S = T$ dann, wenn σS und σT identisch sind. Eine Substitution σ *unifiziert* (erfüllt) \mathcal{C} dann, wenn σ jede Gleichung aus \mathcal{C} unifiziert.

Wir werden in 3.4 mehr zur Unifikation erfahren.

3.3.1.2 Erzeugung von Bedingungen:

Im Unterschied zur expliziten Typbestimmung werden hier die für jede Ableitung notwendigen Zusammenhänge zwischen den Prämissen nicht sofort geprüft, sondern in den Bedingungen aufgezeichnet.

Beispiel: Für eine Applikation $t_1 t_2$ mit $\Gamma \vdash t_1 : T_1$ und $\Gamma \vdash t_2 : T_2$ wird bei der expliziten Typbestimmung geprüft, ob $T_1 = T_2 \rightarrow R$ gilt. Hier wird stattdessen mit einer noch unbenutzten Typvariable X festgehalten, dass $T_1 = T_2 \rightarrow X$ (wird in die Bedingungen \mathcal{C} aufgenommen). Für die Applikation wird der Typ X zurückgegeben. Die Applikation hat also den Typ X , falls $T_1 = T_2 \rightarrow X$ gilt.

3.3.1.3 Lösung von (Γ, t)

Ein Paar (σ, T) heißt *Lösung* von $(\Gamma, t, S, \mathcal{C})$, falls $\sigma S = T$ und die Bedingungen \mathcal{C} gemäß 3.3.1.1 erfüllt werden.

3.3.2 Ableitungsregeln mit Bedingungen

Notation: $\Gamma \vdash t : T \mid \mathcal{C}$ kann informell gelesen werden als „der Term t hat Typ T in der Umgebung Γ , falls die Bedingungen \mathcal{C} erfüllt sind“ (siehe eben in Abschnitt 3.3.1.2). Dabei ist T im Allgemeinen eine Typvariable.

Vereinfachung: Es wurden einige technische Details aus Gründen der Übersichtlichkeit weggelassen. So müssten die benutzten Variablennamen in jedem Schritt aufgezeichnet werden, so dass zwei Sachverhalte sichergestellt werden:

1. alle Typvariablen sind einzigartig, d.h. neu eingeführte Typvariablen unterscheiden sich von den bisher vorkommenden Typvariablen.
2. bei Ableitungsregeln, die mehr als eine Prämisse haben, muss der Schnitt *aller* verwendeten Typvariablen leer sein (keine Überschneidungen).

Diese Bedingungen sind immer erfüllbar, da der Vorrat von Bezeichnern (abzählbar) unendlich ist (wodurch schon Punkt 1 erfüllt wird) und durch Umbenennung der Typvariablen Punkt 2 gewährleistet werden kann. Da wir die Ableitungsregeln allerdings nicht implementieren, können wir über diese Details hinwegsehen um uns dem Wesentlichen zu widmen.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid \{\}} \quad (\text{CT-VAR})$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid \mathcal{C}}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \mid \mathcal{C}} \quad (\text{CT-ABS})$$

Diese Regeln erweitern die bereits bekannten Ableitungsregeln (T-VAR) und (T-ABS) aus Abschnitt 2.2.1.2 um Bedingungen \mathcal{C} (jeweils nach dem Trennstrich \mid).

Implizite Typannotation

Programmiersprachen, die Typrekonstruktion unterstützen, erlauben üblicherweise das Auslassen von Typannotationen bei Abstraktionen (also z.B. statt $\lambda x : T_1 . t_2$ nur $\lambda x . t_2$ zu schreiben).

Eine Lösung hierfür wäre, dem Parser das Einfügen einer noch unbenutzten Typvariable zu überlassen. Eine elegantere Lösung hierfür ist die Einführung einer Ableitungsregel für unannotierte Abstraktionen, welche eine noch unbenutzte „frische“ Typvariable X einführt.

Im Hinblick auf **let**-Polymorphismus in 3.6 sei außerdem erwähnt, dass bei eventuellen Kopien einer unannotierten Abstraktion jeweils eine *unterschiedliche* neue Typvariable eingeführt wird; im Gegensatz dazu würden Kopien den gleichen Typ erhalten, wenn wir uns – wie in der ersten Alternative – die unannotierte Abstraktion mit einer „unsichtbaren“ Typvariablen denken (vgl. hierzu auch [Pie02] S. 330-331).

Hier schließlich die erwähnte unannotierte Ableitungsregel, welche eine „frische“ Typvariable X einführt:

$$\frac{\Gamma, x : X \vdash t_1 : T \mid \mathcal{C}}{\Gamma \vdash \lambda x . t_2 : X \rightarrow T \mid \mathcal{C}} \quad (\text{CT-ABSINF})$$

Weitere Ableitungsregeln

$$\frac{\Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2 \quad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = T_2 \rightarrow X\}}{\Gamma \vdash t_1 t_2 : X \mid \mathcal{C}'} \quad (\text{CT-APP})$$

Bei (CT-APP) trifft die Vereinfachung zu. So müssten wir sicherstellen, dass X eine „frische“ Variable ist, dass die verwendeten Variablentypen in der linken und rechten Prämisse sich nicht gegenseitig oder mit den freien Variablen $FV(T)$ des jeweils anderen Typs schneiden.

$$\Gamma \vdash true : Bool \mid \{\}$$
 (CT-TRUE)

$$\Gamma \vdash false : Bool \mid \{\}$$
 (CT-FALSE)

$$\frac{\Gamma \vdash t_1 : T_2 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2 \quad \Gamma \vdash t_3 : T_3 \mid \mathcal{C}_3 \quad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{T_1 = Bool, T_2 = T_3\}}{\Gamma \vdash f t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \mid \mathcal{C}'}$$
 (CT-IF)

Auch bei (CT-IF) wäre zusätzlich noch sicherzustellen, dass alle Typvariablen, die in den drei Prämissen verwendet werden, sich nicht überschneiden. Wie bereits erwähnt haben wir immer die Möglichkeit, dies zu bewerkstelligen: Umbenennung aller gleichen Typvariablen in einem der Ausdrücke (eventuell in zweien) leistet das gewünschte ohne das Ergebnis „modulo Umbenennung“ zu verfälschen.

3.3.3 Beispiel

Finde S und \mathcal{C} , so dass $\vdash \lambda x : X . \lambda y : Y . \lambda z : Z . (x z)(y z) : S \mid \mathcal{C}$.

Ableitungsbaum mit den Ableitungsregeln (CT-ABS), (CT-APP) und (CT-VAR):

$$\frac{\frac{\frac{x : X \in \Gamma \quad z : Z \in \Gamma}{\Gamma \vdash x : X} \quad \frac{z : Z \in \Gamma}{\Gamma \vdash z : Z}}{\mathcal{C}_1 := \{X = Z \rightarrow A\}} \quad \frac{\frac{y : Y \in \Gamma \quad z : Z \in \Gamma}{\Gamma \vdash y : Y} \quad \frac{z : Z \in \Gamma}{\Gamma \vdash z : Z}}{\mathcal{C}_2 := \{Y = Z \rightarrow B\}}}{\Gamma \vdash (x z) : A \mid \mathcal{C}_1 \quad \Gamma \vdash (y z) : B \mid \mathcal{C}_2} \text{CT-APP}$$

$$\frac{\Gamma := x : X, y : Y, z : Z \vdash (x z)(y z) : C \mid \mathcal{C}'}{\Gamma := x : X, y : Y, z : Z \vdash \lambda z : Z . (x z)(y z) : Z \rightarrow C \mid \mathcal{C}'} \text{CT-ABS}$$

$$\frac{x : X, y : Y \vdash \lambda z : Z . (x z)(y z) : Z \rightarrow C \mid \mathcal{C}'}{x : X \vdash \lambda y : Y . \lambda z : Z . (x z)(y z) : Y \rightarrow (Z \rightarrow C) \mid \mathcal{C}'} \text{CT-ABS}$$

$$\frac{x : X \vdash \lambda y : Y . \lambda z : Z . (x z)(y z) : Y \rightarrow (Z \rightarrow C) \mid \mathcal{C}'}{\vdash \lambda x : X . \lambda y : Y . \lambda z : Z . (x z)(y z) : X \rightarrow (Y \rightarrow (Z \rightarrow C)) \mid \mathcal{C}'} \text{CT-ABS}$$

Also:

- $S = X \rightarrow (Y \rightarrow (Z \rightarrow C)) = X \rightarrow Y \rightarrow Z \rightarrow C$
- $\mathcal{C} = \{X = Z \rightarrow A, Y = Z \rightarrow B, A = B \rightarrow C\}$

3.3.4 Äquivalenz der Lösungen für (Γ, t) und (Γ, t, S, C)

Wir möchten nun beweisen, dass die Typinferenz mit Bedingungen uns tatsächlich die Lösungen von (Γ, t) beschreibt, die wir ursprünglich gesucht haben.

3.3.4.1 Lemma „ \Leftarrow “ [soundness of constraint typing]

Behauptung: Es sei $\Gamma \vdash t : S \mid \mathcal{C}$. Falls (σ, T) eine Lösung von $(\Gamma, t, S, \mathcal{C})$ ist, so ist (σ, T) auch eine Lösung von (Γ, t) .

Beweis: Induktion über den Ableitungsbaum von $\Gamma \vdash t : S \mid \mathcal{C}$.

- Fall (CT-VAR): $t = x$. Gegeben $x : S \in \Gamma$, $\mathcal{C} = \{\}$.
Es gilt, dass (σ, T) eine Lösung von $(\Gamma, t, S, \mathcal{C})$ ist. Da $\mathcal{C} = \emptyset$ bedeutet dies schlicht, dass $\sigma S = T$. Mit (T-VAR) erhalten wir aus $x : \sigma S \in \sigma\Gamma$ direkt $\sigma\Gamma \vdash x : T$. ✓
- Fall (CT-ABS): $t = \lambda x : T_1 . t_2$. Gegeben $S = T_1 \rightarrow S_2$, $\Gamma, x : T_1 \vdash t_2 : S_2 \mid \mathcal{C}$.
Es gilt, dass (σ, T) eine Lösung von $(\Gamma, t, S, \mathcal{C})$ ist, also erfüllt σ die Bedingungen \mathcal{C} und es gilt $T = \sigma S = \sigma T_1 \rightarrow \sigma S_2$. Also ist $(\sigma, \sigma S_2)$ eine Lösung der Prämisse $((\Gamma, x : T_1), t_2, S_2, \mathcal{C})$.
Laut Induktionsvoraussetzung ist $(\sigma, \sigma S_2)$ auch Lösung von $((\Gamma, x : T_1), t_2)$, d.h. $\sigma\Gamma, x : \sigma T_1 \vdash \sigma t_2 : \sigma S_2$. Mit (T-ABS) folgt aber $\sigma\Gamma \vdash \lambda x : \sigma T_1 . \sigma t_2 : \sigma T_1 \rightarrow \sigma S_2 = \sigma(T_1 \rightarrow S_2) = T$, was zu zeigen war. ✓
- Fall (CT-APP): $t = t_1 t_2$. Gegeben $S = X$, $\Gamma \vdash t_1 : S_1 \mid \mathcal{C}_1$, $\Gamma \vdash t_2 : S_2 \mid \mathcal{C}_2$, $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 = S_2 \rightarrow X\}$.
Laut Definition erfüllt σ die Bedingungen \mathcal{C}_1 und \mathcal{C}_2 und es gilt $\sigma S_1 = \sigma(S_2 \rightarrow X)$. Somit sind $(\sigma, \sigma S_1)$ und $(\sigma, \sigma S_2)$ Lösungen für $(\Gamma, t_1, S_1, \mathcal{C}_1)$ und $(\Gamma, t_2, S_2, \mathcal{C}_2)$. Laut Induktionsvoraussetzung gilt $\sigma\Gamma \vdash \sigma t_1 : \sigma S_1$ und $\sigma\Gamma \vdash \sigma t_2 : \sigma S_2$. Wegen $\sigma S_1 = \sigma S_2 \rightarrow \sigma X$ gilt aber $\sigma\Gamma \vdash \sigma t_1 : \sigma S_2 \rightarrow \sigma X$. Mit (T-APP) folgt schließlich $\sigma\Gamma \vdash \sigma(t_1 t_2) : \sigma X$ mit $\sigma X = \sigma S = T$, was zu beweisen war. ✓
- weitere Fälle analog. \square

3.3.4.2 Lemma „ \Rightarrow “ [completeness of constraint typing]

Behauptung: Es sei $\Gamma \vdash t : S \mid \mathcal{C}$. Falls (σ, T) eine Lösung für (Γ, t) , dann ist (σ, T) auch eine Lösung für $(\Gamma, t, S, \mathcal{C})$.

Beweis: Der Beweis nimmt an, dass Typvariablen einzigartig benannt sind (siehe 3.3.2). Ansonsten wären weitere Vorkehrungen zu treffen und Konzepte einzuführen, um den Beweis zu führen.

- Fall (CT-VAR): (σ, T) ist eine Lösung von (Γ, x) , d.h. $\sigma\Gamma \vdash x : \sigma S = T$. Dann ist (σ, T) auch eine Lösung von $(\Gamma, x, S, \{\})$ ✓
- Fall (CT-ABS): $t = \lambda x : T_1 . t_2 \quad \Gamma, x : T_1 \vdash t_2 : S_2 \mid \mathcal{C} \quad S = T_1 \rightarrow S_2$.
Es gilt, dass (σ, T) eine Lösung von $(\Gamma, \lambda x : T_1 . t_2)$. Aus (T-ABS) folgt $\sigma\Gamma, x : \sigma T_1 \vdash \sigma t_2 : T_2$ und $T = \sigma T_1 \rightarrow T_2$ für ein T_2 .
Laut Induktionsvoraussetzung ist (σ, T_2) auch eine Lösung von $(\Gamma, x : \sigma T_1, t_2, S_2, \mathcal{C})$. Es gilt also $\sigma S = \sigma T_1 \rightarrow \sigma S_2 = \sigma T_1 \rightarrow T_2 = T$ ✓
- Fall (CT-APP): $t = t_1 t_2 \quad \Gamma \vdash t_1 : S_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : S_2 \mid \mathcal{C}_2 \quad S = X$
 $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 = S_2 \rightarrow X\}$. Es gilt, dass (σ, T) eine Lösung von $(\Gamma, t_1 t_2)$ ist. Aus (T-APP) folgt sofort $\sigma\Gamma \vdash \sigma t_1 : T_1 \rightarrow T$ und $\sigma\Gamma \vdash \sigma t_2 : T_1$ für ein T_1 .
Laut Induktionsvoraussetzung ist $(\sigma, T_1 \rightarrow T)$ und (σ, T_1) auch eine Lösung von $(\Gamma, t_1, S_1, \mathcal{C}_1)$ und $(\Gamma, t_2, S_2, \mathcal{C}_2)$. Es gilt also $\sigma S_1 = T_1 \rightarrow T$ und $\sigma S_2 = T_1$, also $\sigma S_1 = T_1 \rightarrow T = \sigma S_2 \rightarrow \sigma X = \sigma(S_2 \rightarrow X)$ ✓
- weitere Fälle analog. \square

Satz von die Äquivalenz der Lösungen von (Γ, t) und $(\Gamma, t, S, \mathcal{C})$

Behauptung: (σ, T) Lösung von $(\Gamma, t) \iff (\sigma, T)$ Lösung von $(\Gamma, t, S, \mathcal{C})$

Beweis: Lemma „ \Leftarrow “ und Lemma „ \Rightarrow “.

Somit sind beide Lösungen äquivalent, womit wir die algorithmische Variante mit Bedingungen benutzen können, um die Menge aller Lösungen zu beschreiben.

3.4 Unifikation [unification]

Es ist schließlich erstrebenswert, das Resultat der Typinferenz mit Bedingungen dahingehend zu verfeinern, dass wir aus dem Paar (S, \mathcal{C}) einen Typ folgern, der die Menge der Lösungen bestmöglich beschreibt. Aus einem solchen „allgemeinsten Typ“ sollen schließlich alle möglichen Lösungen generierbar sein.

3.4.1 Allgemeinere und allgemeinste Substitution

Allgemeinere Substitution: Eine Substitution σ heißt *allgemeiner* als eine Substitution σ' , in Zeichen $\sigma \sqsubseteq \sigma'$, falls $\sigma' = \gamma \circ \sigma$ für eine Substitution γ .

Allgemeinste Substitution: Ein *allgemeinster Unifikator* [principal unifier, most general unifier] für eine Menge von Bedingungen \mathcal{C} ist eine Substitution σ (die \mathcal{C} im Sinne von 3.3.1.1 erfüllt) so dass für alle σ' (die ebenfalls \mathcal{C} erfüllen) gilt: $\sigma \sqsubseteq \sigma'$.

Es gilt offensichtlich, dass falls es für einen Term eine Instanziierung (Substitution) von Typvariablen gibt, so dass der Term typkorrekt ist, so hat dieser einen allgemeinsten Unifikator (eine allgemeinste Substitution).

Beispiel: Man bestimme den allgemeinsten Unifikator (falls vorhanden) für folgende Mengen von Bedingungen:

- $\{X = Bool, Y = X \rightarrow X\} : [X \mapsto Bool, Y \mapsto Bool \rightarrow Bool]$
- $\{Y = Bool \rightarrow Y\} : \text{Nicht unifizierbar. Warum?}$
- $\{Bool \rightarrow Bool = X \rightarrow Y\} : [X \mapsto Bool, Y \mapsto Bool]$

3.4.2 Unifikations-Algorithmus

Der Unifikationsalgorithmus leistet im Prinzip das, was wir im Beispiel ergründet haben.

`unify(C) =`

```

if C = ∅ then []
else let {S = T} ∪ C' = C in
  if S = T then
    unify(C')
  else if S = X and X ∉ FV(T) then
    unify([X ↦ T]C') ∘ [X ↦ T]
  else if T = X and X ∉ FV(S) then
    unify([X ↦ S]C') ∘ [X ↦ S]
  else if S = S1 → S2 and T = T1 → T2 then
    unify(C' ∪ {S1 = T1, S2 = T2})
  else fail

```

Dabei ist die Anweisung „let $\{S = X\} \cup C' = C$ “ zu lesen als „wähle eine Bedingung $S = T$ aus der Menge C , wobei $C' = C \setminus \{S = T\}$ ist“.

Die Nebenbedingungen $X \notin FV(T)$ und $X \notin FV(S)$ werden auch als *occur check* bezeichnet. Sie verhindern die Erzeugung einer Lösung, welche zyklische Substitutionen wie $X \mapsto X \rightarrow X$ beinhaltet; dadurch können wir unsere Lösungen endlich halten (d.h. sie beinhalten keine rekursiven Typen).

Dies ist in unserem Zusammenhang insofern sinnvoll, als dass wir Ausdrücke mit einem nicht endlichen Typ gar nicht eingeführt haben. Sollen hingegen rekursive Typen erlaubt sein, so kann der *occur check* ausgelassen werden.

Bemerkung: Der Unifikationsalgorithmus terminiert immer:

- er bricht ab falls eine nicht-unifizierbare Menge von Bedingungen als Eingabe erfolgt,
- sonst gibt er einen allgemeinsten Unifikator zurück.

Beweis z.B. in [Pie02] S. 328-329.

3.4.3 Allgemeinsten Typ eines Terms

Wir haben in Abschnitt 3.4.1 beobachtet: gibt es eine Möglichkeit, Typvariablen eines Terms zu instanziiieren, so dass der Term typkorrekt ist, so gibt es eine allgemeinste Möglichkeit, dies zu tun. Wir möchten diesen Sachverhalt nun formalisieren und einen „allgemeinsten Typ“ berechnen können.

Allgemeinste Lösung: Eine *allgemeinste Lösung* für $(\Gamma, t, S, \mathcal{C})$ ist eine Lösung (σ, T) , so dass für alle weiteren Lösungen (σ', T') von $(\Gamma, t, S, \mathcal{C})$ gilt: $\sigma \sqsubseteq \sigma'$. Der Unifikationsalgorithmus aus Abschnitt 3.4.2 erlaubt uns zu entscheiden, ob $(\Gamma, t, S, \mathcal{C})$ eine Lösung hat (wenn der Algorithmus abbricht, gibt es keine Lösung). Falls es eine Lösung gibt, so können wir mit dem Algorithmus weiterhin die allgemeinste Lösung von $(\Gamma, t, S, \mathcal{C})$ berechnen¹.

Allgemeinster Typ: Falls (σ, T) die allgemeinste Lösung von $(\Gamma, t, S, \mathcal{C})$ ist, so nennen wir T den *allgemeinsten Typ* von t unter den Annahmen Γ .

Fazit: Wir können dank dieser Gegebenheiten also für jeden typkorrekten Term t die allgemeinste Lösung und schließlich den allgemeinsten Typ für (Γ, t) bestimmen, indem wir die Typinferenz mit Bedingungen und den Unifikationsalgorithmus verwenden. Dies wird in Abschnitt 3.5 zusammengefasst.

3.4.4 Beispiel

Wir greifen das Beispiel aus Abschnitt 3.4.2 wieder auf. Durch Verwendung des bedingungs-basierten Typinferenzalgorithmus hatten wir für den Ausdruck

$$\lambda x : X . \lambda y : Y . \lambda z : Z . (x z)(y z)$$

S und \mathcal{C} bestimmt, so dass gilt:

$$\vdash \lambda x : X . \lambda y : Y . \lambda z : Z . (x z)(y z) : S \mid \mathcal{C}$$

¹Folgt aus der Definition einer Lösung für $(\Gamma, t, S, \mathcal{C})$ und den Eigenschaften der Unifikation. Daraus folgt auch mit der Äquivalenz der Lösungen von $(\Gamma, t, S, \mathcal{C})$ und (Γ, t) , dass es entscheidbar ist, ob (Γ, t) eine Lösung besitzt.

Dabei wurden folgende Werte ermittelt:

- $S = X \rightarrow (Y \rightarrow (Z \rightarrow C)) = X \rightarrow Y \rightarrow Z \rightarrow C$
- $\mathcal{C} = \{X = Z \rightarrow A, Y = Z \rightarrow B, A = B \rightarrow C\}$

Wir wenden nun den Unifikationsalgorithmus auf die Menge der Bedingungen \mathcal{C} an und erhalten laut Definition eine allgemeinste Substitution σ :

$$\begin{aligned}
 \text{unify}(\mathcal{C}) &= \text{unify}([X \mapsto Z \rightarrow A]\{Y = Z \rightarrow B, A = B \rightarrow C\}) \circ [X \mapsto Z \rightarrow A] \\
 &= \text{unify}(\{Y = Z \rightarrow B, A = B \rightarrow C\}) \circ [X \mapsto Z \rightarrow A] \\
 &= \text{unify}([Y \mapsto Z \rightarrow B]\{A = B \rightarrow C\}) \circ [Y \mapsto Z \rightarrow B] \circ [X \mapsto Z \rightarrow A] \\
 &= \text{unify}(\{A = B \rightarrow C\}) \circ [Y \mapsto Z \rightarrow B, X \mapsto Z \rightarrow A] \\
 (i) &= \text{unify}([A \mapsto B \rightarrow C]\{\}) \circ [A \mapsto B \rightarrow C] \circ [Y \mapsto Z \rightarrow B, X \mapsto Z \rightarrow A] \\
 (ii) &= \text{unify}(\{\}) \circ [A \mapsto B \rightarrow C, Y \mapsto Z \rightarrow B, X \mapsto Z \rightarrow A] \\
 &= [] \circ [A \mapsto B \rightarrow C, Y \mapsto Z \rightarrow B, X \mapsto Z \rightarrow A] \\
 &= [A \mapsto B \rightarrow C, Y \mapsto Z \rightarrow B, X \mapsto Z \rightarrow A] =: \sigma
 \end{aligned}$$

Interpretation des Resultats: Der Unifikationsalgorithmus liefert eine Substitution, die nicht zeitgleich, sondern von rechts nach links (wie bei einer Funktionskomposition $f \circ g(x) = f(g(x))$) anzuwenden ist. Dies entspricht nicht mehr dem Begriff der Substitution, wie wir ihn formal in Abschnitt 3.1.2 kennengelernt haben.

Warum? Wollten wir eine Substitution im Sinne von Abschnitt 3.1.2 erhalten, so müssten wir im Algorithmus die neue Ersetzung (z.B. bei Zeile (i) $[A \mapsto B \rightarrow C]$) auf die bereits vorhandenen Substitutionen (im Beispiel $[Y \mapsto Z \rightarrow B, X \mapsto Z \rightarrow A]$) anwenden, womit Zeile (ii) (und folgenden Entsprechend) statt $X \mapsto Z \rightarrow A$ dann $X \mapsto Z \rightarrow (B \rightarrow C)$ als Substitution besitzen.

Wie leicht ersichtlich ist, ist die Anwendung dieses Ausdruckes simultan möglich, da keine der Typvariablen aus $\text{range}(\sigma)$ auch in $\text{dom}(\sigma)$ vorkommen. Insbesondere wäre dieser Ausdruck auch sequenziell (von rechts nach links oder links nach rechts) anwendbar.

Es ist allerdings nachvollziehbar, dass für die Konstruktion der eben erwähnten Substitution die bereits generierten Substitutionen in jedem Schritt nochmals durchlaufen werden müssen. Außerdem ist eine Substitution, die zeitgleich geschehen soll, aufwendiger zu realisieren, als Schrittweise die Ersetzungen anzuwenden. Somit wäre eine solche Alternative in jeder Hinsicht kostspieliger, und das ohne Mehrwert.

Anwendung der Substitution: Durch Anwendung von σ auf S erzeugen wir nun den allgemeinste Typ, welcher die Bedingungen \mathcal{C} auf Grund der Konstruktion von σ eben erfüllt:

$$\begin{aligned}\sigma S &= [A \mapsto B \rightarrow C \ Y \mapsto Z \rightarrow B] (Z \rightarrow A) \rightarrow Y \rightarrow Z \rightarrow C \\ &= [A \mapsto B \rightarrow C] (Z \rightarrow A) \rightarrow (Z \rightarrow B) \rightarrow Z \rightarrow C \\ &= (Z \rightarrow B \rightarrow C) \rightarrow (Z \rightarrow B) \rightarrow Z \rightarrow C\end{aligned}$$

Die Auswertung (in OCaml) von `fun x -> fun y -> fun z -> (x z) (y z);;` ergibt den (bis auf Umbenennung der Variablen) äquivalenten Ausdruck

```
- : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Womit wir den Typinferenzmechanismus von ML im Wesentlichen beschrieben hätten.

3.5 Zusammenfassung der Typrekonstruktion

Die Schritte zur Bestimmung des allgemeinsten Typs sind:

1. Bestimme S und \mathcal{C} für (Γ, t) mittels der Typinferenz (3.3.2)
2. Bestimme mit dem Unifikationsalgorithmus $\sigma := \text{unify}(\mathcal{C})$ (3.4.2)
3. Berechne σS , was den allgemeinsten Typ liefert².

Es ist auch möglich, bei jedem Schritt der Typinferenz einen allgemeinsten Typ zu bestimmen. Dadurch ist die erneute Analyse von Subtermen gespart, außerdem ist die Lokalisierung von Fehlern im Programm viel genauer (vgl. [Pie02] S. 330).

3.6 Let-Polymorphismus

Der Typrekonstruktionsalgorithmus kann insofern erweitert werden, als dass eine einfache Form von Polymorphie bereitgestellt werden kann. Diese Form wird als *let-Polymorphismus* (auch ML-Style oder Damas-Milner-Polymorphismus) bezeichnet. Dieses *Feature* wurde im Originaldialekt ML (Milner, 1978) eingeführt und formt die Basis für generische Bibliotheken für unterschiedliche Strukturen wie Listen, Arrays, Bäume, u.v.a.

²Vergleiche auch [Nip04] Theorem 3.3.2: „Betrachte Regeln als Prolog-Programm. Prolog berechnet die allgemeinste Lösung. Die Regeln sind deterministisch, d.h. es gibt maximal eine Lösung.“

3.6.1 Anforderungen an ein `let`-Konstrukt

Wir nähern uns dem Begriff mit einem motivierenden Beispiel. Nehmen wir an, wir möchten eine Funktion `double` definieren (mit `let double = t`), welche das erste ihr übergebene Argument (hier zunächst mit $(Nat \rightarrow Nat)$ annotiert) zweifach hintereinander auf das zweite Argument (hier vom Typ Nat) anwendet:

```
let double = λf : Nat → Nat . λa : Nat . f(f(a)) in
double (λx : Nat . succ (succ x)) 2;
```

Der Vollständigkeit halber sei erwähnt, dass die Funktion `succ` eine natürliche Zahl x auf ihren Nachfolger (im Sinne von Peano, d.h. $x + 1$) abbilden soll, und dass der Typ Nat die Funktion `succ` : $Nat \rightarrow Nat$ bereitstellt. Der Typ von `double` ist laut den Annotationen dann $(Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)$.

Wir können `double` alternativ auch als doppelte Anwendung einer $(Bool \rightarrow Bool)$ -Funktion auf ein $Bool$ -Argument definieren:

```
let double = λf : Bool → Bool . λa : Bool . f(f(a)) in
double (λx : Bool . x) false;
```

Möchten wir die gleiche `double`-Funktion für beide Typen Nat und $Bool$ (generisch) nutzen, so wäre eine Möglichkeit, zwei Funktionen zu definieren. Da dies dem Abstraktionsprinzip widerspricht³, wäre die Verwendung einer Typvariablen nach derzeitigem Wissensstand wohl der nächste denkbare Schritt:

```
let double = λf : X → X . λa : X . f(f(a)) in
let a = double (λx : Nat . succ(succ(x)) 1) in
let b = double (λx : Bool . x) false in...
```

Die implizite Typinferenz würde nun versuchen, die notwendigen Bedingungen für den Gesamtausdruck zu ermitteln. Dies geschieht für `double` sowohl bei `a` als auch bei `b`, was uns folgende unerfüllbare Bedingungen erzeugt:

$$X \rightarrow X = Nat \rightarrow Nat \quad (\text{Verwendung von } \mathbf{double} \text{ in } \mathbf{a})$$

$$X \rightarrow X = Bool \rightarrow Bool \quad (\text{Verwendung von } \mathbf{double} \text{ in } \mathbf{b})$$

womit das ganze Programm letztendlich nicht typkorrekt ist.

³Das Abstraktionsprinzip besagt, dass jedes signifikante Stück Funktionalität nur einmal im Quellcode vorkommen sollte, um Redundanz zu vermeiden.

Analyse: Wo wird das Problem erzeugt?

Die Typvariable X spielt zwei Rollen. Einerseits wird in der Definition von `a` festgehalten, dass das erste Argument von `double` eine Funktion ist, dessen Urbild- und Bildtyp gleich ist (hier $Nat \rightarrow Nat$) und weiterhin mit dem Typ des zweiten Arguments übereinstimmen (hier Nat). Entsprechendes wird in der Definition von `b` verlangt.

Da die *gleiche* Typvariable X beide Gegebenheiten erfüllen soll, erhalten wir den Konflikt.

3.6.2 Schrittweise Herleitung der `let`-Ableitung (CT-LetPoly)

Um diesen Zusammenhang aufzubrechen, möchten wir verschiedene Typvariablen bei jeder Verwendung von `double` einsetzen. Dies erhalten wir, indem wir die gewöhnliche Ableitungsregel dahingehend verändern, dass statt zuerst einen Typ für die rechte Seite t_1 zu ermitteln und diesen dann als Annotation für x (zur Bestimmung des Typs von t_2) zu verwenden:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad (\text{T-LET})$$

nun erst eine Substitution $[x \mapsto t_1]t_2$ und dann der Typ dieses neuen Terms bestimmt wird:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad (\text{T-LETPOLY})$$

Wir können dies zu einer Ableitungsregel für Typinferenz mit Bedingungen (siehe 3.3.2) erweitern:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \mid \mathcal{C}}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2 \mid \mathcal{C}} \quad (\text{CT-LETPOLY})$$

Im Wesentlichen ist die Ableitungsregel dahingehend verändert, als dass zunächst ein Auswertungsschritt

$$\text{let } x = v_1 \text{ in } t_2 \longrightarrow [x \mapsto v_1]t_2$$

vorgenommen wird, bevor der Typ rekonstruiert wird.

3.6.3 Hinzunahme der impliziten Annotation

Noch haben wir das ursprüngliche Problem mit der Typvariablen X noch nicht gelöst. Ein zweiter Schritt besteht darin, die Definition von `double` mit impliziten Typannotationen vorzunehmen (vgl. 3.3.2)

```
let double = λf . λa . f(f(a)) in
let a = double (λx : Nat . succ(succ(x))) 1 in
let b = double (λx : Bool . x) false in...
```

Die Kombination von (CT-LETPOLY) aus 3.6.2 und die Verwendung der impliziten Typnotation liefert uns schließlich das gewünschte Verhalten: (CT-LETPOLY) erzeugt zwei Kopien der Definition von `double` ($\lambda f . \lambda a . f(f(a))$), wonach (CT-ABSINF) aus Abschnitt 3.3.2 jeder dieser Abstraktionen eine unterschiedliche „frische“ Typvariable zuteilt. Die bisherigen Mechanismen erledigen dann das Übrige.

3.6.4 Verfeinerung von (T-LETPOLY)

Bevor dieses Schema in der Praxis anwendbar ist, müssen einige Makel noch angesprochen werden. Beispielsweise wäre ein Programm wie

```
let x = <völliger Blödsinn> in 5
```

typkorrekt. Um dem vorzubeugen, fügen wir eine Prämisse

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \quad \Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad (\text{T-LETPOLY})$$

ein, die sicherstellt, dass t_1 typkorrekt ist (wir können eine entsprechende Prämisse für (CT-LETPOLY) einfügen, vgl. [Pie02] S.333).

Es gibt weitere Verbesserungsbemühungen, insbesondere aus dem Bereich der Effizienzsteigerung: Falls beispielsweise eine mit `let` gebundene Variable (in (T-LETPOLY) wäre dies `x`) oft im Körper von `let` (also in t_2) auftritt, so wird die rechte Seite der `let`-Definition (d.h. t_1) bei jedem Vorkommen Typinferiert. Da die rechte Seite wiederum `let`-Bindungen enthalten kann, kann der Aufwand im schlechtesten Fall exponentiell wachsen (vgl. [Pie02] S.333). Diest kann allerdings nie ganz ausgeschlossen werden, wenngleich solche Beispiele in der Praxis eher eine Seltenheit darstellen.

Einige Konzepte des `let`-Polymorphismus wurden nun dargestellt. Die Vorstellung von weiteren Betrachtungen würde nun jedoch ein weiteres Ausholen und die Einführung weiterer Konzepte benötigen, weshalb dies dem interessierten Leser überlassen wird. Zum grundlegenden Einstieg in die Thematik wird [Pie02] empfohlen, für einen kürzeren Überblick [Nip04], sowie für Grundlagen der Logik [Sch00].

4 Zusammenfassung

Es wurden im Wesentlichen zwei Mechanismen für die Typbestimmung von λ -Termen vorgestellt:

- die explizite Typinferenz, welche durch die Erstellung eines Ableitungsbaumes jedem Ausdruck, der typkorrekt ist, einen eindeutigen Typ zuordnet
- die implizite Typinferenz, welche im Allgemeinen keine konkreten Typen voraussetzt. Dabei werden zwei algorithmische Verfahren eingesetzt, um einen allgemeinsten Typ zu bestimmen:
 - Typinferenz mit Bedingungen, wobei statt Terme auf die Vorgaben der Ableitungsregeln zu prüfen, diese Vorgaben in Bedingungen \mathcal{C} aufgezeichnet werden. Unter diesen Voraussetzungen hat dann der Ausdruck den Rückgabotyp.
 - Unifikation, welche aus dem Rückgabotyp und der Menge der Bedingungen einen allgemeinsten Typ (falls existent) generiert.

Schließlich wurde der `let`-Polymorphismus angeschnitten, welcher es erlaubt, Funktionen generisch zu definieren und in verschiedenen Kontexten zu verwenden (d.h. mit verschiedenen konkreten Typen).

Der Vorteil des Typrekonstruktionsverfahrens ist, dass keine Typangaben in Programmen vonnöten sind. Außerdem können Terme generisch verwendet werden, um damit Programmfunktionalität redundanzfrei zu halten.

Weiterführende Themen umfassen auf der praxisnahen Seite Sicherheitsaspekte, rekursive Typen, sowie mächtigere Formen von Polymorphie und anderer Typsysteme wie in *System F*, F_ω , u.v.a. in denen verschiedene Systeme zusammengeschmolzen werden – auf der theoretischen Seite können mittels dem „Curry-Howard-Isomorphismus“ verschiedene Formen von „reinen typisierten λ -Kalkülen“ und Logiken auf Zusammenhänge hin betrachtet werden.

Literaturverzeichnis

[Nip04] Tobias Nipkow. *Lambda-Kalkül*. 2004.

PDF Download: <http://www4.in.tum.de/lehre/vorlesungen/logik/WS0607/lambda.pdf>

[Pie02] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[Sch00] Uwe Schöning. *Logik für Informatiker*. Spektrum akademischer Verlag, 2000.