Proseminar "Programmiersysteme"

WS 2003/04

Thema: Typinferenz

Christian Kersten

Betreuer: Andreas Rossberg

8. April 2004

Zusammenfassung

Standard Typchecker benötigen sog. Typannotationen, d.h. jeder Lambda-Abstraktion muss explizit eine Typ annotiert werden. Wir werden einen Algorithmus präsentieren, der ohne diese Annotationen auskommt und trotzdem entscheiden kann, ob ein Term wohlgetypt ist oder nicht. Er ist zusätlich in der Lage für typisierbare Terme einen allgemeinsten Typ zu liefern. Wir werden im Folgenden den einfach getypte Lambda Kalkül verwenden. Wir beschränken uns dabei auf einfache Typen.

1 Der einfach getypte Lambda Kalkül

Als Grundlage dient uns der einfach getypte Lambda Kalkül, mit arithmetischen und boolschen Ausdrücken. Dabei sei Var eine unendliche Menge von Variablen und TV eine unendliche Menge von Typvariablen.

Terme:	t ::=	$x \in \mathsf{Var}$	(Variablen)
		$\mid \lambda x{:}\mathsf{T}.\mathsf{t}$	(Abstraktion)
		$\mid \mathsf{t}\ \mathsf{t}$	(Applikation)
		$\mid \mathsf{succ\ t} \mid \mathsf{pred\ t} \mid \mathsf{iszero\ t}$	(Operationen auf nat. Zahlen)
		$\mid 0$	(Null)
		$\mid \mathsf{true} \mid \mathsf{false}$	(Booleans)
		$\mid \mathsf{if\ t\ then\ t\ else\ t}$	(If)
Typen:	T::=	X	(Typvariablen)
		$\mid \mathsf{T} \to \mathsf{T}$	(Funtionstyp)
		$\mid \mathsf{Nat}$	(Natürliche Zahlen)
		$\mid \mathsf{Bool}$	(Booleans)

Dabei ist ein Term t unter einem Kontext Γ genau dann typisierbar, wenn ihm durch die Typregeln in Abbildung. 1 ein Typ zugeordnet werden kann. Der Kontext Γ dient dabei zur Bindung freier Variablen in t.

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

$$\frac{\Gamma' \vdash \mathsf{t}_2{:}\mathsf{T}' \quad \Gamma' = \Gamma, x{:}\mathsf{T}}{\Gamma \vdash \lambda x{:}\mathsf{T}.\, \mathsf{t}_2{:}\mathsf{T} \to \mathsf{T}'} \quad (\text{T-Abs})$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T} \to \mathsf{T}' \quad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \mathsf{t}_1\, \mathsf{t}_2 : \mathsf{T}'} \quad (\text{T-App})$$

$$\Gamma \vdash 0 : \mathsf{Nat} \quad (\text{T-Zero})$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ\, t}_1 : \mathsf{Nat}} \quad (\text{T-Succ})$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred\ t}_1 : \mathsf{Nat}} \quad (\text{T-Pred})$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{Nat}}{\Gamma \vdash \mathsf{iszero\ t}_1 : \mathsf{Bool}} \quad (\text{T-IsZero})$$

$$\Gamma \vdash \mathsf{true} : \mathsf{Bool} \quad (\text{T-True})$$

$$\Gamma \vdash \mathsf{false} : \mathsf{Bool} \quad (\text{T-False})$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{Bool} \quad \Gamma \vdash \mathsf{t}_2 : \mathsf{T} \quad \Gamma \vdash \mathsf{t}_3 : \mathsf{T}}{\Gamma \vdash \mathsf{if\ t}_1 \mathsf{\ then\ t}_2 \mathsf{\ else\ t}_3 : \mathsf{T}} \quad (\text{T-If})$$

Abbildung 1: *Typisierungsregeln*

2 Typvariablen und Substitution

Als Teil unserer Typen haben wir eine unendliche Menge von Typvariablen festgelegt. Sie können als Platzhalter für konkrete Typen (wie z.B. Nat oder Bool) verstanden werden. Haben wir nun einen Term, der Typvariablen als Typannotationen besitzt, dann stellt sich die Frage, ob wir durch Substitution dieser Variablen mit anderen Typen einen wohlgetypten Term erhalten können.

D.h. wir stellen uns Fragen wie z.B.: Können wir in dem folgenden Term, X so wählen, dass er typisierbar wird?

$$(\lambda \mathsf{x}{:}\mathsf{X}.\mathsf{x})\ 5$$

Definition 1 *Eine Typsubstitution σ ist eine endlich, partielle Funktion, die Typvariablen auf Typen abbildet. Somit gilt: $\sigma \in \mathsf{TV} \rightharpoonup_{\mathsf{fin}} \mathsf{T}$. Mit dom($\sigma$) bezeichnen wir den Definitionsbereich und mit range(σ) den Wertebereich der Substitution.*

Beispiel 1 *Sei $\sigma_1 = [\mathsf{X} \mapsto \mathsf{T}, \mathsf{Y} \mapsto \mathsf{U}]$, dann gilt:*

- $dom(\sigma_1) = \{\mathsf{X}, \mathsf{Y}\}$
- $range(\sigma_1) = \{\mathsf{T}, \mathsf{U}\}$

Beispiel 2 *Sei $\sigma = [\mathsf{X} \mapsto \mathit{Bool}]$ und $\mathsf{T} = \mathsf{X} \to \mathsf{X}$ dann erhalten wir*

$$\sigma\mathsf{T} = \mathsf{Bool} \to \mathsf{Bool}$$

Die Applikation $\sigma(\mathsf{T})$ einer Substitution σ und eines Typs T definieren wir erwartungsgemäß wie folgt:

$$\begin{array}{ll} \sigma(\mathsf{X}) & = \mathsf{if}\ \mathsf{X} \mapsto \mathsf{T} \in \sigma\ \mathsf{then}\ \mathsf{T}\ \mathsf{else}\ \mathsf{X} \\ \sigma(\mathsf{Nat}) & = \mathsf{Nat} \\ \sigma(\mathsf{Bool}) & = \mathsf{Bool} \\ \sigma(\mathsf{T}_1 \to \mathsf{T}_2) & = \sigma\mathsf{T}_1 \to \sigma\mathsf{T}_2 \end{array}$$

Eine Substitution σ wird punktweise auf einen Kontext Γ angewendet.

$$\sigma(\mathsf{x}_1 : \mathsf{T}_1, ..., \mathsf{x}_n : \mathsf{T}_n) = (\mathsf{x}_1 : \sigma\mathsf{T}_1, ..., \mathsf{x}_n : \sigma\mathsf{T}_n)$$

Ähnlich wird auch eine Substitution auf einen Term appliziert.

Ein entscheidende Eigenschaft von Typsubstitution ist es, dass ein wohlgetypter Term t durch Substitution immer wohlgetypt bleibt.

Theorem 1 *(Preservation of Typing under Type Substitution) Sei σ eine beliebige Substitution und gelte $\Gamma \vdash \mathsf{t} : \mathsf{T}$, dann gilt auch: $\sigma\Gamma \vdash \sigma\mathsf{t} : \sigma\mathsf{T}$*

3 Zwei Betrachtungsweisen für Typvariablen

Haben wir nun einen Term t, der Typvariablen enthält und sei Γ der zugehörige Kontext, dann stellen sich zwei verschiedene Fragen:

1. "Sind alle Substitutionsinstanzen von t wohlgetypt?" D.h. gilt für alle σ:
   $\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$ wobei T beliebig
2. "Gibt es überhaupt wohlgetypte Substitutionsinstanzen von t ?" D.h. können wir σ finden, sodass gilt:
   $\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$ wobei T beliebig

Gemäß der ersten Sichtweise sollten Typen mithilfe von Typvariablen während der Typprüfung so abstrakt wie möglich gehalten werden. Dadurch erreicht man laut *Theorem 1*, dass ein wohlgetypter Term auch wohlgetypt bleibt, egal welche konkreten Typen man später einsetzt. Als Beispiel:

$$\lambda\mathsf{f} : \mathsf{X} \to \mathsf{X}.\ \lambda\mathsf{x} : \mathsf{X}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x})$$

hat den Typ $(\mathsf{X} \to \mathsf{X}) \to \mathsf{X} \to \mathsf{X}$, und egal durch welchen konkreten Typ T wir X ersetzen , ist auch

$$\lambda\mathsf{f} : \mathsf{T} \to \mathsf{T}.\ \lambda\mathsf{x} : \mathsf{T}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x})$$

wohlgetypt. Halten wir Typvariablen auf diese Weise abstrakt, erhalten wir *parametrischen Polymorphismus*. D.h. wir benutzen Typvariablen um auszudrücken, dass ein Term in mehreren konkreten Kontexten mit unterschiedlichen Typen verwendet werden kann. Wir kommen auf dieses Thema in Kapitel 7 zurück.

Bei der zweiten Betrachtungsweise kann der Term t noch nicht einmal wohlgetypt sein und wir wollen wissen, ob wir die Typvariablen in t so wählen können, dass er typisierbar wird. Als Beispiel:

$$\lambda\mathsf{f} : \mathsf{Y}.\ \lambda\mathsf{x} : \mathsf{X}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x})$$

ist so nicht typisierbar. Ersetzen wir jedoch Y durch $\mathsf{Nat} \to \mathsf{Nat}$ und X durch Nat, dann ist

$$\lambda\mathsf{f} : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda\mathsf{x} : \mathsf{Nat}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x})$$

vom Typ $(\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$. Wenn wir dagegen nur Y durch $\mathsf{X} \to \mathsf{X}$ ersetzen, erhalten wir den bekannten Term

$$\lambda\mathsf{f} : \mathsf{X} \to \mathsf{X}.\ \lambda\mathsf{x} : \mathsf{X}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x})$$

der wohlgetypt ist, obwohl er Typvariablen enthält. Er unterscheidet sich von dem vorherigen darin, dass er die allgemeinste Instanz von $\lambda \mathsf{f} : \mathsf{Y}.\ \lambda \mathsf{x} : \mathsf{X}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x})$ ist, da er die geringsten Forderungen an die Typvariablen stellt und trotzallem eine wohlgetypte Terminstanz liefert.

Das Suchen nach solchen Instanzen führt uns zur Idee von Typinferenz. Dabei fügt der Compiler die fehlenden Typannotationen ein, die der Programmierer ausgelassen hat. Somit wird jede fehlende Annotation durch eine frische Variable aufgefüllt. Im Extremfall können alle Typannotationen weggelassen werden und wir haben die Syntax wie im ungetypten Lambda Kalkül. Als nächstes versuchen wir durch Typinferenz, einen allgemeinste Instanz zu finden, die typisierbar ist.

Definition 2 *Sei* Γ *ein Kontext und* t *ein Term. Eine Lösung für* (Γ, t) *ist ein Paar* (σ, T)*, sodass* $\sigma\Gamma \vdash \sigma\mathsf{t} : \mathsf{T}$*.*

4 Constraintbasierte Typisierung

Wir werden nun einen Algorithmus präsentieren, der gegeben ein Term t und ein Kontext Γ eine Menge von Constraints - Gleichungen von Typausdrücken - bestimmt, die von jeder Lösung für (Γ, t) erfüllt sein müssen. Die Intuition dahinter ist im wesentlichen dieselbe wie bei Typprüfung, statt jedoch bestimmte Constraints zu prüfen, werden sie einfache für spätere Betrachtungen aufgezeichnet. So werden wir z.B. bei einer Applikation $\mathsf{t}_1\ \mathsf{t}_2$ mit $\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1$ und $\Gamma \vdash \mathsf{t}_2 : \mathsf{T}_2$ nicht prüfen ob T_1 die Form $\mathsf{T}_2 \to \mathsf{T}_{12}$ hat und T_{12} als Typ zurückgeben, sondern werden eine frische Variable wählen und $\mathsf{T}_1 = \mathsf{T}_2 \to \mathsf{X}$ als Constraint aufzeichnen und X als Typ zurückgeben.

Definition 3 *Eine* ***Constraintmenge*** C *ist eine Menge von Gleichungen* $\{\mathsf{S}_\mathsf{i} = \mathsf{T}_\mathsf{i}$ *mit* $\mathsf{i} \in 1 \ldots n\}$*. Eine Substitution* σ *erfüllt ein Constraint* $\mathsf{S} = \mathsf{T}$ *genau dann, wenn gilt:* $\sigma S = \sigma\mathsf{T}$*.*
σ erfüllt C genau dann, wenn sie jede Gleichung in C erfüllt.

Die zugehörige Constraint-Typing-Relation $\Gamma \vdash \mathsf{t} : \mathsf{T} \mid C$ ist definiert durch die Constraint Typisierungsregeln in Abbildung. 2.
$\Gamma \vdash \mathsf{t} : \mathsf{T} \mid C$ kann folgendermaßen gelesen werden "Term t hat unter dem Kontext Γ den Typ T, wenn die Constraints in C erfüllt sind. $\mathsf{FV}(\mathsf{T})$ ist die Menge aller Variablen in T.

Die Constraint Typisierungsregeln beschreiben eine Prozedur, die uns zu einem Kontext Γ und einem Term t einen Typ T und eine Constraintmenge C berechnet, sodass $\Gamma \vdash \mathsf{t} : \mathsf{T} \mid C$. Im Gegensatz zu dem normalen Typisierungsalgorithmus schlägt dieser nie fehl. Er liefert also immer ein Ergebnis. Dies liegt daran, dass er die Constraints nur aufzeichnet, jedoch nie ihre Erfüllbarkeit prüft.

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

$$\frac{\Gamma, x : \mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2 \mid C}{\Gamma \vdash \lambda x : \mathsf{T}_1 . \mathsf{t}_2 : \mathsf{T}_1 \to \mathsf{T}_2 \mid C} \quad \text{(CT-Abs)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1 \mid C_1 \quad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_2 \mid C_2 \quad C' = C_1 \cup C_2 \cup \{\mathsf{T}_1 = \mathsf{T}_2 \to X\} \quad X \text{ neu}}{\Gamma \vdash \mathsf{t}_1\, \mathsf{t}_2 : X \mid C'} \quad \text{(CT-App)}$$

$$\Gamma \vdash 0 : \mathsf{Nat} \mid \{\} \quad \text{(CT-Zero)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T} \mid C \quad C' = C \cup \{\mathsf{T} = \mathsf{Nat}\}}{\Gamma \vdash \mathsf{succ}\, \mathsf{t}_1 : \mathsf{Nat} \mid C'} \quad \text{(CT-Succ)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T} \mid C \quad C' = C \cup \{\mathsf{T} = \mathsf{Nat}\}}{\Gamma \vdash \mathsf{pred}\, \mathsf{t}_1 : \mathsf{Nat} \mid C'} \quad \text{(CT-Pred)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T} \mid C \quad C' = C \cup \{\mathsf{T} = \mathsf{Nat}\}}{\Gamma \vdash \mathsf{iszero}\, \mathsf{t}_1 : \mathsf{Bool} \mid C'} \quad \text{(CT-IsZero)}$$

$$\Gamma \vdash \mathit{true} : \mathsf{Bool} \mid \{\} \quad \text{(CT-True)}$$

$$\Gamma \vdash \mathit{false} : \mathsf{Bool} \mid \{\} \quad \text{(CT-False)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1 \mid C_1 \quad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_2 \mid C_2 \quad \Gamma \vdash \mathsf{t}_3 : \mathsf{T}_3 \mid C_3 \quad C' = C_1 \cup C_2 \cup C_3 \cup \{\mathsf{T}_1 = \mathsf{Bool}, \mathsf{T}_2 = \mathsf{T}_3\}}{\Gamma \vdash \mathsf{if}\, \mathsf{t}_1\, \mathsf{then}\, \mathsf{t}_2\, \mathsf{else}\, \mathsf{t}_3 : \mathsf{T}_2 \mid C'} \quad \text{(CT-If)}$$

Abbildung 2: *Constraint Typisierungsregeln*

Wichtig ist, dass die neu gewählten Variablen aus der (CT-App)-Regel sich von den Variablen aller Unterterme unterscheiden müssen.

Die Idee der Constraint-Typisierungs Relation ist folgende:
Wir haben einen Term t und einen Kontext Γ gegeben. Um nun zu prüfen, ob t unter Γ typisierbar ist, sammeln wir erst die Constraints C, die erfüllt sein müssen. S bezeichne dabei den Typ, den t dann hätte. Wir suchen nun nach einer Substitution σ, die C erfüllt. Haben wir eine solche Substitution gefunden, so bildet $(\sigma, \sigma\mathsf{S})$ eine mögliche Lösung für unser Typisierungsproblem.
Falls keine Substitution existiert, die C erfüllt, wissen wir, dass es keine typisierbare Instanz von t gibt.

Als Beispiel nehmen wir den Term $\mathsf{t} = \lambda \mathsf{x} : \mathsf{X} \to \mathsf{Y}.\, \mathsf{x}\, 0$ unter dem leeren Kontext. Unser Algorithmus würde uns nun Constraint $\{\mathsf{Nat} \to \mathsf{Z} = \mathsf{X} \to \mathsf{Y}\}$ und den zugehörigen Typ $(\mathsf{X} \to \mathsf{Y}) \to \mathsf{Z}$ liefern.

Die Substitution $\sigma = [\mathsf{X} \mapsto \mathsf{Nat}, \mathsf{Z} \mapsto \mathsf{Bool}, \mathsf{Y} \mapsto \mathsf{Bool}]$ erfüllt unser Constraint , also wäre $\sigma((\mathsf{X} \to \mathsf{Y}) \to \mathsf{Z})$ bzw. $(\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}$ ein möglicher Typ für t.

Wir können unsere Idee durch folgende Definition formalisieren.

Definition 4 *Gegeben sei $\Gamma \vdash t : S \mid C$. Eine Lösung für (Γ, t, S, C) ist nun ein Paar (σ, T), sodass gilt: σ erfüllt C und $\sigma S = T$*

Wir haben nun zwei Möglichkeiten Lösungen für ein Typproblem zu finden:

1. [Deklarativ] als die Menge aller Lösungen im Sinne von *Definition 2.*
2. [Algorithmisch] mithilfe unserer Constraint-Typing Relation, indem wir erst S und C bestimmen und die Lösungsmenge für (Γ, t, S, C) nehmen gemäß *Definition 3.*

Wir müssen in dem Zusammenhang zeigen, dass diese beiden Verfahren äquivalent sind. D.h. dass jede Lösung für (Γ, t, S, C) auch eine Lösung für (Γ, t) ist und jede Lösung für (Γ, t) so erweitert werden kann, dass sie auch eine Lösung für (Γ, t, S, C) ist. Wir werden hier nur die beiden Theoreme präsentieren. Die Beweise können im Buch von B.C.Pierce *Types and Programming Languages* nachgeschlagen werden. (S.323-325)

Theorem 2 (Soundness of Constraint Typing) *Gegeben sei* $\Gamma \vdash t : S \mid C$. *Wenn* (σ, T) *eine Lösung für* (Γ, t, S, C) *ist, dann ist sie auch Lösung für* (Γ, t)

Theorem 3 (Completeness of Constraint Typing) *Gegeben sei* $\Gamma \vdash t : S \mid C$. *Wenn* (σ, T) *eine Lösung für* (Γ, t), *dann gibt es auch eine Lösung* (σ', T) *für* (Γ, t, S, C), *sodass* $\sigma'|_{\mathrm{dom}(\Gamma)} = \sigma$

Beispiel 3 *Wenden wir unseren Algorithmus auf den Term* $\mathsf{t} = \mathsf{if\ x\ then\ y\ else}\ 0$ *an,*

$$\dfrac{\dfrac{\dfrac{(\mathsf{y}:\mathsf{Y}) \in \Gamma}{(\mathsf{x}:\mathsf{X}, \mathsf{y}:\mathsf{Y}) \vdash \mathsf{y} : \mathsf{Y} \mid \emptyset} \quad \dfrac{(\mathsf{x}:\mathsf{X}) \in \Gamma}{(\mathsf{x}:\mathsf{X}, \mathsf{y}:\mathsf{Y}) \vdash \mathsf{x} : \mathsf{X} \mid \emptyset} \quad \dfrac{}{(\mathsf{x}:\mathsf{X}, \mathsf{y}:\mathsf{Y}) \vdash 0 : \mathsf{Nat} \mid \emptyset}\ C = \{\mathsf{X} = \mathsf{Bool}, \mathsf{Y} = \mathsf{Nat}\}}{(\mathsf{x}:\mathsf{X}, \mathsf{y}:\mathsf{Y}) \vdash \mathsf{if\ x\ then\ y\ else}\ 0 : \mathsf{Nat} \mid \{\mathsf{X} = \mathsf{Bool}, \mathsf{Y} = \mathsf{Nat}\}}\,(\text{CT-If})}{(\mathsf{x}:\mathsf{X}) \vdash \lambda\,\mathsf{y}:\mathsf{Y}.\ \mathsf{if\ x\ then\ y\ else}\ 0 : \mathsf{Y} \to \mathsf{Nat} \mid \{\mathsf{X} = \mathsf{Bool}, \mathsf{Y} = \mathsf{Nat}\}}\,(\text{CT-Abs})}{\emptyset \vdash \lambda\,\mathsf{x}:\mathsf{X}.\ \lambda\,\mathsf{y}:\mathsf{Y}.\ \mathsf{if\ x\ then\ y\ else}\ 0 : \mathsf{X} \to \mathsf{Y} \to \mathsf{Nat} \mid \{\mathsf{X} = \mathsf{Bool}, \mathsf{Y} = \mathsf{Nat}\}}\,(\text{CT-Abs})$$

so erhalten wir $C = \{\mathsf{X} = \mathsf{Bool}, \mathsf{Y} = \mathsf{Nat}\}$ *und* $\mathsf{S} = \mathsf{X} \to \mathsf{Y} \to \mathsf{Nat}$.

5 Unifikation

$$
\begin{array}{lcl}
\text{unify}(C) & = & \text{if } C = \emptyset \text{ then } [\,] \\
& & \text{else let } \{S = T\} \cup C' = C \text{ in} \\
& & \quad \text{if } S = T \\
& & \quad\quad \text{then unify}(C') \\
& & \quad \text{else if } S = X \text{ and } X \notin FV(T) \\
& & \quad\quad \text{then unify}([X \mapsto T]C') \circ [X \mapsto T] \\
& & \quad \text{else if } T = X \text{ and } X \notin FV(S) \\
& & \quad\quad \text{then unify}([X \mapsto S]C') \circ [X \mapsto S] \\
& & \quad \text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2 \\
& & \quad\quad \text{then unify}(C' \cup \{S_1 = T_1, S_2 = T_2\}) \\
& & \quad \text{else} \\
& & \quad\quad \text{fail}
\end{array}
$$

Abbildung 3: *Unifikation*

Um nun aus einer betsimmten Constraintmenge eine Lösung zu bestimmen, folgen wir der Idee von Hindley (1969) und Milner (1978). Wir beutzen Unifikation (Robinson, 1971), um zu ermitteln, ob die Menge der Lösungen nicht leer ist, und wenn ja um eine beste Lösung zu erhalten. D.h. dass sich jede Lösung aus ihr herleiten lässt.

Definition 5 *Ein Substitution σ ist weniger spezifisch (oder allgemeiner) als eine Substitution σ', geschrieben $\sigma \sqsubseteq \sigma'$, wenn $\sigma' = \gamma \circ \sigma$ für eine beliebige Substitution γ.*

Definition 6 *Ein allgemeinste Substitution für eine Constraintmenge C ist eine Substitution σ, die C erfüllt und für alle Substitutionen σ', die ebenfalls C erfüllen, $\sigma \sqsubseteq \sigma'$ gilt.*

Definition 7 *Der Unifikationsalgorithmus für Typen ist in Abbildung. 3 definiert. Die Zeile "let $\{S = T\} \cup C' = C$" sollte als "wähle ein Constraint $\mathsf{S} = \mathsf{T}$ aus der Menge C und sei C' der Rest des Constraintmenge" gelesen werden.*

Die Prüfung $\mathsf{X} \notin \mathsf{FV}(\mathsf{T})$ soll verhindern, dass zyklische Substitutionen wie z.B. $\mathsf{X} \mapsto \mathsf{X} \to \mathsf{X}$ gebaut werden. In Typsystemen mit unendlichen Typen (wie z.B. rekursiven Typen) macht ein solcher Test natürlich keinen Sinn und kann weggelassen werden.

Theorem 4 *Zu bemerken ist ,dass Unifikation immer terminiert. Der Algorithmus wirft einen Fehler, wenn die gegebene Constraintmenge nicht lösbar ist und ansonsten liefert er eine allgemeinste Substitution.*

1. $\mathsf{unify}(\mathsf{C})$ *hält entweder mit einem Fehler oder liefert eine Substitution.*
2. *Sei* $\mathsf{unify}(\mathsf{C}) = \sigma$*, dann erfüllt* σ *die Constraintmenge* C
3. *Sei* γ *eine weiter Substitution, die* C *erfüllt, dann gilt für* $\mathsf{unify}(\mathsf{C}) = \sigma$*, dass* $\sigma \sqsubseteq \gamma$

Der Beweis kann ebenfalls in Pierce's Buch (S.328-329) nachgelesen werden.

6 Allgemeinste Typen

Wie wir oben schon angemerkt haben, können wir sobald es eine Lösung gibt eine allgemeinste Lösung finden.

Definition 8 *Ein allgemeinste Lösung für* (Γ, t, S, C) *ist eine Lösung* $(\sigma, , \mathsf{T})$*, sodass für jede weiter Lösung* (σ', T')*,* $\sigma \sqsubseteq \sigma'$ *gilt. Wenn* (σ, T) *eine allgemeinste Lösung ist, dann nennen wir* T *den allgemeinsten Typ von* t *unter* Γ

Theorem 5 (Principal Types) *Wenn* (Γ, t, S, C) *eine Lösung hat, dann gibt es auch ein allgemeinste Lösung. Der Unifikations Algorithmus in Abbildung. 3 kann dazu genutzt werden, eine allgemeinste Lösung zu bestimmen, falls es überhaupt eine Lösung gibt.*

Der Beweis hierzu folgt aus der Definition einer Lösung für (Γ, t, S, C) und den Eigenschaften der Unifikation.

Korollar 1 *Somit ist entscheidbar, ob* (Γ, t, S, C) *eine Lösung besitzt.*

7 Let Polymorphismus

Polymorphismus findet man in vielen Sprachen. Er bietet die Möglichkeit, bestimmte Teile eines Programms auf verschiedenen Typen zu verwenden. Unser vorgestellter Algorithmus kann so verallgemeinert werden, dass er eine einfache Art des Polymorphismus zur Verfügung stellen kann, den sogenannte Let-Polymorphismus.
Die Motivation hierzu folgte aus Beispielen wie diesem.
Wir definieren eine einfache Funktion *double*, die ihr erstes Argument zweimal auf ihr zweites anwendet:

$$\begin{array}{l}\mathsf{let}\ \mathit{double} = \lambda\,\mathsf{f} : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda\mathsf{a} : \mathsf{Nat}.\ \mathsf{f}(\mathsf{f}\ (\mathsf{a}))\ \mathsf{in} \\ \mathit{double}\ (\lambda\mathsf{x} : \mathsf{Nat}.\ \mathsf{succ}\ (\mathsf{succ}\ \mathsf{x}))\ 2;\end{array}$$

Weil wir double auf eine Funtion vom Typ $\mathsf{Nat} \to \mathsf{Nat}$ anwenden wollen, wählen wir Typannotationen, die uns für *double* den Typ $(\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$ geben.Alternativ können wir *double* aber auch auf eine boolsche Funktion anwenden:

$$\begin{array}{l}\mathsf{let}\ \mathit{double} = \lambda\,\mathsf{f} : \mathsf{Bool} \to \mathsf{Bool}.\ \lambda\mathsf{a} : \mathsf{Bool}.\ \mathsf{f}(\mathsf{f}\ (\mathsf{a}))\ \mathsf{in} \\ \mathit{double}\ (\lambda\mathsf{x} : \mathsf{Bool}.\ \mathsf{x})\ \mathsf{false};\end{array}$$

Was wir nicht können, ist *double* in beiden Versionen in demselben Programm verwenden. Auch die Annotation mit einer Typvariablen bringt keinen Erfolg, denn der folgende Ausdruck

$$\begin{array}{l}\mathsf{let}\ \mathit{double} = \lambda\,\mathsf{f} : \mathsf{X} \to \mathsf{X}.\ \lambda\mathsf{a} : \mathsf{X}.\ \mathsf{f}(\mathsf{f}\ (\mathsf{a}))\ \mathsf{in} \\ \mathsf{let}\ \mathsf{a} = \mathit{double}\ (\lambda\mathsf{x} : \mathsf{Nat}.\ \mathsf{succ}\ (\mathsf{succ}\ \mathsf{x}))\ 1\ \mathsf{in} \\ \mathsf{let}\ \mathsf{b} = \mathit{double}\ (\lambda\mathsf{x} : \mathsf{Bool}.\ \mathsf{x})\ \mathsf{false}\ \mathsf{in}\ \ldots\end{array}$$

würde uns ein Constraint liefern, das sowohl $\mathsf{X} \to \mathsf{X} = \mathsf{Nat} \to \mathsf{Nat}$ als auch $\mathsf{X} \to \mathsf{X} = \mathsf{Bool} \to \mathsf{Bool}$ verlangt, was natürlich nicht erfüllbar ist.

Das Problem, das sich hier stellt, ist, dass für beide Teile dieselbe Variable X verwendet wird. Dies können wir jedoch leicht beheben, indem in einem ersten Schritt eine neue Regel

$$\frac{X\ \mathsf{neu} \qquad \Gamma, x : X \vdash \mathsf{t_1} : \mathsf{T} \mid C}{\Gamma \vdash \lambda\, x.\, \mathsf{t_1}\ : X \to \mathsf{T} \mid C} \quad (\text{CT-AbsInf})$$

eingeführt wird. Sie erlaubt es Typannotationen vorerst auszulassen und dann bei der Typprüfung mit einer frischen Variablen X zu binden.

Als nächstes ändern wir die normale (T-Let)-Regel,

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1 \qquad \Gamma, x : \mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let}\ x = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 : \mathsf{T}_2} \quad \text{(T-Let)}$$

die wir hier benutzt haben, wie folgt um.

$$\frac{\Gamma \vdash [x \mapsto \mathsf{t}_1]\mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let}\ x = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 : \mathsf{T}_2} \quad \text{(T-LetPoly)}$$

Zusätlich führen wir eine entsprechende Constraint-Regel ein

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1 \,|\, C_1 \qquad \Gamma, x : \mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2 \,|\, C_2 \qquad C = C_1 \cup C_2}{\Gamma \vdash \mathsf{let}\ x = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 : \mathsf{T}_2 \,|\, C} \quad \text{(CT-Let)}$$

Somit wird nun zuerst ein Reduktionsschritt

$$\mathsf{let}\ \mathsf{x} = \mathsf{v}_1\ \mathsf{in}\ \mathsf{t}_2 \rightarrow [\mathsf{x} \mapsto \mathsf{v}_1]\mathsf{t}_2 \quad \text{(E-LetV)}$$

vollzogen, bevor wir Typen bestimmen.

Mithilfe dieser Regeln kommen wir unserem gewünschten Ziel schon ein großes Stück näher, denn

$$\begin{array}{l}\mathsf{let}\ \mathit{double} = \lambda\,\mathsf{f}\,.\,\lambda\mathsf{a}:.\,\mathsf{f}(\mathsf{f}\ (\mathsf{a}))\ \mathsf{in}\\ \mathsf{let}\ \mathsf{a} = \mathit{double}\ (\lambda\mathsf{x}:\mathsf{Nat}.\,\mathsf{succ}\ (\mathsf{succ}\ \mathsf{x}))\ 1\ \mathsf{in}\\ \mathsf{let}\ \mathsf{b} = \mathit{double}\ (\lambda\mathsf{x}:\mathsf{Bool}.\,\mathsf{x})\ \mathsf{false}\ \mathsf{in}\ \ldots\end{array}$$

ist nun mit der (CT-AbsInf) und der (CT-LetPoly)-Regel typisierbar. (CT-LetPoly) erstellt dabei zwei Kopien der Funktion *double* und (CT-AbsInf) wählt danach zwei voneinander verschiedene Typvariablen.

Ein Problem bleibt jedoch noch. Folgender Term wäre typisierbar

$$\mathsf{let}\ x = <\text{utter garbage}>\ \mathsf{in}\ 5$$

Dies können wir durch eine kleine Änderung in der (T-LetPoly)-Regel beheben.

$$\frac{\Gamma \vdash [x \mapsto \mathsf{t}_1]\mathsf{t}_2 : \mathsf{T}_2 \quad \Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{let}\ x = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 : \mathsf{T}_2} \quad \text{(T-LetPoly)}$$

D.h. es wird vor der Reduktion noch geprüft ob t_1 typisierbar ist. Die gleiche Änderung nehmen wir in der (CT-LetPoly)-Regel vor.

Ein weiteres Problem, das sich bei dem bisherigen Algorithmus bemerkbar macht, ist die exponentielle Laufzeit. Wenn z.B. der Körper des Let-Ausdrucks viele Auftreten der durch let gebundenen Variable enthält, dann wird die rechte Seite der Let-Definition bei jedem Auftreten geprüft. Da die rechte Seite wiederum Let-Ausdrücke enthalten kann, verursacht dies in betsimmten Fällen eine Menge Arbeit, die exponentiell zur eigentlichen Grösse des Terms werden kann. Dies kann man vermeiden indem man eine bessere Vorgehensweise wählt. Wir prüfen einen Term $\mathsf{let}\ x = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2$ unter dem Kontext Γ dann wie flogt:

1. Wir benutzen die Constraint Typisierungsregeln, um einen Typ S_1 und eine Constraintmenge C_1 für den Term t_1 zu bestimmen.
2. Mithilfe der Unifikation bestimmen wir eine allgemeinste Lösung σ für C_1 und wenden σ auf S_1 (und Γ) an um den allgemeinsten Typen T_1 von t_1 zu erhalten.
3. Wir quantifizieren jede freie, verbleibende Variable in T_1, die nicht im Kontext gebunden ist. Wenn z.B. $\mathsf{X}_1 \ldots \mathsf{X}_n$ in T_1 frei vorkommen, schreiben wir $\forall \mathsf{X}_1 \ldots \mathsf{X}_n.\mathsf{T}_1$ für das allgemeinste Typschema für t_1.
4. Wir erweitern den Kontext um $(\mathsf{x} : \forall \mathsf{X}_1 \ldots \mathsf{X}_n.\mathsf{T}_1)$ und beginnen dann mit der Typprüfung für t_2
5. Bei jedem Auftreten von x erzeugen wir für $\forall \mathsf{X}_1 \ldots \mathsf{X}_n.\mathsf{T}_1$ neue Variablen $\mathsf{Y}_1 \ldots \mathsf{Y}_n$ und benutzen sie um eine Instanz $[\mathsf{X}_1 \mapsto \mathsf{Y}_1, \ldots \mathsf{X}_n \mapsto \mathsf{Y}_n]\mathsf{T}_1$ des Typschemas zu erhalten.

Dieser Algorithmus ist wesentlich effizienter als die einfache Variante und es hat sich rausgestellt, dass er in der Praxis nahezu linear ist. Deshalb war es um so überraschender, als Kfoury, Tiuryn und Urzyczyn (1990) unabhängig von Mairson (1990) zeigten, dass er eine im "worst-case" expontielle Laufzeit hat. Das Beispiel, das sie brachteten, sah wie folgt aus:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{f}_0 = \mathsf{fun}\ \mathsf{x} \rightarrow (\mathsf{x}, \mathsf{x})\ \mathsf{in} \\
\mathsf{let}\ \mathsf{f}_1 = \mathsf{fun}\ \mathsf{y} \rightarrow \mathsf{f}_0(\mathsf{f}_0\ \mathsf{y})\ \mathsf{in} \\
\mathsf{let}\ \mathsf{f}_2 = \mathsf{fun}\ \mathsf{y} \rightarrow \mathsf{f}_1(\mathsf{f}_1\ \mathsf{y})\ \mathsf{in} \\
\mathsf{let}\ \mathsf{f}_3 = \mathsf{fun}\ \mathsf{y} \rightarrow \mathsf{f}_2(\mathsf{f}_2\ \mathsf{y})\ \mathsf{in} \\
\mathsf{let}\ \mathsf{f}_4 = \mathsf{fun}\ \mathsf{y} \rightarrow \mathsf{f}_3(\mathsf{f}_3\ \mathsf{y})\ \mathsf{in} \\
\mathsf{let}\ \mathsf{f}_5 = \mathsf{fun}\ \mathsf{y} \rightarrow \mathsf{f}_4(\mathsf{f}_4\ \mathsf{y})\ \mathsf{in} \\
\mathsf{f}_5(\mathsf{fun}\ \mathsf{z} \rightarrow \mathsf{z})
\end{array}
$$

Zum Abschluss noch ein letztes erwähnenswertes Problem. Lassen wir Features zu, wie z.B. Referenzen, dann müssen wir auf die Seiteneffekte achtgeben. Ein Beispiel:

$$
\begin{array}{l}
\mathsf{let}\ \mathsf{r} = \mathsf{ref}(\lambda \mathsf{x}.\ \mathsf{x})\ \mathsf{in} \\
(\mathsf{r} := (\lambda \mathsf{x} : \mathsf{Nat}.\ \mathsf{succ}\ \mathsf{x}); \\
(!\mathsf{r})\ \mathsf{true}
\end{array}
$$

Dieser Term wäre typisierbar, da wir für beide Auftreten von r jeweils verschiedene Typvariablen annotieren würden. Dies können wir verhindern, indem wir fordern, dass ein Let-Binding nur dann polymorph sein kann, also dass wir die freien Variablen quantifizieren dürfen, wenn die rechte Seite einen syntaktischen Wert darstellt. D.h. wenn wir den Typ von r dem Kontext hinzufügen wählen wir den Typ $\mathsf{X} \rightarrow \mathsf{X}$ statt $\forall \mathsf{X}.\mathsf{X} \rightarrow \mathsf{X}$. Somit würde dieser Term ordnungsgemäß als nicht typisierbar gelten.

8 HM(X) Grundidee

Das Hindley/Milner System ist eines der populärsten und meist studiertesten Typsysteme. Zwei der Hauptstärken des Systems sind die *Typ Soundness* Eigenschaft und die Existenz eines Typinferenzalgorithmus. Es sind nun viele Erweiterungen dieses Hindley-/Milner Systems erschienen. Viele dieser Erweiterungen haben mit Constraints zu tun, wie z.B. Record Systeme, Overloading und Subtyping, um nur ein paar zu nennen. Erweiterungen des Hindley/Milner Systems gewinnen auch in der Programmanalyse an Popularität.
Obwohl diese Typsysteme verschieden Constraintdomains nutzen, sind sie in ihrem typtheoretischen Aspekt sehr ähnlich. Deshalb entwickelten Martin Odersky, Martin Sulzmann und Martin Wehr den "general framework HM(X)" für Typsysteme mit Constraints. Konkrete Typsysteme können dabei über den Parameter X instanziiert werden. HM(X) bleibt in der Tradition von Hindley/Milner. So erfüllen die Typsysteminstanzen von HM(X) , unter bestimmten Voraussetzungen für X, die *Typ Soundness* Eigenschaft und HM(X) stellt einen generischen Typinferenzalgorithmus zur Verfügung, der ebenfalls unter bestimmten Bedingungen für X, immer den allgemeinsten Typ eines Term berechnet.

9 Geschichte

Der Begriff "allgemeinster Typ" bzw. "principal types" im Zusammenhang mit dem Lambda Kalkül, lässt sich rückverfolgen bis mindestens zur Arbeit von Curry in den 50ern (Curry und Feys, 1958). Ein Algorithmus der auf Currys Ideen basiert und allgemeinste Typen bestimmt, wurde von Hindley (1969) geschaffen. Ähnliche Algorithmen haben unabhängig davon auch Morris (1968) und Milner (1978) präsentiert. In der Welt der propositionalen Logik geht der Begriff sogar noch weiter zurück, zu Tarski in den 20ern und den Merediths Cousins in den 50ern (Lemmon, Meridith, Meridith, Prior und Thomas 1957). Eine erste Implementierung auf einem Computer war dann 1957 durch David Meridith. Weiter historische Anmerkungen können in Pierce's Buch (S.336-338) gefunden werden.

10 Referenzen

[1.] Benjamin C. Pierce *Types and Programming Languages* MIT Press 2002

[2.] M. Odersky, M. Sulzmann and M. Wehr,
Type inference with constrained types.
Theory and Practice of Object Systems, 5(1), 1999