Let-Polymorphismus

Simon Roßkopf

14. Dezember 2016

Zusammenfassung

Es wird eine Erweiterung des Lambda Kalküls und ein zugehöriges Typsystem betrachtet, welche lokale polymorphe Definitionen erlauben. Es werden sowohl Typisierungsregeln als auch ein Inferenzalgorithmus gegeben. Außerdem werden Aussagen über Korrektheit, Vollständigkeit und Komplexität des Algorithmus zusammengefasst.

1 Erweiterter Lambda Kalkül

Die folgenden Kapitel 1 bis 6 basieren auf einer Veröffentlichung von Damas und Millner [2]. Als unterliegende Sprache wird ein mit *let-Ausdrücken* erweiterter Lambda Kalkül verwendet. Ausdrücke in diesem können die folgende Form haben, wobei x aus einer Menge von Bezeichnern für Variablen und Konstanten stammt:

$$t ::= x \mid t\ t' \mid \lambda x.t \mid \text{let } x = t \text{ in } t'$$

Let-Ausdrücke erlauben es *lokale* Definitionen vorzunehmen. Der Term t wird hierbei an den Namen x gebunden, welcher in t' verwendet werden kann. Bei der Auswertung wird ein let-Ausdruck der obigen Form zu $(\lambda x.t')\, t$, die Bindung an einen Namen wird nun von der Abstraktion übernommen. Ohne Beachtung der Typen können let-Ausdrücke also als syntaktischer Zucker gesehen werden.

Dies suggeriert bereits einen naiven Ansatz für die Typinferenz. Durch Umwandlung aller vorkommenden let-Ausdrücke in Abstraktionen erhält man einen Ausdruck des Lambda Kalküls, für welchen man einen Typ mit den bekannten Methoden inferieren kann.

Dieser Ansatz kann jedoch für viele eigentlich sinnvolle Ausdrücke keinen Typ inferieren. Ein Beispiel dafür ist unter anderem let $I = \lambda x.x$ in $I\ I$. Entfernen des let-Ausdrucks liefert $(\lambda I.I\ I)\ (\lambda x.x)$. Für diesen Ausdruck lässt sich kein Typ inferieren, da für beide I aufgrund der vorkommenden Selbstanwendung ein je unterschiedlicher Typ berechnet werden müsste. Unabhängig davon gelingt die Auswertung zu $(\lambda x.x)$ problemlos.

Der im folgenden dargestellte Ansatz erlaubt es einen Typ für das Beispiel anzugeben. Hierfür ist es notwendig, dass mit let-Ausdrücken vorgenommene Definitionen *polymorph* sind, dass heißt sie können mit unterschiedlichen Typen verwendet werden.

Das Beispiel enthält zwar die Selbstanwendung $I\ I$, dies ist jedoch ein Spezialfall. Das vorgestellte Typsystem ermöglicht keine Typisierung von Selbstanwendungen im Allgemeinen. In ähnlicher Weise erlaubt es keine allgemeine Rekursion. Wie auch schon beim einfach getypten Lambda

Kalkül, kann diese aber durch einen expliziten Fixpunkt Kombinator hinzugefügt werden.

2 Typen und Typschemata

Die Menge der möglichen Typen eines Terms unterscheidet sich nicht von der für den einfach getypten Lambda Kalkül genutzten. Typen bestehen aus primitiven Basistypen ι (z.B. Int, Bool, ...) und Typvariablen α. Zusätzlich ist es möglich aus diesen kompliziertere Funktionstypen zu erzeugen.

$$\tau ::= \iota \mid \alpha \mid \tau \to \tau'$$

Im folgenden werden Typen mit τ_i und Typvariablen mit $\alpha_i, \beta_i, ...$ bezeichnet Wenn ein Term t den Typ τ hat schreib man $t : \tau$.

Diese Typen reichen jedoch nicht aus um den gewünschten durch let-Ausdrücke in Termen vorkommenden Polymorphismus darzustellen. Hierfür ist es nötig gewisse Typvariablen explizit als beliebig zu markieren. Die im folgenden definierten Typschemata erlauben genau dies. Sie bestehen aus einem Typ, in welchem manche Typvariablen allquantifiziert sind. Dies können keine, einige oder auch alle vorkommenden Variablen sein. Insbesondere ist deshalb jeder Typ auch ein Typschema.

$$\sigma ::= \tau \mid \forall \alpha.\sigma$$

Genau wie bei Typen schreibt man t : σ wenn σ ein zu t passendes Typschema ist. Die Menge aller freien Typvariablen in σ wird mit $FV(\sigma)$ bezeichnet. Dies sind alle Typvariablen die nicht durch einen Allquantor gebunden sind.

3 Substitutuionen

Eine *Substitution* auf Typen ist eine Funktion, welche Typvariablen durch Typen ersetzt. Man schreibt $[\tau_i/\alpha_1, ..., \tau_n/\alpha_n]$ für die Substitution welche α_i durch τ_i ersetzt. Da Substitutionen Funktionen sind, wird ihre Anwendung auf einen Typ durch einfaches Voranstellen ausgedrückt. So wertet zum Beispiel $[int/\alpha_1, \alpha_3 \to \alpha_4/\alpha_2](\alpha_1 \to \alpha_2)$ zu $int \to \alpha_3 \to \alpha_4$ aus.

Das Konzept der Substitutionen lässt sich von Typen auf Typschemata übertragen. Bei Anwendung einer Substitution auf ein Typschema ersetzt man nur die freien Variablen und lässt die gebundenen unberührt. Wenn man das vorherige Beispiel leicht abändert, liefert $[int/\alpha_1, \alpha_3 \to \alpha_4/\alpha_2](\forall\alpha_2.\alpha_1 \to \alpha_2)$ das Typschema $\forall\alpha_2.int \to \alpha_2$.

Wie bei Substitutionen in prädikatenlogischen Formeln ist auch bei Typschemata Vorsicht geboten, wenn freie Typvariablen eingesetzt werden, da diese nun ungewollt durch einen Quantor gebunden sein können. Als Lösung bietet sich an, wenn notwendig, die gebundenen Typvariablen vorher implizit umzubenennen.

Ein Typschema σ' heißt eine *Instanz* eines anderen Typschemas σ wenn es eine Substitution S gibt, sodass $\sigma' = S\sigma$.

Im Gegensatz dazu hat ein Typschema $\sigma = \forall\alpha_1...\alpha_m.\tau$ eine *generische Instanz* $\sigma' = \forall\beta_1...\beta_n.\tau'$, geschrieben $\sigma \succeq \sigma'$, wenn $\tau' = [\tau_i/\alpha_1...\tau_n/\alpha_n]\tau$

für Typen $\tau_1, ..., \tau_m$ und die $\beta_1, ..., \beta_n$ nicht frei in σ vorkommen. Zu beachten ist hierbei, dass die Anzahl der α und β unterschiedlich sein kann. Es ist also möglich generische Variablen zu entfernen oder hinzuzufügen. So sind zum Beispiel $\forall\alpha_2.\alpha_1 \to int$, $\forall\alpha_3.\alpha_1 \to \alpha_3$ und $\forall\alpha_3\forall\alpha_4.\alpha_1 \to \alpha_3 \to \alpha_4$ alle generische Instanzen von $\forall\alpha_2.\alpha_1 \to \alpha_2$, da jeweils die Substitutionen $[int/\alpha_2]$, $[\alpha_3/\alpha_2]$ und $[\alpha_3 \to \alpha_4/\alpha_2]$ existieren.

Kurz, in einer Instanz wurde also in freie Variablen eingesetzt, in einer generischen Instanz in gebundene Variablen.

4 Kontext

Im folgenden ist ein *Kontext* Γ eine Menge von Aussagen der Form $x : \sigma$. Ein Kontext enthält zusätzlich zu jedem möglichem Bezeichner höchstens eine Aussage. Deshalb muss, wenn eine neue Aussage gespeichert werden soll, die eventuell bereits enthaltene zuerst entfernt werden. Soll aus einem Kontext Γ die Aussage über Bezeichner x entfernt werden, so schreibt man Γ_x. Neue Elemente können über die normale Mengenvereinigung hinzugefügt werden. Das Konzept der Substitutionen aus dem vorherigen Abschnitt lässt sich auf Kontexte übertragen. Wenn S eine Substitution und Γ ein Kontext ist bezeichnet SΓ den Kontext in welchem die Substitution auf alle Typschemata in den enthaltenen Aussagen angewandt wurde, das heißt die Menge $\{x : S\sigma | x : \sigma \in \Gamma\}$.

5 Inferenzregeln

Die Inferenzregel zur Ableitung eines Typschemas sind in Abbildung 1 gegeben. Man schreibt $\Gamma \vdash t : \sigma$ wenn sich mit diesen Regeln, unter den Annahmen im Kontext Γ, das Typschema σ für den Term t herleiten lässt. Hierbei entsprechen die ersten drei Regeln Taut, Comb und Abs genau den Regeln für den einfach getypten Lambda Kalkül. Insbesondere treten in ihnen also noch keine Typschemata auf.

Die Regeln Inst erlaubt es anstelle eines speziellen Typschemas σ' ein allgemeineres σ zu berechnen. Dabei muss σ' eine generische Instanz von σ sein.

Die Gen Regel findet man in ähnlicher Form beispielsweise auch in der Prädikatenlogik. Wenn für einen Term ein Typschema berechnet wurde erlaubt sie in diesem freie Variablen in generische umzuwandeln. Dies ist aber nur möglich wenn sie wirklich beliebig sind, also keine Informationen über sie im Kontext vorliegen.

Nach der Let Regel hat ein let-statement das selbe Typschema wie sein Körper. Hierfür wird aber ein Typschema für den lokal gebundenen Bezeichner benötigt, weshalb auch für den gebundenen Term ein Schema berechnet werden muss.

$$\text{Taut: } \frac{}{\Gamma \vdash x : \sigma}\ (x : \sigma \in \Gamma) \qquad \text{Inst: } \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \sigma'}\ (\sigma \succeq \sigma')$$

$$\text{Comb: } \frac{\Gamma \vdash t : \tau' \to \tau \qquad \Gamma \vdash t' : \tau'}{\Gamma \vdash t\ t' : \tau} \qquad \text{Gen: } \frac{\Gamma \vdash x : \sigma}{\Gamma \vdash t : \forall\alpha.\sigma}\ (\alpha \text{ nicht frei in } \Gamma)$$

$$\text{Abs: } \frac{\Gamma_x \cup \{x : \tau'\} \vdash t : \tau}{\Gamma \vdash \lambda x.t : \tau' \to \tau} \qquad \text{Let: } \frac{\Gamma \vdash t : \sigma \qquad \Gamma_x \cup \{x : \sigma\} \vdash t' : \tau}{\Gamma \vdash \text{let } x = t \text{ in } t' : \tau}$$

Abbildung 1: Inferenzregeln

Die Anwendung dieser Regeln wird in Abbildung 2 gezeigt. Hierbei wird mithilfe der Let Regel zuerst ein allgemeines Typschema für i gewählt. Die Wahl wird im linken Teilbaum unter Verwendung von Gen für einen beliebigen Typ gerechtfertigt. Im rechten Teilbaum wird für die beiden Auftreten von I je ein unterschiedlicher Typ verlangt. Da jedoch beide generische Instanzen des gewählten Typschemas sind, lässt sich dieses Problem beide Male mithilfe von Inst lösen.

Es ist hervorzuheben, dass sich die Regeln nicht automatisch anwenden lassen. Im Beispiel ist es bereits im ersten Schritt, der Anwendung der Let Regel, nötig ein Typschema σ geschickt zu wählen. Des weiteren können die Regeln Gen und Inst an jeder Stelle der Inferenz angewendet werden, da sie nicht wie die anderen Regeln durch die Syntax des betrachteten Terms gesteuert werden. Es ist also notwendig einen Algorithmus zu finden, welcher Inferenz ohne menschliches Zutun durchführt.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\alpha \vdash x:\alpha}\,\text{Taut}}{\vdash \lambda x.x:\alpha\to\alpha}\,\text{Abs}}{\vdash \lambda x.x:\forall\alpha.\alpha\to\alpha}\,\text{Gen} \qquad \dfrac{\dfrac{\dfrac{}{I:\forall\alpha.\alpha\to\alpha \vdash I:\forall\alpha.\alpha\to\alpha}\,\text{Taut}}{I:\forall\alpha.\alpha\to\alpha \vdash I:(\alpha\to\alpha)\to(\alpha\to\alpha)}\,\text{Inst} \qquad \dfrac{\dfrac{}{I:\forall\alpha.\alpha\to\alpha \vdash I:\forall\alpha.\alpha\to\alpha}\,\text{Taut}}{I:\forall\alpha.\alpha\to\alpha \vdash I:\alpha\to\alpha}\,\text{Inst}}{I:\forall\alpha.\alpha\to\alpha \vdash I\ I:\alpha\to\alpha}\,\text{Comb}}{\vdash \text{let } I=\lambda x.x \text{ in } I\ I:\alpha\to\alpha}\,\text{Let}$$

Abbildung 2: Beispiel

6 Algorithmus

Im folgenden soll nun ein Algorithmus $ty(\Gamma, t)$ definiert werden, welcher unter einer Menge von Annahmen einen allgemeinsten Typ τ für einen gegebenen Term berechnet. Neben dem Typ wird außerdem eine Substitution S zurückgegeben. Diese enthält Instanziierungen, welche während der Ausführung durchgeführt wurden.

Im Algorithmus ist es notwendig Typen zu unifizieren. Hierfür wird der Unifikations Algorithmus U von Robinson [4] verwendet. Dieser nimmt ein Paar von Typen und berechnet entweder eine Substitution oder schlägt fehl. Wenn $U(\tau_1, \tau_2)$ eine Substitution V zurückgibt, unifiziert diese τ_1 und τ_2, dass heißt $V\tau_1 = V\tau_2$. Wenn τ_1 und τ_2 durch eine Substitution S unifizierbar sind, findet U immer eine Substitution V, welche die beiden ebenfalls unifiziert. Des weiteren ist dabei immer $S = VR$ für eine passende Substitution R. Intuitiv setzt V also nur soviel wie unbedingt nötig ein, ist also so generell wie möglich.

Außerdem ist es nötig neue, bis jetzt ungenutzte Typvariablen zu generieren. Im folgenden wird deshalb die Existenz einer Funktion *new_variable*() angenommen, welche bei jedem Aufruf eine solche Variable generiert.

Des weiteren benötigt man eine Möglichkeit Typvariablen in einem Typ explizit als polymorph zu markieren. Für diesen Zweck wird die Funktion $gen(\Gamma, \tau) = \forall\alpha_1 ... \forall\alpha_n.\tau$ mit $\{\alpha_1, ...\alpha_n\} = FV(\tau) \setminus FV(\Gamma)$ definiert. Diese allquantifiziert alle Typvariablen in τ welche nicht frei in Γ vorkommen, über die also noch keine einschränkende Information bekannt ist [1].

Der Algorithmus durchläuft den zu typisierenden Term rekursiv. Die dabei auftretenden vier Fälle werden im folgenden als Pseudocode präsentiert und kurz erläutert.

$$
\begin{array}{l}
ty(\Gamma, x) = \\
\quad \mathtt{case}\ \ \Gamma(x)\ \ \mathtt{of} \\
\quad\quad \forall\alpha_1 ... \forall\alpha_n.\tau \Rightarrow \\
\quad\quad\quad ([new\_variable()/\alpha_1, ..., new\_variable()/\alpha_n]\tau, []) \\
\quad | \_ \Rightarrow\ \mathtt{throw\ \ Exception}
\end{array}
$$

Wenn der Algorithmus auf einen Bezeichner stößt, lässt sich dieser nicht weiter zerlegen. Die einzige verfügbare Typinformation findet sich also im Kontext, weshalb eine Exception geworfen werden muss, falls dieser keine passende enthält. Ein gefundenes Typschema muss vor Rückgabe in einen Typ umgewandelt werden. Dafür werden alle generischen Variablen mit neu generierten freien Variablen ersetzt. Insbesondere werden dadurch für unterschiedliche Auftreten eines Bezeichners unterschiedliche Typvariablen ohne Abhängigkeiten untereinander gewählt. Dieser Schritt kann als Kombination der Regeln Taut und Inst gesehen werden.

$$
\begin{array}{l}
ty(\Gamma\, , t_1\, t_2) = \\
\quad \mathtt{let}\ \ (S_1, \tau_1) = ty(\Gamma, t_1)\ \ \mathtt{in} \\
\quad \mathtt{let}\ \ (S_2, \tau_2) = ty(S_1\Gamma, t_2)\ \ \mathtt{in} \\
\quad \mathtt{let}\ \ \beta = new\_variable()\ \ \mathtt{in} \\
\quad \mathtt{let}\ \ V = U(S_2\tau_1, \tau_2 \to \beta)\ \ \mathtt{in} \\
\quad\quad (VS_2S_1, V\beta)
\end{array}
$$

Die Behandlung der Applikation folgt der Regel Comb. Zuerst werden Typen für die Funktion t_1 und das Argument t_2 berechnet. Anstelle des Pattern Matching in der Regel wird hier Unifikation benutzt um den Rückgabetyp zu erhalten.

$$
\begin{array}{l}
ty(\Gamma,\ \lambda x.t) = \\
\quad \mathtt{let}\ \ \beta = new\_variable()\ \ \mathtt{in} \\
\quad \mathtt{let}\ \ (S, \tau) = ty(\Gamma_x \cup \{x : \beta\}, t)\ \ \mathtt{in} \\
\quad\quad (S, \beta \to \tau)
\end{array}
$$

Im Falle einer Abstraktion muss wie auch in der Regel Abs der Typ des Funktionskörpers t' berechnet werden. Da die Sprache keinerlei Typannotationen enthält, können keine Annahmen über den Typ von x gemacht werden, weshalb eine neue Typvariable generiert werden muss. Zu beachten ist, dass der angenommene Typ für x nicht generisch ist, da von Abstraktionen gebundene Variablen nicht polymorph sind.

$$
\begin{array}{l}
ty(\Gamma\, ,\ \ \mathtt{let}\ \ x = t\ \ \mathtt{in}\ \ t') = \\
\quad \mathtt{let}\ \ (S_1, \tau_1) = ty(\Gamma, t)\ \ \mathtt{in} \\
\quad \mathtt{let}\ \ (S_2, \tau_2) = ty(S_1\Gamma_x \cup \{x : gen(S_1\Gamma, \tau_1)\}, t')\ \ \mathtt{in} \\
\quad\quad (S_2S_1, \tau_2)
\end{array}
$$

Der Fall für let-Ausdrücke entspricht einer Kombination der Let und Gen Regel. Zuerst wird ein Typ für t berechnet. Dieser kann aber nicht als Annahme für x in der Berechnung für t' benutzt werden, da eventuell vorkommende polymorphe Typvariablen nicht als solche markiert sind. Aus diesem Grund wird der Typ mithilfe von *gen* zuerst in ein Typschema umgewandelt.

Der Algorithmus ist korrekt im folgenden Sinne: Wenn der Algorithmus einen Typ berechnet $ty(\Gamma, t) = (S, \tau)$, dann kann dieser auch mithilfe

der Regeln abgeleitet werden: $S\Gamma \vdash t : \tau$. Durch Anwendung von gen auf das Ergebnis kann der Algorithmus auch zur Berechnung eines Typschemas $gen(S\Gamma, \tau)$ verwendet werden. Auch dieses kann mithilfe der Regeln abgeleitet werden, wofür nur einige Anwendungen der Regel Gen notwendig sind: $S\Gamma \vdash t : gen(S\Gamma, \tau)$.

Der Algorithmus ist vollständig in dem Sinne, dass er, wenn eine Ableitung der Form $\Gamma \vdash t : \sigma$ existiert, einen Typ berechnet $ty(\Gamma, t) = (S, \tau)$ berechnet. Das zugehörige Typschema $gen(S\Gamma, \tau)$ ist dabei so generell wie möglich, da $gen(S\Gamma, \tau) \succeq \sigma$ gilt.

Als Konsequenz dieser beiden Eigenschaften folgt die Entscheidbarkeit des Problems, ob ein Term typkorrekt ist.

7 Komplexität

Es existieren Ausdrücke für die der Inferenzalgorithmus exponentielle Laufzeit hat. Ein Beispiel ist der folgende Ausdruck. In diesem wird die Syntax (x, y) für ein Paar von Termen x, y benutzt. Diese ist zwar in der gegebenen Definition des erweiterten Lambda Kalküls nicht vorhanden, kann aber leicht nachgebaut werden, indem man (x, y) als Abkürzung für $\lambda z.z\, x\, y$ gesehen wird. Die beiden Komponenten können dann durch Anwendung von $\lambda xy.x$ oder $\lambda xy.y$ wieder extrahiert werden.

$$
\begin{array}{l}
\mathtt{let}\ \ x_0 = \lambda x.x\ \ \mathtt{in} \\
\quad \mathtt{let}\ \ x_1 = (x_0, x_0)\ \ \mathtt{in} \\
\quad\quad \mathtt{let}\ \ x_3 = (x_1, x_1)\ \ \mathtt{in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad\quad \mathtt{let}\ \ x_n = (x_{n-1}, x_{n-1})\ \ \mathtt{in} \\
\quad\quad\quad\quad\quad x_n
\end{array}
$$

In diesem Ausdruck verdoppelt sich mit jeder Zeile die Anzahl der benötigten Typvariablen, da die Terme in den beiden Komponenten, obwohl sie gleich sind, unabhängige Typen haben. Dementsprechend hat der berechnete Typ eine Länge in Größenordnung $\Omega(2^{cn})$.

Man kann außerdem zeigen, dass das Problem, ob ein Term typkorrekt ist, DEXPTIME-hart ist. Für den Beweis wird eine, in polynomieller Zeit laufende, Übersetzungsprozedur konstruiert, welche eine beliebige Turing Maschine mit exponentieller Laufzeit und eine Eingabe in einen Ausdruck des erweiterten Lambda Kalküls übersetzt. Dieser ist genau dann typisierbar, wenn die Turingmaschine die Eingabe akzeptieren würde. Somit können beliebige Probleme mit exponentieller Laufzeit auf eine Typinferenz reduziert werden.

Jedoch ist zu bemerken, das exponentielle Laufzeit praktisch nur in konstruierten Beispielen auftritt. In den meisten vorkommenden Fällen verhält sich die Laufzeit linear zur Eingabelänge [3].

Literatur

[1] Dominique Clement, Joelle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. Research Report RR-0529, INRIA, 1986.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, New York, NY, USA, 1982. ACM.

[3] Harry G. Mairson. Deciding ml typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 382–401, New York, NY, USA, 1990. ACM.

[4] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.