Ludwig-Maximilians-Universität

Fachbereich Informatik

Seminararbeit

im Studiengang Informatik

Thema: Typinferenzalgorithmus –
mit Kontrollflussanalyse

eingereicht von: Robert Kufner <robert-kufner@gmx.de>

eingereicht am: 11. Juli 2009

Betreuer: Herr Andreas Abel

Inhaltsverzeichnis

1 Ziel dieser Arbeit

Programme die von Menschen geschrieben werden, enthalten manchmal die sonderbarsten Ausdrücke. Falls jemand anderes diesen Quelltext lesen muss, wird er sich sicherlich bei mancher Funktion fragen, für welche Typen sie anwendbar ist. Hierbei fällt es ihm noch vergleichsweise einfach, den richtigen Typen zu raten, da er über genügend Voraussicht über den Code verfügt. Schwieriger wird es hingegen für den Compiler. Er benutzt einen Mechanismus, der als erstes einen vorläufigen Typen erstellt und diesen dann später verfeinert um zu einem Typen zu gelangen.
Im Folgenden soll nun das Grundprinzip dieses Mechanismus und seine Funktionsweise genauer erklärt werden. Zuvor muss dafür aber noch das zugrundeliegende Typsystem erklärt werden, sowie einige Symbole daraus. Später wird der Algorithmus um eine Kontrollflussanalyse erweitert. Jeder Schritt wird dabei durch ein Beispiel erläutert.

2 Begriffserklärung

Um die nachfolgende Arbeit verstehen zu können, sind zuerst einmal einige Begriffe und Symbole des zugrundeliegenden Typsystems von Nöten, die im Folgenden erklärt werden.
In dieser Arbeit wird zur Illustration eine funktionale Sprache verwendet, die den Namen FUN trägt und eine gewisse Ähnlichkeit zu z.B. SML aufweist. In dieser Sprache können wir Ausdrücke festlegen, die wir im weiteren Verlauf als e (für Expression) bezeichnen. Diese Ausdrücke haben keine festgelegte Form. Sie können sowohl etwas einfaches wie eine Konstante c oder eine Variable x bezeichnen, als aber auch etwas komplexeres wie z.B.
$fun_\pi fx = x$. Die sogenannten *program points*, $\pi \in \mathbf{Pnt}$, werden hierbei benutzt um die Funktionsabstraktionen zu bennen.
Damit der Typ einer Funktion ausgedrückt werden kann, benötigt man dafür ebenfalls ein Symbol. Deshalb wird das Symbol τ eingeführt, das Typen bezeichnet und die Werte *int*, *bool* oder $\tau_1 \rightarrow \tau_2$ annehmen kann. *Int* und *bool* sind dabei die einzigen zwei Grundtypen und es werden, wie üblich, Pfeile als Funktionstyp benutzt.
Somit kann beispielsweise den oben genannten Konstanten nun ein Typ zugewiesen werden, der τ_c geschrieben wird. So hat beispielsweise true den Typ $\tau_{true} = \text{bool}$ und 7 hat den Typ $\tau_7 = \text{int}$. Falls ein Typ gefunden wurde für eine Variable, wird das durch die Notationen der Form $x \mapsto \tau$ festgehalten, die dabei in eine Typumgebung Γ geschrieben wird. Obwohl Γ also eine Liste ist, kann sie trotzdem als eine endliche Abbildung von Variablen auf Typen bezeichnet werden. Damit in diesem Typsystem eine Kontrollflussanalyse implementiert werden kann, müssen wir unsere vorhandenen Typen um Annotationen erweitern, damit notiert werden kann, welche Funktion den Typ $\tau_1 \rightarrow \tau_2$ annehmen kann. Diese Annotationen erhalten das Symbol φ und beinhalten die Werte $\{\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset$.
Durch diese Einführung verändert sich nun aber auch der Typ. Er wird dabei zu einem annotierten Typ $\hat{\tau}$ der statt $\tau_1 \rightarrow \tau_2$ den Wert $\hat{\tau_1} \xrightarrow{\varphi} \hat{\tau_2}$ annimmt. Die Typumgebung wird entsprechend von normalen Typen auf annotierte Typen umgestellt und erhält das Symbol $\hat{\Gamma}$.

Der Übergang vom Typsystem das von Menschen ausgewertet werden kann, zu einem selbstständig ablaufenden Algorithmus, erfordert nun etwas drastischere Änderungen bei den bisherigen Typen und bringt auch gewisse Probleme mit sich.
Wie bereits erwähnt, versucht der Algorithmus einen vorläufigen Typen zu erstellen, um diesen dann später zu verfeinern. Damit der Algorithmus einen vorläufigen Typen erstellen kann, muss zunächst der einfache Typ um einen möglichen Wert erweitert werden. Neben den bisherigen Werten
int, *bool*, $\tau_1 \rightarrow \tau_2$, kann er nun auch den Wert α annehmen, welches eine Variable der Art ´a,´b,´c etc. bezeichnet. Durch diese Erweiterung, wird der normale Typ in erweiterter Typ umbenannt.
Falls der Algorithmus zu einem Ergebnis kommt, müssen natürlich diese vorläufigen Typen auch wieder verfeinert werden können. Dies geschieht über die Substitution θ, die eine endliche, partielle Abbildung von Typvariablen auf erweiterte Typen ist. Beispielsweise führt die Anwendung $\theta\ \alpha$ zu τ, falls $\theta\ \alpha = \tau$
Damit die Kontrollflussanalyse auch im Algorithmus implementiert werden kann, werden später noch sogenannte einfache Typen eingeführt, die dann sowohl Variablen, als auch Annotationsvariablen enthalten werden.
Zusammenfassend ist noch einmal der Unterschied zwischen den Typen des deklarativen Typsystems und den Typen des Algorithmus hervorzuheben. Falls es um den Teil ohne Kontrollflussanalyse geht besitzt das Typsystem sogenannte Typen, die weder Variablen, noch Annotationen besitzen. Der Algorithmus hingegen besitzt erweiterte Typen, die zwar keine Annotationen erlauben aber Variablen. Wenn die Kontrollflussanalyse dazu genommen wird, gibt es im deklarativen Typsystem annotierte Typen, die keine Variablen besitzen aber Annotationen. Auf der Seite des Algorithmus gibt es dabei sogenannte einfache Typen, die sowohl Variablen, als auch Annotationsvariablen enthalten. Dieser Unterschied ist wichtig und wird bei der Einführung der Kontrollflussanalyse im Algorithmus noch Probleme bereiten.

3 Der W_{UL} Algorithmus

3.1 Idee

Die eigentliche Idee die hinter dem W_{UL}, der auch Typrekonstruktionsalgorithmus genannt wird, steckt, ist Folgende: Er erhält eine Typumgebung Γ und einen Ausdruck e. Falls der Algorithmus erfolgreich ist, liefert er den erweiterten Typen τ des Ausdrucks und die Substitution θ, die angibt wie die Umgebungsvariable verfeinert werden muss, um einen Typ zu erhalten.

$$
\begin{aligned}
W_{UL}(\Gamma, c) &= (\tau_c, id) \\
W_{UL}(\Gamma, x) &= (\Gamma(x), id) \\
W_{UL}(\Gamma, fn_\pi x \Rightarrow e) &= \text{let } \alpha_x \text{ be fresh} \\
(\tau, \theta) &= W_{UL}(\Gamma[x \mapsto \alpha_x], e) \\
&\quad \text{in } ((\theta\alpha_x) \to \tau, \theta) \\
W_{UL}(\Gamma, fun_\pi f x \Rightarrow e_0) &= \text{let } \alpha_x, \alpha_0 \text{ be fresh} \\
(\tau_0, \theta_0) &= W_{UL}(\Gamma[f \mapsto \alpha_x \to \alpha_0][x \mapsto \alpha_x], e_0) \\
\theta_1 &= U_{UL}(\tau_0, \theta_0\alpha_0) \\
&\quad \text{in } (\theta_1(\theta_0\alpha_x) \to \theta_1\tau_0, \theta_1 \circ \theta_0)
\end{aligned}
$$

Tabelle 1: Auszug aus den Spezifikationen

In Tabelle 1 befindet sich ein Teil der Spezifikationen dieses Algorithmus. Die frischen Typvariablen die an manchen Stellen erstellt werden, bezeichnen Variablen, die nicht in den Aufrufargumenten von W_{UL} enthalten sind, oder in einem anderen Teil des Algorithmus bereits als frisch deklariert wurden.
Der eigentliche Ablauf des Algorithmus sieht nun wie folgt aus: In den beiden "Grundfällen"des Algorithmus wird als erstes Argument jeweils die Typumgebung Γ übergeben und dann entweder eine Konstante c oder eine Variable x. Im ersten Fall bezeichnet der Aufruf nichts anderes als eine Identitätsabbildung mit dem Ergebnistyp τ_c. Der zweite Fall bezeichnet auch eine Identitätsabbildung, allerdings wird hierbei die Typumgebung zu Hilfe

genommen um den erweiterten Typen von x zu bestimmen.
Was macht der Algorithmus bei komplexeren Ausdrücken? Der Ausdruck lautet nun $fn_\pi x \Rightarrow e$: Als erstes wird, wie oben bereits erklärt, eine frische Typvariable α_x angelegt. Danach erfolgt ein rekursiver Aufruf von W_{UL} mit dem Ausdruck e und der Typumgebung, wobei nun ein Eintrag der Form $[\mathrm{x} \mapsto \alpha_x]$ hinzugefügt wurde. Die Substitution die man durch diesen rekursiven Aufruf erhält wird danach auf α_x angewandt um den ersten Teil des Gesamttyps zu finden. Der zweite Teil wurde bereits im rekursiven Aufruf von W_{UL} gefunden, genauso wie die endgültige Substitution.
Der Ablauf beim Ausdruck $fun_\pi fx \Rightarrow e_0$ ist ähnlich zu vorherigem Ablauf. Zuallererst wird wieder α_x erstellt und dabei eine zweite Typvariable α_0. Danach wird W_{UL} wieder rekursiv aufgerufen, allerdings mit einer, um den Eintrag $[f \mapsto \alpha_x \to \alpha_0]$, erweiterten Typumgebung. Daraufhin wird die Substitution θ_1 über den Aufruf von U_{UL}, der später noch erklärt wird, mit den Argumenten $\tau_0, \theta_0\alpha_0$, bestimmt. Das Endergebnis des Typs, wird wieder durch die Anwendung der vorher bestimmten Substitutionen auf die Typvariablen bzw. erweiterten Typen bestimmt. Die endgültige Substitution findet sich durch das Verschmelzen der Substitutionen.
Ebenfalls möglich ist eine Verkettung von Ausdrücken. Der Ablauf ist dabei relativ trivial. Der Algorithmus teilt die Kette in Einzelausdrücke auf und bestimmt dann jeweils durch einen rekursiven Aufruf den Typ und die Substitution. Zu beachten ist, dass die Substitution die durch die Auswertung des ersten Teilausdrucks gefunden wurde, vorher auf die Typumgebung des zweiten W_{UL} Aufrufs anzuwenden ist. Am Ende wird durch einen Aufruf von U_{UL} mit den Parametern $\theta_2\tau_1$ und $\tau_2 \to \alpha$, wobei α wieder eine frische Typvariable ist, die Substitution θ_3 bestimmt. Das Endergebnis ergibt sich wieder durch Anwendung von θ_3 auf α (Typ) und Verkettung der Substitutionen (Gesamtsubstitution).

3.2 Beispiel

Um diese etwas trockene Theorie besser verständlich zu machen, ist das Heranziehen des folgenden Beispiel ratsam $e = (fn_X\ \mathrm{x} \Rightarrow \mathrm{x})\ (\ fn_Y\ \mathrm{y} \Rightarrow \mathrm{y})$

Wie wertet der Algorithmus diesen Ausdruck aus? Als erstes wird der Algorithmus mit e in der Form $W_{UL}([\,], (fn_X\ \text{x}\Rightarrow\text{x})\ (fn_Y\ \text{y}\Rightarrow\text{y}))$ aufgerufen. Da eine Verkettung von Ausdrücken vorliegt, spaltet der Algorithmus diese in $W_{UL}([\,],\ fn_X\ \text{x}\Rightarrow\text{x})$ und $W_{UL}([\,],\ fn_Y\ \text{y}\Rightarrow\text{y})$ auf und verwertet sie einzeln. Hierbei kommt er durch den rekursiven Aufruf auf die Ergebnisse ('a → 'a,*id*) und ('b → 'b,*id*). Diese beiden Ergebnisse werden mitsamt der frischen Typvariable α dem Algorithmus U_{UL} übergeben, um θ_3 zu erhalten. Der Aufruf sieht dabei aus wie folgt:
U_{UL}('a → 'a, ('b → 'b) → 'c). Als Endergebnis des obigen Ausdrucks bekommen wir also 'b → 'b als Typ und ['a ↦ 'b → 'b]['c ↦ 'b → 'b] als Substitution.

4 Der U_{UL} Algorithmus

4.1 Idee

Im vorherigen Kapitel wurde der U_{UL} Algorithmus verwendet und es ist bereits bekannt, dass er eine Substitution erstellt. Doch was macht er genau? Der Algorithmus wird immer mit zwei erweiterten Typen aufgerufen und gibt dabei, falls möglich, eine Substitution zurück. Er versucht also ein θ zu finden, sodass $\theta\tau_1 = \theta\tau_2$ ist. Beispielsweise würde das im Fall
$(\alpha \to \text{bool}, \text{int} \to \beta)$ bedeuten, dass α auf int gemappt wird und β auf bool. Es gibt allerdings auch Hürden für den Algorithmus. Er bricht sowohl bei top-level Konstruktoren ab, was in diesem Fall int, bool, $\to$ entspricht, als auch bei Funktionstypen welche die Typvariable enthalten, mit der sie vereinigt werden sollen. Da W_{UL} sich auf U_{UL} stützt, kann es dazu kommen, dass W_{UL} zu keinem Ergebnis führt.

$$
\begin{aligned}
U_{UL}(\tau_1 \to \tau_2, \tau_1{}' \to \tau_2{}') = \text{let}& \\
&\theta_1 = U_{UL}(\tau_1, \tau_1{}') \\
&\theta_2 = U_{UL}(\theta_1\tau_2, \theta_1\tau_2{}') \\
&\text{in } \theta_2 \circ \theta_1 \\
U_{UL}(\tau, \alpha) = &\begin{cases} [\alpha \mapsto \tau] & \text{falls } \alpha \text{ nicht in } \tau \text{ auftaucht} \\ & \text{oder } \alpha = \tau \text{ ist} \\ \text{fail} & \text{sonst} \end{cases} \\
U_{UL}(\alpha, \tau) = &\text{ s.o.} \\
U_{UL}(\tau_1, \tau_2) = &\text{ fail}
\end{aligned}
$$

Tabelle 2: Auszug aus den Spezifikationen

Bei der Betrachtung des Algorithmus fällt auf, dass der wichtigste Teil wieder auf Rekursion beruht. Bei einem Aufruf von U_{UL} mit den Argumenten $\tau_1 \to \tau_2$ und $\tau_1{}' \to \tau_2{}'$ werden zwei neue Aufrufe von U_{UL} getätigt, mit den Argumenten $\tau_1, \tau_1{}'$ bzw. $\tau_2, \tau_2{}'$, um θ_1 und θ_2 zu finden, die danach als engültige

Ergebnissubstitution zusammengeführt werden.

Vergleich Die andauernde Anwendung von bereits gefundenen Substitutionen auf Typvariablen, erweiterten Typen und Typumgebungen, ist im ersten Moment wohl etwas verwirrend, soll aber durch folgendes Beispiel erläutert werden. Stellen wir uns eine fiktive Gesellschaft vor in der gewisse Gesetze erlassen wurden. Eines davon besagt, dass ein Computerverkäufer dem Käufer alle CDs an Software, die auf dem PC verblieben ist, übergeben muss. Daraufhin wird der Verkäufer nach einigen Überlegungen einen angemessenen Preis bestimmen.
Nun wird von der Regierung allerdings gesetzlich festgelegt, dass bei einem Verkauf von Software 10% des Verkaufpreises an eine Kampagne gegen Softwarepiraterie abgeführt werden muss. Damit der Verkäufer nun kein Verlustgeschäft macht, muss er seinen Preis neu bestimmen und dabei die Abgaben, die durch das Gesetz anfallen, in seine Überlegungen miteinbeziehen.
Ähnlich verhält es sich mit dem Algorithmus. Die Typen die durch W_{UL} gefunden wurden, entsprechen dabei den Überlegungen des Verkäufers und die Substitutionen den Gesetzen, die neu erlassen werden. Falls nun ein neues Gesetz (neue Substitution) gefunden wird, müssen wir die Preisfindung (Typen) überdenken.

4.2 Beispiel

Weiter mit obigem Beispiel. Der Aufruf erfolgte durch
$U_{UL}(\acute{a} \to \acute{a}, (\acute{b} \to \acute{b}) \to \acute{c})$. Dabei ist $\acute{a} \to \acute{a}$ gleich $\tau_1 \to \tau_2$, $\acute{b} \to \acute{b}$ entspricht $\tau_1\acute{}$ und $\acute{c}$ ist $\tau_2\acute{}$
Dadurch wird die erste Regel des Algorithmus aus Tabelle 2 aufgerufen. Die rekursiven Aufrufe von U_{UL} mit den Argumenten $\acute{a}$, $\acute{b} \to \acute{b}$ und
$\acute{b} \to \acute{b}, \acute{c}$ bringen als Substitutionen $[\acute{a} \mapsto \acute{b} \to \acute{b}]$ bzw. $[\acute{c} \mapsto \acute{b} \to \acute{b}]$. Dadurch ergibt sich die endgültige Substitution $[\acute{a} \mapsto \acute{b} \to \acute{b}][\acute{c} \mapsto \acute{b} \to \acute{b}]$.

5 CFA Algorithmus

Der Algorithmus erfüllt nun alle Kriterien und liefert zuverlässig ein Ergebnis (auch ein Abbruch ist ein Ergebnis!). Doch das reicht noch nicht ganz für einen Einsatz in einem guten Compiler, denn dieser stellt u.a. auch unerreichbaren Quelltext fest. Das Feststellen von unerreichbarem Quelltext ist ein Teil der sogenannten Kontrollflussanalyse, die zudem duplizierten Quelltext sowie GOTO-Anweisungen feststellt, was aber für diesen Fall uninteressant ist.
Um diese Kontrollflussanalyse nun in den Algorithmus zu implementieren, kann nicht einfach der W_{UL} Algorithmus benutzt werden, da dieser auf U_{UL} basiert. Das Problem des U_{UL} Algorithmus ist nämlich Folgendes: Zwei Typen sind nur dann gleich, wenn ihre syntaktischen Repräsentationen gleich sind. Zwei annotierte Typen hingegen **KÖNNEN** gleich sein obwohl ihre syntaktischen Repräsentationen **NICHT** gleich sind. Das bedeutet, dass $\text{int} \xrightarrow{\{\pi_1\}\cup\{\pi_2\}} \text{int}$ das gleiche sein kann wie $\text{int} \xrightarrow{\{\pi_2\}\cup\{\pi_1\}} \text{int}$.
U_{UL} kann allerdings nur auf Typen in einer freien Algebra angewandt werden, also auf die erweiterten Typen. Es könnte auch ein anderer Algorithmus benutzt werden, den es auch durchaus gibt, allerdings ist dieser mit seinen Eigenschaften nicht so simpel wie U_{UL}. Deshalb wird der bekannte Algorithmus modifiziert um der Kontrollflussanalyse gerecht zu werden.

5.1 Constraints

Da das Problem aufgrund des Unterschieds zwischen einfachen Typen und annotierten Typen besteht, die im Gegensatz zu einfachen Typen über Annotationen verfügen, erweitern wir unseren Algorithmus um Annotationsvariablen. Diese Variablen befinden sich auf den Funktionspfeilen in der Form $\hat{\tau_1} \xrightarrow{\beta} \hat{\tau_2}$. Doch nur Variablen genügen nicht. Deshalb werden sogenannte Constraints eingeführt. Ein Constraint ist eine Inklusion der Form $\beta \supseteq \varphi$ wobei $\varphi = \{\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset \mid \beta$.
Des Weiteren gibt es ein sogenanntes Constraintset C, welches eine endliche Menge von Constraints bezeichnet. Anzumerken ist, dass Substitutionen auf C anwendbar sind, wobei gilt : wenn $\beta \supseteq \varphi$ in C dann auch $\theta\beta \supseteq \theta\varphi$ in θ C.

$$
\begin{aligned}
U_{CFA}(\hat{\tau}_1 \xrightarrow{\beta} \hat{\tau}_2, \hat{\tau}_1{}' \xrightarrow{\beta'} \hat{\tau}_2{}') &= \text{let } \theta_0 = [\beta' \mapsto \beta] \\
&\quad\ \theta_1 = U_{CFA}(\theta_0\hat{\tau}_1, \theta_0\hat{\tau}_1{}') \\
&\quad\ \theta_2 = U_{CFA}(\theta_1(\theta_0\hat{\tau}_2), \theta_1(\theta_0\hat{\tau}_2{}')) \\
&\quad\ \text{in } \theta_2 \circ \theta_1 \circ \theta_0 \\
U_{CFA}(\hat{\tau}, \alpha) &= \begin{cases} [\alpha \mapsto \hat{\tau}] & \text{falls } \alpha \text{ nicht in } \hat{\tau} \\ & \text{auftaucht od. } \alpha = \hat{\tau} \text{ ist} \\ \text{fail} & \text{sonst} \end{cases} \\
U_{CFA}(\alpha, \hat{\tau}) &= \text{s.o.} \\
U_{CFA}(\hat{\tau}_1, \hat{\tau}_2) &= \text{fail}
\end{aligned}
$$

Tabelle 3: Auszug aus den Spezifikationen

5.2 Unification

Da U_{UL} modifiziert werden konnte, ändert sich nicht viel am Ablauf bzw. der Spezifikation des Algorithmus. Die einzigen Unterschiede sind, dass der Algorithmus mittlerweile einfache Typen statt erweiterten Typen erhält (samt den Annotationsvariablen!) und er dabei eine neue Substitution bildet indem er β' auf β mapt. Diese Substitution wird daraufhin auch auf $\hat{\tau}_1$ und $\hat{\tau}_2$ angewandt!
Durch die Änderungen erhält der Algorithmus nun den Namen U_{CFA}.

Beispiel: Betrachtet wird wieder der oben genannte Fall, der bereits um Annotationsvariablen erweitert wurde: $('a \xrightarrow{'1} 'a, ('b \xrightarrow{'2} 'b) \xrightarrow{'3} 'c)$
Dabei wird nun wie bereits erwähnt $\beta' \mapsto \beta$ ausgeführt, was in unserem Fall $['3 \mapsto '1]$ entspricht. Endergebnis ist somit $['3 \mapsto '1][' a, 'b \xrightarrow{'2} 'b][' c, 'b \xrightarrow{'2} 'b]$, da wie bereits erwähnt der Ablauf des Algorithmus gleich bleibt und somit wieder rekursive Aufrufe von U_{CFA} getätig wurden.

5.3 Der Algorithmus

Nachdem der Unifizierungsalgorithmus erfolgreich geändert wurde, liegt die genauere Betrachtung nun auf dem Typrekonstruktionsalgorithmus. Die Änderungen

sind ähnlich trivial wie beim vorherigen Algorithmus. Als erstes sticht die veränderte Form hervor. Sie sieht nun aus wie folgt:
$W_{CFA}(\hat{\Gamma}, e) = (\hat{\tau}, \theta, C)$, wobei C ein spezielles Constraintset ist, das nur Constraints der Form $\beta \supseteq \{\pi\}$ enthält. Wie man an Tabelle 4 erkennen kann, gibt es keine wirklich gravierenden Änderungen am Algorithmus, bis auf die Einführung des Constraintsets und der Annotationsvariablen, die nach jeder Substitution wieder überprüft werden muss.
Da das Prinzip aber ansonsten gleich bleibt, kann eine weitere Erklärung des Algorithmus ausbleiben und bloss noch durch ein Beispiel verdeutlicht.

$$
\begin{aligned}
W_{CFA}(\hat{\Gamma}, c) &= (\tau_c, id, \emptyset) \\
W_{CFA}(\hat{\Gamma}, x) &= (\hat{\Gamma}(x), id, \emptyset) \\
W_{CFA}(\hat{\Gamma}, fn_\pi x \Rightarrow e_0) &= \text{let } \alpha_x \text{ be fresh} \\
(\hat{\tau_0}, \theta_0, C_0) &= W_{CFA}(\hat{\Gamma}[x \mapsto \alpha_x], e_0) \\
&\quad \beta_0 \text{ be fresh} \\
&\quad \text{in } ((\theta_0\alpha_x) \xrightarrow{\beta_0} \tau_0, \theta_0, C_0 \cup \{\beta_0 \supseteq \{\pi\}\}) \\
W_{CFA}(\hat{\Gamma}, fun_\pi f x \Rightarrow e_0) &= \text{let } \alpha_x, \alpha_0, \beta_0 \text{ be fresh} \\
(\hat{\tau_0}, \theta_0, C_0) &= W_{CFA}(\hat{\Gamma}[f \mapsto \alpha_x \xrightarrow{\beta_0} \alpha_0][x \mapsto \alpha_x], e_0) \\
\theta_1 &= U_{CFA}(\hat{\tau_0}, \theta_0\alpha_0) \\
&\quad \text{in } (\theta_1(\theta_0\alpha_x) \xrightarrow{\theta_1(\theta_0\beta_0)} \theta_1\hat{\tau_0}, \theta_1 \circ \theta_0), \\
&\quad (\theta_1 C_0) \cup \{\theta_1(\theta_0\beta_0) \supseteq \{\pi\}\}) \\
W_{CFA}(\hat{\Gamma}, e_1 e_2) &= \text{let } (\hat{\tau_1}, \theta_1, C_1) = W_{CFA}(\hat{\Gamma}, e_1) \\
(\hat{\tau_2}, \theta_2, C_2) &= W_{CFA}(\theta_1\hat{\Gamma}, e_2) \\
&\quad \alpha, \beta \text{ be fresh} \\
\theta_3 &= U_{CFA}(\theta_2\hat{\tau_1}, \hat{\tau_2} \xrightarrow{\beta} \alpha) \\
&\quad \text{in } (\theta_3\alpha, \theta_3 \circ \theta_2 \circ \theta_1, \theta_3(\theta_2 C_1) \cup \theta_3 C_2)
\end{aligned}
$$

Tabelle 4: Auszug aus den Spezifikationen

Beispiel: Wieder der Ausdruck (fn_X x ⇒ x) (fn_Y y ⇒ y). Bei einem Aufruf der Form W_{CFA}([ ], (fn_X x ⇒ x) (fn_Y y ⇒ y)) werden wieder die Teilausdrücke durch W_{CFA}([ ] (fn_X x ⇒ x)) und W_{CFA}([ ] (fn_Y y ⇒ y)) ausgewertet. Sie liefern (′a $\xrightarrow{'1}$ ′a, *id*, {′1 ⊇ {X}}) und (′b $\xrightarrow{'2}$ ′b, *id*, {′2 ⊇ {Y}}). Danach erfolgt ein Aufruf von U_{CFA} durch U_{CFA}(′a $\xrightarrow{'1}$ ′a, (′b $\xrightarrow{'2}$ ′b)$\xrightarrow{'3}$′c).
Dieser liefert wie oben gesehen [′3↦ ′1][′a↦ ′b$\xrightarrow{'2}$′b][′c↦ ′b$\xrightarrow{'2}$ ′b].
Das Endergebnis unseres Aufrufes ist somit der Typ ′b$\xrightarrow{'2}$′b, die Substitution [′3↦ ′1][′a↦ ′b$\xrightarrow{'2}$′b][′c↦ ′b$\xrightarrow{'2}$ ′b] und das Constraintset
{′1 ⊇ {X}, ′2 ⊇ {Y}}

6 Abschließende Worte

Diese Arbeit sollte eine kurze Einführung in einen Typinferenzalgorithmus geben, der in einer genaueren Form Verwendung in Programmiersprachen wie SML und $Huskell$ findet, als auch in dem Theorembeweiser $Isabelle$ der TU München. Als Grundlage mussten natürlich einige Symbole erklärt werden, sowie das zugrunde liegende Typsystem, das durch die Verwendung von anders deklarierten Typen einfacher zu handhaben war. Wichtig dabei war es die Arbeitsweise des W_{UL} Algorithmus zu verstehen, inwiefern er mit dem U_{UL} Algorithmus zusammenarbeitet und zu sehen, dass Rekursion bei beiden eine zentrale Rolle spielt.
Ein weiterer entscheidender Punkt, war der Übergang von W_{UL} bzw U_{UL} zu Algorithmen mit Kontrollflussanalyse. Nicht, dass die Algorithmen sich grundlegend verändert hätten, aber die Defizite des vorher verwendeten Typs, waren hierbei entscheidend. Die eigentliche Entscheidung den Algorithmus um eine Kontrollflussanalyse zu erweitern, war immens wichtig und wurde durch die Einführung von Constraints und Annotationsvariablen gut gelöst. Abschließend bleibt zu sagen, dass der W_{CFA} Algorithmus eine gute Grundlage für einen Compiler bildet und eine relativ simple Lösung ist, um Typen zu inferieren.