Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Prof. Dr.-Ing. Gregor Snelting
gregor.snelting@kit.edu
Prof. Dr. Ralf Reussner
reussner@kit.edu

Programmierparadigmen – WS 2021/22

https://pp.ipd.kit.edu/lehre/WS202122/paradigmen/uebung

Blatt 6: Rekursionsoperatoren, Typprüfung

Abgabe: 3.12.2021, 14:00
Besprechung: 6.12.2021–7.12.2021

Reichen Sie Ihre Abgabe bis zum 3.12.2021 um 14:00 in unserer Praktomat-Instanz unter https://praktomat.cs.kit.edu/pp_2021_WS ein.

1 Von Haskell zum λ-Kalkül

Übersetzen Sie folgende Haskell-Funktion in einen äquivalenten Term des λ-Kalküls. Der resultierende Term muss nur auf Church-Zahlen funktionieren. Sie können dazu alle in der Vorlesung und den Übungen bisher definierten Funktionen verwenden.

Definieren Sie bei Bedarf die Vergleichsfunktion *lessEq* im λ-Kalkül.[1]

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

2 λ-Terme und ihre Typen

Gegeben seien folgende λ-Terme:

$A = \lambda \mathtt{f}.\ \lambda \mathtt{x}.\ \mathtt{f}\ \mathtt{x}$
$B = \lambda \mathtt{f}.\ \lambda \mathtt{x}.\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$
$C = \lambda \mathtt{f}.\ (\lambda \mathtt{x}.\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda \mathtt{x}.\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))$
$D = \lambda \mathtt{x}.\ \lambda \mathtt{y}.\ \mathtt{y}\ (\mathtt{x}\ \mathtt{y})$
$E = \lambda \mathtt{z}.\ \mathtt{z}$
$F = D\ E = (\lambda \mathtt{x}.\ \lambda \mathtt{y}.\ \mathtt{y}\ (\mathtt{x}\ \mathtt{y}))\ (\lambda \mathtt{z}.\ \mathtt{z})$

Geben Sie für jeden der folgenden Typen *all* die Terme aus A – F an, die diesen Typ haben können.

1. $\alpha \to \alpha$
2. $(\alpha \to \alpha) \to \alpha \to \alpha$
3. $(\alpha \to \beta) \to \alpha \to \beta$
4. $((\alpha \to \beta) \to \alpha) \to (\alpha \to \beta) \to \beta$
5. nicht typisierbar

[1] Hinweis: Subtraktion auf Church-Zahlen ist sättigend, d.h. $n - m = 0$ für $n \leq m$

3 Typ-Prüfung

Es sei $\Gamma = \texttt{a} : \mathsf{int}, \texttt{b} : \mathsf{bool}, \texttt{c} : \mathsf{char}$.

Vervollständigen Sie den folgenden Herleitungsbaum:

$$\frac{\Gamma \vdash \lambda \texttt{x}.\ \lambda \texttt{y}.\ \texttt{x} : \alpha \to \beta \to \alpha \qquad \Gamma, k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \vdash \texttt{k a (k b c)} : \mathsf{int}}{\Gamma \vdash \textbf{let}\ \texttt{k} = \lambda \texttt{x}.\ \lambda \texttt{y}.\ \texttt{x}\ \textbf{in}\ \texttt{k a (k b c)} : \mathsf{int}}\ \textit{Let}$$

Benutzen Sie (neben trivialen Instanziierungen) die Instanziierungen:

- $\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \succeq \mathsf{int} \to \mathsf{bool} \to \mathsf{int}$
- $\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \succeq \mathsf{bool} \to \mathsf{char} \to \mathsf{bool}$

B-Seite: Ausflug: Mit starken Typen abhängen

Ist Ihnen in der Vorlesung eine gewisse Ähnlichkeit zwischen Logik- und Typregeln aufgefallen?

$$\Rightarrow\!\text{I}\ \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\ t : \tau_1 \to \tau_2}$$

$$\text{MP}\ \frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \qquad \textsc{App}\ \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$\textsc{AssmI}\ \frac{}{\Gamma, \varphi \vdash \varphi} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Wenn wir auf der rechten Seite die Terme plus Doppelpunkt wegstreichen (davor $\Gamma(x) = \tau$ durch das äquivalente $\Gamma = \Gamma', x : \tau$ ersetzen) und die verschiedene Pfeile miteinander identifizieren, sind diese Regel nicht nur ähnlich, sondern sogar äquivalent!

$$\Rightarrow\!\text{I}\ \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \qquad \textsc{Abs:}\ \frac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}$$

$$\text{MP}\ \frac{\Gamma \vdash \varphi \Rightarrow \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \qquad \textsc{App:}\ \frac{\Gamma \vdash \tau_2 \to \tau \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau}$$

$$\textsc{AssmI}\ \frac{}{\Gamma, \varphi \vdash \varphi} \qquad \textsc{Var:}\ \frac{\Gamma = \Gamma', \tau}{\Gamma \vdash \tau}$$

Insbesondere erkennen wir, dass man Abstraktions- und Applikationsregel als Introduktions- bzw. Eliminationsregel des Funktionstyps ansehen kann.

William Alvin Howard zeigte 1969 aufbauend auf Erkentnissen von Haskell Curry, dass wir alle Regeln der (intuitionistischen) Aussagenlogik auf der einen Seite und des einfach typisierten Lambda-Kalküls auf der anderen Seite ineinander überführen können, die sogenannte *Curry-Howard-Korrespondenz*.

Eine solche Aussage ist also genau dann beweisbar, wenn ein Lambdaterm mit dem entsprechenden Typ existiert. Damit können wir Lambdaterme als formale Beweissprache verwenden und Typchecker dafür als Beweischecker! Auch Haskell-Terme sind statt Lambdatermen geeignet, solange wir etwas aufpassen[2]. So beweist folgende Haskell-Funktion beispielsweise die Aussage $(A \implies A \implies B) \implies (A \implies B)$:

```
f :: (a -> a -> b) -> (a -> b)
f g a = g a a
```

1. Definieren Sie Haskell-Typen `And a b` und `Or a b`, die den Aussagen $A \wedge B$ bzw. $A \vee B$ entsprechen, übersetzen Sie deren Introduktions- und Eliminationsregeln zu Haskell-Typsignaturen und geben Sie Implementierungen dazu an. Falls Sie die entsprechenden Haskell-Typen nicht erkennen, kann es auch hilfreich sein, zuerst die Regeln zu übersetzen und sich dann zu überlegen, für welche Typen diese Funktionen implementierbar sind.

2. Beweisen Sie in Haskell:

   (a) $A \wedge B \implies B \wedge A$

   (b) $(A \vee B) \wedge C \implies (A \wedge C) \vee (B \wedge C)$

3. Etwas interessantere Beweise können wir zusammen mit dem leeren (also *unbeweisbaren*) Typ `False` und dem Negationstyp `Not a` führen:

   ```
   data False
   type Not a = a -> False
   ```

   Die Definition von `Not a` drückt aus, dass wir die Aussage $\neg A$ beweisen können, indem wir jeden Beweis von A zum Widerspruch führen.

   Beweisen Sie in Haskell:

   (a) $A \implies \neg\neg A$

   (b) $\neg(A \wedge \neg A)$

   (c) $\neg A \vee \neg B \implies \neg(A \wedge B)$

4. Letztere Aussage gilt in der klassischen Logik auch andersherum. Können Sie sie auch in Haskell beweisen?

Die Korrespondenz auf Ebene der Aussagenlogik ist zwar theoretisch interessant, praktisch aber zu schwach, um damit die meisten interessanten Aussagen zu kodieren. Dafür benötigen wir mindestens Prädikatenlogik erster Stufe, also insbesondere den Allquantor. Weder das Typsystem der Vorlesung noch das von Standard-Haskell kennt aber einen dazu passenden Typ: Eine Formel wie $\forall x \in M.\ P(x)$ können wir uns nun vorstellen als eine Funktion, die jedes Element x aus M abbildet auf einen Beweis von $P(x)$. Funktionstypen in Haskell haben aber einen festen Wertebereich, der nicht vom abzubildenden Element abhängen kann! Solche (Wert-)*abhängigen Typen* könnten beispielsweise den Typ `Vector a n` der Listen der Länge `n` (einem Wert!) mit Elementen aus `a` beschreiben. Mit solch einem Typ könnten wir beispielsweise eine sicherere `head`-Funktion schreiben, die erst gar nicht auf leere Listen anwendbar ist:

```
head :: (n :: Nat) -> Vector a (n + 1) -> a
```

[2] Alle Haskell-Terme müssen hier *wohldefiniert* sein, also auf allen Eingaben terminieren, sonst können wir sie nicht in den einfach typisierten Lambda-Kalkül (ohne `define` aus der Vorlesung) übersetzen. Insbesondere würde `undefined` jede Aussage trivial beweisen.

Die wenigsten Programmiersprachen erlauben solch präzise Typen, in der Mathematik sind entsprechende Mengen dagegen gang und gäbe. Beispielsweise könnte eine Haskell-artige Typsignatur für Matrixmultiplikation angegeben werden als

```
matmul :: (n m :: Nat) -> Matrix a n m -> Matrix a m k -> Matrix a n k
```

Abhängige Typen und die Curry-Howard-Korrespondenz werden deswegen gern als theoretisches Fundament von *Theorembeweisern* verwendet: Programme, die zum Erstellen und Überprüfen von Computerbeweisen, beispielsweise über Mathematik oder die Korrektheit von anderen Programmen, entwickelt sind. Im Rahmen dieser Vorlesung möchten wir zu diesen Themen nicht weiter ausschweifen; wenn Sie es aber bis hierhin geschafft haben, laden wir Sie herzlich dazu ein, auf eigene Faust zum Beispiel das Kapitel 3.6 und 3.7 aus *The Hitchhiker's Guide to Logical Verification* über den an unserem Lehrstuhl mitentwickelten Theorembeweiser *Lean* anzuschauen. [3]

Desweiteren kann die Curry-Howard-Korrespondenz sowie die Anwendung von Lean in der Master-Veranstaltung *Theorembeweiserprakikum – Anwendungen in der Sprachtechnologie* weiter vertieft werden.

[3] https://github.com/blanchette/logical_verification_2020/raw/master/hitchhikers_guide.pdf#section.3.6