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)
```

Beispiellösung:

`fib` ohne Pattern-Matching sowie das dazugehörige Funktional `F`:

```
fib n = if n == 0 || n == 1 then 1 else fib (n - 1) + fib (n - 2)
F f n = if n == 0 || n == 1 then 1 else f   (n - 1) + f   (n - 2)
```

Die Bedingung `n == 0 || n == 1` kann im λ-Kalkül einfach als *isZero* (*pred* `n`) geschrieben werden.

$$F = \lambda \texttt{f}.\ \lambda \texttt{n}.\ (\mathit{isZero}\ (\mathit{pred}\ \texttt{n}))\ c_1\ (\mathit{plus}\ (\texttt{f}\ (\mathit{sub}\ \texttt{n}\ c_1))\ (\texttt{f}\ (\mathit{sub}\ \texttt{n}\ c_2)))$$
$$\mathit{fib} = \mathrm{Y}\ F$$

Die <=-Operation lässt sich übrigens wie folgt umsetzen:

$$\mathit{lessEq} = \lambda \texttt{m}.\ \lambda \texttt{n}.\ \mathit{isZero}\ (\mathit{sub}\ \texttt{m}\ \texttt{n})$$

Bemerkung: Der Rekursionsoperator Y ist in Haskell nicht direkt typisierbar. Über einen Trick lässt sich trotzdem ein sehr ähnlicher Fixpunktoperator definieren, der *ohne rekursive Funktionsdefinition* auskommt:

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

```
data T a = T (T a -> a)

unT :: T a -> T a -> a
unT (T f) = f

fix :: (a -> a) -> a
fix f = (\x -> f (unT x x)) (T (\x -> f (unT x x)))

fib = fix F
```

Haskell wäre also auch ohne rekursive Funktionsdefinitionen Turing-vollständig. Das ist kompatibel zu Folie 21.16, denn die bezieht sich nur auf den typisierten Lambdakalkül mit **`let`**. Hier aber spielt der *rekursive Datentyp* `T` (genauer gesagt das rekursive Vorkommen im Argument der Funktion) eine wichtige Rolle, denn ohne ihn kann man `fix` nicht typisieren. Tatsächlich ist obiges Programm Beweis dafür, dass rekursive Typen schon für Turing-Vollständigkeit reichen!

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. $\alpha \to \alpha$: E
2. $(\alpha \to \alpha) \to \alpha \to \alpha$: A, B, E
3. $(\alpha \to \beta) \to \alpha \to \beta$: A, E
4. $((\alpha \to \beta) \to \alpha) \to (\alpha \to \beta) \to \beta$: D
5. nicht typisierbar : C, F

3 Typ-Prüfung

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

Vervollständigen Sie den folgenden Herleitungsbaum:

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

Benutzen Sie (neben trivialen Instanziierungen) die Instanziierungen:

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

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

Beispiellösung: siehe nächste Seite

(A) Herleitungsbaum für $\Gamma_{let} \vdash \texttt{k a} : \mathsf{bool} \to \mathsf{int}$

$$\dfrac{\dfrac{\dfrac{\Gamma_{let}(\texttt{k}) = \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha}{\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \succeq \mathsf{int} \to \mathsf{bool} \to \mathsf{int}}\,\mathit{Var}}{\Gamma_{let} \vdash \texttt{k} : \mathsf{int} \to \mathsf{bool} \to \mathsf{int}} \qquad \dfrac{\dfrac{\Gamma_{let}(\texttt{a}) = \mathsf{int}}{\mathsf{int} \succeq \mathsf{int}}\,\mathit{Var}}{\Gamma_{let} \vdash \texttt{a} : \mathsf{int}}}{\Gamma_{let} \vdash \texttt{k a} : \mathsf{bool} \to \mathsf{int}}\,\mathit{App}$$

(B) Herleitungsbaum für $\Gamma_{let} \vdash \texttt{k b c} : \mathsf{bool}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_{let}(\texttt{k}) = \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha}{\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \succeq \mathsf{bool} \to \mathsf{char} \to \mathsf{bool}}\,\mathit{Var}}{\Gamma_{let} \vdash \texttt{k} : \mathsf{bool} \to \mathsf{char} \to \mathsf{bool}} \qquad \dfrac{\dfrac{\Gamma_{let}(\texttt{b}) = \mathsf{bool}}{\mathsf{bool} \succeq \mathsf{bool}}\,\mathit{Var}}{\Gamma_{let} \vdash \texttt{b} : \mathsf{bool}}}{\Gamma_{let} \vdash \texttt{k b} : \mathsf{char} \to \mathsf{bool}}\,\mathit{App} \qquad \dfrac{\dfrac{\Gamma_{let}(\texttt{c}) = \mathsf{char}}{\mathsf{char} \succeq \mathsf{char}}\,\mathit{Var}}{\Gamma_{let} \vdash \texttt{c} : \mathsf{char}}}{\Gamma_{let} \vdash \texttt{k b c} : \mathsf{bool}}\,\mathit{App}$$

(C) Herleitungsbaum für $\Gamma \vdash \textbf{let}\ \texttt{k} = \lambda\texttt{x}.\ \lambda\texttt{y}.\ \texttt{x}\ \textbf{in}\ \texttt{k a (k b c)} : \mathsf{int}$

$$\dfrac{\dfrac{\dfrac{\dfrac{(\Gamma, x : \alpha, y : \beta)(\texttt{x}) = \alpha}{\alpha \succeq \alpha}\,\mathit{Var}}{(\Gamma, x : \alpha, y : \beta) \vdash \texttt{x} : \alpha}\,\mathit{Abs}}{\Gamma, x : \alpha \vdash \lambda\texttt{y}.\ \texttt{x} : \beta \to \alpha}\,\mathit{Abs}}{\Gamma \vdash \lambda\texttt{x}.\ \lambda\texttt{y}.\ \texttt{x} : \alpha \to \beta \to \alpha} \qquad \dfrac{\dfrac{\text{siehe (A)}}{\Gamma_{let} \vdash \texttt{k a} : \mathsf{bool} \to \mathsf{int}}\,\mathit{App} \qquad \dfrac{\text{siehe (B)}}{\Gamma_{let} \vdash \texttt{k b c} : \mathsf{bool}}\,\mathit{App}}{\Gamma_{let} \vdash \texttt{k a (k b c)} : \mathsf{int}}\,\mathit{App}}{\Gamma \vdash \textbf{let}\ \texttt{k} = \lambda\texttt{x}.\ \lambda\texttt{y}.\ \texttt{x}\ \textbf{in}\ \texttt{k a (k b c)} : \mathsf{int}}\,\mathit{Let}$$

Dabei ist $\Gamma_{let} = \Gamma, k : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$.

B-Seite: Ausflug: Mit starken Typen abhängen

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

$$\Rightarrow\!\triangleright\text{I}\ \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow\!\triangleright\ \psi} \qquad \text{MP}\ \frac{\Gamma \vdash \varphi \Rightarrow\!\triangleright\ \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \qquad \textsc{AssmI}\ \frac{}{\Gamma, \varphi \vdash \varphi}$$

$$\textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\ t : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

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!

$$\Rightarrow\!\triangleright\text{I}\ \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow\!\triangleright\ \psi} \qquad \text{MP}\ \frac{\Gamma \vdash \varphi \Rightarrow\!\triangleright\ \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \qquad \textsc{AssmI}\ \frac{}{\Gamma, \varphi \vdash \varphi}$$

$$\textsc{Abs:}\ \frac{\Gamma, \tau_1 \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \textsc{App:}\ \frac{\Gamma \vdash \tau_2 \to \tau \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau} \qquad \textsc{Var:}\ \frac{\Gamma = \Gamma', \tau}{\Gamma \vdash \tau}$$

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 $(A \Longrightarrow A \Longrightarrow B) \Longrightarrow (A \Longrightarrow B)$:

```
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] 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.

Beispiellösung:

```
type And a b = (a, b)

andI :: a -> b -> And a b
andI hA hB = (hA, hB)

andE1 :: And a b -> a
andE1 (hA, hB) = hA

andE2 :: And a b -> b
andE2 (hA, hB) = hB

data Or a b =  -- auch bekannt als 'Either'
    OrI1 a  -- OrI1 :: a -> Or a b
  | OrI2 b  -- OrI2 :: b -> Or a b

orE :: Or a b -> (a -> c) -> (b -> c) -> c
orE (OrI1 hA) f _ = f hA
orE (OrI2 hB) _ g = g hB
```

2. Beweisen Sie in Haskell:

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

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

   Beispiellösung:

   ```
   andComm :: And a b -> And b a
   andComm (hA, hB) = (hB, hA)

   orDistr :: And (Or a b) c -> Or (And a c) (And b c)
   orDistr (OrI1 hA, hC) = OrI1 (hA, hC)
   orDistr (OrI2 hB, hC) = OrI2 (hB, hC)
   ```

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) $A \Longrightarrow \neg\neg A$

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

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

   Beispiellösung:

   ```
   notNot :: a -> Not (Not a)
   notNot hA = \hNotA -> hNotA hA

   aAndNotANeg :: Not (And a (Not a))
   ```

```
aAndNotANeg = \hAAndNotA -> (andE2 hAAndNotA) (andE1 hAAndNotA)

deMorgan :: Or (Not a) (Not b) -> Not (And a b)
deMorgan (OrI1 hNotA) = \hAAndB -> hNotA (andE1 hAAndB)
deMorgan (OrI2 hNotB) = \hAAndB -> hNotB (andE2 hAAndB)
```

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

   Beispiellösung: Nein, die Umkehrung kann in der *intuitionistischen* Logik, der unser Haskell-Fragment entspricht, nicht bewiesen werden. Grob gesagt ist unser `Or` stärker, als man es normalerweise aus der Logik kennt: Es sagt/fordert nicht nur, dass eine der beiden Seiten wahr ist, sondern auch *welche* von beiden, denn wir müssen ein *deterministisches* Programm angeben, das einen Wert von `Or` berechnet. Wir können unser Fragment aber um das ebenfalls nicht beweisbare *Law of excluded middle* $A \vee \neg A$ erweitern, um alle Aussagen der klassischen Aussagenlogik zu beweisen:

   ```
   lem :: Or a (Not a)
   lem = undefined

   deMorganRev :: Not (And a b) -> Or (Not a) (Not b)
   deMorganRev hNotAAndB = case lem of
     OrI1 hA -> OrI2 (\hB -> hNotAAndB (hA, hB))
     OrI2 hNotA -> OrI1 hNotA
   ```

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
```

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]

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

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