Friedrich-Alexander-Universität
Technische Fakultät

FAU

Theorie der Programmierung

Übung 08 - Curry-Howard und induktive Datentypen

Leon Vatthauer

19. Juni 2023

Aufgabe 1

Der Curry-Howard-Isomorphismus

Wie bereits in der Hausaufgabe zum letzten Blatt angedeutet, gibt es eine Korrespondenz zwischen *minimaler Logik* und Inhabitation im einfach getypten λ-Kalkül.

Minimale Logik besitzt eine sehr eingeschränkte Syntax; allerdings haben wir in den letzten Wochen auch gelernt, dass sich im ungetypten λ-Kalkül mittels Church-Kodierung auch Terme für z.B. Paare von Werten finden lassen: Nehmen wir an, $\Gamma \vdash t : \tau$ sowie $\Gamma \vdash s : \sigma$. Per Definition aus Übung 5, Blatt 4, ist dann $\mathit{pair}\ t\ s = \lambda \mathit{select}.\mathit{select}\ t\ s$ Der Prinzipaltyp dieses Terms lässt sich einfach bestimmen: er lautet $(\tau \to \sigma \to \alpha) \to \alpha$. Sie werden jedoch feststellen, dass es keine Möglichkeit gibt (warum?), einen allgemeinen Typkonstruktor für Paare zu simulieren. Wir werden später ein Typsystem kennenlernen, das es uns erlaubt, über Typvariablen zu quantifizieren und damit das gewünschte zu erreichen.

Aufgabe 1.1

Der Curry-Howard-Isomorphismus

Für den Moment wollen wir uns jedoch damit zufriedengeben, den einfach getypten λ-Kalkül um einen solchen Typkonstruktor zu erweitern. Die neue Grammatik für Typen lautet:

$$\tau, \sigma ::= a \mid \mathbf{b} \mid \tau \to \sigma \mid \tau \times \sigma \qquad\qquad a \in \mathbf{V}, \mathbf{b} \in \mathbf{B}$$

Natürlich müssen wir damit auch neue Konstrukte einführen, die zu diesen Typkonstruktoren gehören. Die Termsprache wird also erweitert auf die Grammatik

$$t, s ::= x \mid ts \mid \lambda x.t \mid \{t, s\} \mid \mathit{fst}\ t \mid \mathit{snd}\ t$$

Geben Sie die zusätzlich benötigten Typisierungsregeln an und erweitern Sie ebenso die Auswertungsrelation um Grundreduktion für Paare.

Aufgabe 1.1

Recall: einfach getypter λ-Kalkül

Typisierung

Wir lesen $\Gamma \vdash t : \alpha$ als „im Kontext Γ hat der Term t den Typ α“ und definieren diese Relation wie folgt:

$$(\mathsf{Ax})\ \frac{}{\Gamma \vdash x : \alpha}\ (x : \alpha \in \Gamma) \qquad (\to_i)\ \frac{\Gamma[x \mapsto \alpha] \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta}$$

$$(\to_e)\ \frac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash s : \alpha}{\Gamma \vdash t\ s : \beta}$$

Aufgabe 1.2

Der Curry-Howard-Isomorphismus

Wir behaupten nun, dass diese Erweiterung uns auch mehr Ausdrucksstärke in der Logik auf der anderen Seite des Curry-Howard-Isomorphismus einbringt. Genauer gesagt, dass wir das folgende Fragment der intuitionistischen propositionalen Logik erhalten:

$$\phi, \psi ::= a \mid \phi \to \psi \mid \phi \wedge \psi$$

wobei *a* propositionale Variablen sind. Als Deduktionssystem verwenden wir den Sequentenkalkül minimaler Logik mit folgenden zusätzlichen Regeln:

$$\wedge_{E1}\ \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \wedge_{E2}\ \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \qquad \wedge_{I}\ \frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$$

Sei $\overline{\phi}$ der Typ, der entsteht, wenn man in ϕ alle $\wedge$ durch $\times$ ersetzt. Beweisen Sie: Der Sequent $\vdash \phi$ ist genau dann herleitbar, wenn der Typ $\overline{\phi}$ inhabited ist.

Aufgabe 1.2

Überblick Sequenten- und λ-Kalkül

Sequentenkalkül
mit Γ als Formelmenge $\{\phi_1, \ldots, \phi_n\}$

$$(\mathsf{AX})\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

$$(\to_I)\ \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \qquad (\wedge_I)\ \frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$$

$$(\to_E)\ \frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

$$(\wedge_{E1})\ \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad (\wedge_{E2})\ \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$$

Einfach getypter λ-Kalkül
mit Γ als Kontext $\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$

$$(\mathsf{Ax})\ \frac{}{\Gamma \vdash x : \alpha}\ x : \alpha \in \Gamma$$

$$(\to_i)\ \frac{\Gamma[x \mapsto \alpha] \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta} \qquad (\mathit{prod})\ \frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash \{t, s\} : \tau \times \sigma}$$

$$(\to_e)\ \frac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash s : \alpha}{\Gamma \vdash ts : \beta}$$

$$(\mathit{proj}_1)\ \frac{\Gamma \vdash p : \tau \times \sigma}{\Gamma \vdash \mathit{fst}\ p : \tau} \qquad (\mathit{proj}_2)\ \frac{\Gamma \vdash p : \tau \times \sigma}{\Gamma \vdash \mathit{snd}\ p : \sigma}$$

Aufgabe 1.3

Der Curry-Howard-Isomorphismus

Benutzen Sie die eben hergestellte Korrespondenz, um zu zeigen, dass folgende Formel eine Tautologie ist:

$$(p \wedge q) \to r \to ((r \wedge p) \wedge q)$$

Erweiterte Termgrammatik

$$t, s ::= x \mid ts \mid \lambda x.t \mid \{t, s\} \mid fst\ t \mid snd\ t$$

Aufgabe 2

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Einige Annahmen

Wir nehmen Typ **Nat** von Konstanten $0, 1, 2, \ldots$ und die üblichen Grundoperationen für **Nat** als gegeben an. Weiter nehmen wir den Typ **Bool** mit den Konstanten *True* und *False* sowie den Grundoperationen dazu als gegeben an. Mit `()` bezeichnen wir den Typ *unit*; er enthält nur einen einzigen Wert, den wir ebenfalls mit `()` bezeichnen.

Weiterhin schreiben wir in Beweisen der Gleichheit zweier gegebener Terme *s* und *t* die Aussage $s \leftrightarrow^*_{\beta\delta} t$ abgekürzt als $s = t$.

Aufgabe 2.1

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Beschreiben Sie in eigenen Worten die durch die folgenden Terme gegebenen Listen und Binärbäume mit natürlichen Zahlen bzw. zeichnen Sie diese.

- Nil
- Cons 5 Nil
- Leaf 13
- Inner 5 (Leaf 3) (Leaf 9)
- Cons 5 (Cons 5 Nil)
- Cons 1 (Cons 2 (Cons 3 4 Nil))
- Inner 8 (Inner 4 (Leaf 1) (Leaf 20)) (Leaf 8)
- Inner 6 (Leaf 99) (Inner 1 (Leaf 4) (Leaf 6))

Aufgabe 2.2

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Es können nun Funktionen induktiv über der Struktur von **List** *a* und **Tree** *a* definiert werden, beispielsweise:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

```
size (Leaf x)      = 1
size (Inner x l r) = 1 + size l + size r
```

(a) Welchen Typ hat `length`? Welchen hat `size`?

(b) Werten Sie den Term `length (Cons 4 (Cons 89 (Cons 21 Nil)))` aus.

Aufgabe 2.3

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Es können nun Funktionen induktiv über der Struktur von **List** *a* und **Tree** *a* definiert werden, beispielsweise:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

```
size (Leaf x)      = 1
size (Inner x l r) = 1 + size l + size r
```

Schreiben Sie eine Funktion `element` : **Nat** → **List Nat** → **Bool**, so dass `element a xs = True` wenn `a` in `xs` vorkommt, und andernfalls `element a xs = False`.

Aufgabe 3.1

Fold-Funktionen

Jeder induktive Datentyp besitzt eine Fold-Funktion, die sich aus der initialen Algebrastruktur des Typs ergibt. Der Typ und die Definitionen dieser Fold-Funktionen ergeben sich dabei allein aus den Typen der Konstruktoren. Beispielsweise ist die Fold-Funktion für den Datentyp **Nat** *a* aus der vorangegangenen Übung wie folgt definiert:

```
foldL : c → (a → c → c) → List a → c
foldL n f Nil = n
foldL n f (Cons x xs) = f x (foldL n f xs)
```

Hierbei entspricht der Typparameter *c* einem Ergebnistyp und die beiden Argumente *n* und *f* den Konstruktoren *Nil* und *Cons*, wobei die Typen der Argumente jeweils Operationen auf dem Ergebnistyp *c* beschreiben, mit dem eine Liste **List** *a* in einen einzelnen Wert vom Typ *c* „zusammengefaltet“ werden kann.

Werten Sie den Term `foldL n f (Cons 2 (Cons 3 (Cons 6 Nil)))` so weit es geht aus.

Aufgabe 3.2

Fold-Funktionen

Jeder induktive Datentyp besitzt eine Fold-Funktion, die sich aus der initialen Algebrastruktur des Typs ergibt. Der Typ und die Definitionen dieser Fold-Funktionen ergeben sich dabei allein aus den Typen der Konstruktoren. Beispielsweise ist die Fold-Funktion für den Datentyp **Nat** *a* aus der vorangegangenen Übung wie folgt definiert:

```
foldL : c → (a → c → c) → List a → c
foldL n f Nil = n
foldL n f (Cons x xs) = f x (foldL n f xs)
```

Hierbei entspricht der Typparameter *c* einem Ergebnistyp und die beiden Argumente *n* und *f* den Konstruktoren *Nil* und *Cons*, wobei die Typen der Argumente jeweils Operationen auf dem Ergebnistyp *c* beschreiben, mit dem eine Liste **List** *a* in einen einzelnen Wert vom Typ *c* „zusammengefaltet“ werden kann.

Finden Sie den Typ und die Definition der entsprechenden Fold-Funktion `foldT` des parametrischen Datentyps **Tree** *a*.

Aufgabe 3.3

Fold-Funktionen

Jeder induktive Datentyp besitzt eine Fold-Funktion, die sich aus der initialen Algebrastruktur des Typs ergibt. Der Typ und die Definitionen dieser Fold-Funktionen ergeben sich dabei allein aus den Typen der Konstruktoren. Beispielsweise ist die Fold-Funktion für den Datentyp **Nat** *a* aus der vorangegangenen Übung wie folgt definiert:

```
foldL : c → (a → c → c) → List a → c
foldL n f Nil = n
foldL n f (Cons x xs) = f x (foldL n f xs)
```

Hierbei entspricht der Typparameter *c* einem Ergebnistyp und die beiden Argumente *n* und *f* den Konstruktoren *Nil* und *Cons*, wobei die Typen der Argumente jeweils Operationen auf dem Ergebnistyp *c* beschreiben, mit dem eine Liste **List** *a* in einen einzelnen Wert vom Typ *c* „zusammengefaltet“ werden kann.

Primitiv rekursive Funktionen auf **List** *a* können durch geeignete Instantiierung des Typparameters *c* und der Argumente *n* und *f* alternativ mittels `foldL` ausgedrückt werden. Drücken Sie die Funktionen `length` und `size` jeweils als Folds über Listen und Bäumen aus.