Friedrich-Alexander-Universität
Technische Fakultät

FAU

Theorie der Programmierung

Übung 05 - der (ungetypte) λ-Kalkül

Leon Vatthauer

2. Juni 2023

Der (ungetypte) λ-Kalkül

Terme und Konventionen

λ-Terme

$$t ::= x \mid t_1 t_2 \mid \lambda x.t \qquad (x \in V)$$

Der (ungetypte) λ-Kalkül

Terme und Konventionen

λ-Terme

$$t ::= x \mid t_1 t_2 \mid \lambda x.t \qquad (x \in V)$$

Konventionen

- Applikation ist *links-assoziativ*: $x\ y\ z = (x\ y)\ z$
- Abstraktion reicht so weit wie möglich (Vgl. Quantoren in GLoIn)
- Aufeinanderfolgende Abstraktionen werden zusammengefasst: $\lambda x.\lambda y.\lambda z.y\ x = \lambda x\ y\ z.y\ x$

Der (ungetypte) λ-Kalkül

Freie Variablen und Substitution

Freie Variablen

Sei t ein λ-Term, $FV(t)$ ist dann definiert durch:

$$\begin{aligned} FV(x) &= \{x\} \quad (\text{für } x \in V) \\ FV(t\ s) &= FV(t) \cup FV(s) \\ FV(\lambda x.t) &= FV(t) \setminus \{x\} \end{aligned}$$

Der (ungetypte) λ-Kalkül

Freie Variablen und Substitution

Freie Variablen

Sei t ein λ-Term, $FV(t)$ ist dann definiert durch:

$$\begin{aligned} FV(x) &= \{x\} \quad (\text{für } x \in V) \\ FV(t\ s) &= FV(t) \cup FV(s) \\ FV(\lambda x.t) &= FV(t) \setminus \{x\} \end{aligned}$$

Substitution

Eine Substitution ist eine Abbildung $\sigma : V_0 \to T(V)$, wobei $V_0 \subseteq V$ (*endliche* Teilmenge).
Die Anwendung einer Substitution auf λ-Terme ist ebenfalls rekursiv definiert:

$$\begin{aligned} x\sigma &= \sigma(x) \\ (t\ s)\sigma &= (t\sigma)\ (s\sigma) \\ (\lambda x.t)\sigma &= \lambda y.(t\sigma') \end{aligned}$$

mit y frisch, also $y \notin FV(\sigma(z))$ für alle $z \in FV(\lambda x.t)$ und $\sigma' = \sigma[x \mapsto y]$

Der (ungetypte) λ-Kalkül

α-Äquivalenz und β-Reduktion

α-Äquivalenz

Zwei Terme t_1, t_2 heißen α-Äquivalent, wenn sie durch Umbenennung gebundener Variablen auseinander hervorgehen. Formal:

$$\lambda x.t =_\alpha \lambda y.t[y/x] \quad \text{wenn } y \notin FV(t)$$

Der (ungetypte) λ-Kalkül

α-Äquivalenz und β-Reduktion

α-Äquivalenz

Zwei Terme t_1, t_2 heißen α-Äquivalent, wenn sie durch Umbenennung gebundener Variablen auseinander hervorgehen. Formal:

$$\lambda x.t =_\alpha \lambda y.t[y/x] \quad \text{wenn } y \notin FV(t)$$

β-Reduktion

Die β-Reduktion modelliert das Ausrechnen einer Funktionsanwendung, z.B: $(\lambda x.3 + x)\ 5 \to_\beta 3 + 5$
Die Einschrittreduktion $\to_\beta$ ist:

$$C((\lambda x.t)\ s) \to_\beta C(t[s/x])$$

Aufgabe 1.1

α-Äquivalenz und β-Reduktion

Entscheiden Sie für jedes der folgenden Paare von λ-Termen, ob die Terme jeweils α-Äquivalent zueinander sind:

(a) $\lambda x\ y.x\ y =^?_\alpha \lambda u\ v.u\ v$

(b) $\lambda x\ y.x\ y =^?_\alpha \lambda u\ v.v\ u$

(c) $(\lambda x.x\ x)(\lambda y.y\ y) =^?_\alpha (\lambda x.x\ x)(\lambda x.x\ x)$

α-Äquivalenz

Zwei Terme t_1, t_2 heißen α-Äquivalent, wenn sie durch Umbenennung gebundener Variablen auseinander hervorgehen. Formal:

$$\lambda x.t =_\alpha \lambda y.t[y/x] \quad \text{wenn } y \notin FV(t)$$

Aufgabe 1.2

α-Äquivalenz und β-Reduktion

Entscheiden Sie in jedem der folgenden Fälle, ob der jeweilige Reduktionsschritt eine zulässige β-Reduktion darstellt:

(a) $(\lambda x\ y\ z.x\ y\ z)(\lambda y.y\ y) \to^?_\beta \lambda y\ z.(\lambda y.y\ y)\ y\ z$

(b) $(\lambda x\ y\ z.x\ y\ z)(y\ y) \to^?_\beta \lambda y\ z.(y\ y)\ y\ z$

(c) $(\lambda x\ y\ z.x\ y\ z)(y\ y) \to^?_\beta \lambda y\ z.(u\ u)\ y\ z$

(d) $(\lambda x\ y\ z.x\ y\ ((\lambda u.u\ x)(y\ y)))\ u\ v \to^?_\beta (\lambda x\ y\ z.x\ y\ ((y\ y)\ x))\ u\ v$

β-Reduktion

Die β-Reduktion modelliert das Ausrechnen einer Funktionsanwendung, z.B: $(\lambda x.3 + x)\ 5 \to_\beta 3 + 5$
Die Einschrittreduktion $\to_\beta$ ist:

$$C((\lambda x.t)\ s) \to_\beta C(t[s/x])$$

Aufgabe 2

We Are Not Anonymous (Functions)

Funktionale Programmiersprachen wie zum Beispiel Haskell oder ML erweitern den λ-Kalkül um verschiedene Konstrukte. Insbesondere werden *Definitionen* verwendet, um Funktionen zu benennen. Beispielsweise können wir die folgenden drei Gleichungen angeben:

$$flip = \lambda f\ x\ y.f\ y\ x \qquad const = \lambda x\ y.x \qquad twice = \lambda f\ x.f\ (f\ x)$$

und sie als - von links nach rechts zu lesende - Reduktionsregeln betrachten, die bei der Auswertung eines λ-Terms angewendet werden können. Um sie von β-Reduktionen zu unterscheiden, werden solche Reduktionen als δ-Reduktionen bezeichnet. Beispielsweise ist mit den obigen Definitionen die folgende Reduktion möglich:

$$(\lambda f.f\ u)\ const \to_\beta const\ u \to_\delta (\lambda x\ y.x)\ u \to_\beta \lambda y.u$$

Hinweis: Eine derartige Reduktion, bei der sowohl β- als auch δ-Schritte vorkommen können, bezeichnen wir ab sofort als $\beta\delta$-Reduktion.

Aufgabe 2.1

We Are Not Anonymous (Functions)

Ermitteln Sie die $\beta\delta$-Normalformen der folgenden Terme:

(a) $flip\ const\ twice$

(b) $twice\ flip$

β-Reduktion

Die β-Reduktion modelliert das Ausrechnen einer Funktionsanwendung, z.B: $(\lambda x.3 + x)\ 5 \to_\beta 3 + 5$
Die Einschrittreduktion $\to_\beta$ ist:

$$C((\lambda x.t)\ s) \to_\beta C(t[s/x])$$

δ-Reduktion

$$flip = \lambda f\ x\ y.f\ y\ x$$
$$const = \lambda x\ y.x$$
$$twice = \lambda f\ x.f\ (f\ x)$$

Beispiel: $(\lambda f.f\ u)\ const \to_\beta const\ u \to_\delta (\lambda x\ y.x)\ u \to_\beta \lambda y.u$

Aufgabe 2.2

We Are Not Anonymous (Functions)

Tatsächlich ist es in den angegebenen Programmiersprachen praktischerweise auch möglich, Funktionsargumente auf der linken Seite einer Funktionsdefinition anzugeben. Die obigen Definitionen würden dann beispielsweise wie folgt geschrieben werden:

$$\begin{aligned} flip\ f &= \lambda x\ y.f\ y\ x \\ const\ x\ y &= x \\ twice\ f\ x &= f\ (f\ x) \end{aligned}$$

Die mit diesen Definitionen verbundenen Reduktionsregeln sind nun nicht mehr offensichtlich; deshalb verwenden wir Großbuchstaben F, X, Y, um Variablen, die in einem Term vorkommen, von Variablen, die durch eine Abstraktion gebunden werden können, zu unterscheiden. Außerdem verwenden wir zur Verdeutlichung explizite Klammerung:

$$\begin{aligned} flip\ F &\to_\delta (\lambda x\ y.f\ y\ x)[F/x] \\ (const\ X)\ Y &\to_\delta x[X/x, Y/y] \\ (twice\ F)\ X &\to_\delta (f\ (f\ x))[X/x, F/f] \end{aligned}$$

Aufgabe 2.2

We Are Not Anonymous (Functions)

Entscheiden Sie, welche der folgenden λ-Terme bezüglich dieser neuen Definitionen $\beta\delta$-Normalformen sind:

(a) $\lambda x.flip\ x$
(b) $\lambda x.const\ x$
(c) $const\ flip\ twice$

Neue δ-Reduktion

$$
\begin{aligned}
flip\ F &\to_\delta (\lambda x\ y.f\ y\ x)[F/x] \\
(const\ X)\ Y &\to_\delta x[X/x, Y/y] \\
(twice\ F)\ X &\to_\delta (f\ (f\ x))[X/x, F/f]
\end{aligned}
$$

Aufgabe 3

Church-Numerale

Funktionale Sprachen fügen weiterhin *eingebaute* Typen (*built-in types*) zum λ-Kalkül hinzu und erlauben auch die Definition von benutzerdefinierten Typen (*user-defined types*). Diese Typen können jedoch prinzipiell allesamt unter Einsatz der sogenannten *Church-Kodierung* direkt als λ-Terme kodiert werden.

Eine natürliche Zahl n wird durch einen λ-Term $\lceil n \rceil$ kodiert, der eine gegebene Funktion f wiederholt n-mal auf einen Startwert a anwendet, was wir informell als n „Iterationen“ von f startend bei a ansehen können:

$$\lceil n \rceil := \lambda f\ a.\ \underbrace{f(f(f(\ldots f}_{n}\ a))) \tag{1}$$

Wir kodieren dies einheitlich wie folgt:

$$\begin{aligned} zero &= \lambda f\ a.a \\ succ\ n &= \lambda f\ a.f\ (n\ f\ a) \end{aligned}$$

$$\begin{aligned} one &= succ\ zero \\ two &= succ\ one \\ three &= succ\ two \\ four &= succ\ three \end{aligned}$$

Aufgabe 3.1

Church-Numerale

Zeigen Sie, dass für jede natürliche Zahl n gilt: $succ\ \lceil n \rceil \rightarrow^*_{\beta\delta} \lceil n+1 \rceil$

Church-Numerale

$$\lceil n \rceil := \lambda f\ a.\underbrace{f(f(f(\ldots f}_{n}\ a))) \tag{1}$$

$$zero = \lambda f\ a.a$$
$$succ\ n = \lambda f\ a.f\ (n\ f\ a)$$

$$\begin{aligned} one &= succ\ zero \\ two &= succ\ one \\ three &= succ\ two \\ four &= succ\ three \end{aligned}$$

Aufgabe 3.2

Church-Numerale

Definieren Sie die Addition und Multiplikation natürlicher Zahlen bezüglich dieser Kodierung. Vervollständigen Sie also die folgenden Definitionen:

$$add\ n\ m = \ldots \qquad\qquad mult\ n\ m = \ldots$$

derart, dass für alle x und y gilt:

$$add\ \lceil x \rceil\ \lceil y \rceil \to^*_{\beta\delta} \lceil x + y \rceil \qquad\qquad mult\ \lceil x \rceil\ \lceil y \rceil \to^*_{\beta\delta} \lceil x \cdot y \rceil$$

Church-Numerale

$$\lceil n \rceil := \lambda f\ a.\underbrace{f(f(f(\ldots f}_{n}\ a))) \tag{1}$$

$$zero = \lambda f\ a.a$$
$$succ\ n = \lambda f\ a.f\ (n\ f\ a)$$

$$\begin{aligned} one &= succ\ zero \\ two &= succ\ one \\ three &= succ\ two \\ four &= succ\ three \end{aligned}$$