Theorie der Programmierung SS 2015

Übungsblatt 2

Rev.8703

Abgabe der Lösungen: Tutorium in der Woche 01.06.-05.06.

Zur Notation

Wir schreiben λ-Terme den folgenden Regeln entsprechend abgekürzt:

- *Applikation ist links-assoziativ.*
  Beispielsweise kürzen wir $((x(yz))u)v$ zu $x(yz)uv$.
- *Abstraktion reicht so weit wie möglich.*
  Beispielsweise kürzen wir $\lambda x.(x(\lambda y.(yx)))$ zu $\lambda x.x(\lambda y.yx)$.
- *Aufeinanderfolgende Abstraktionen werden zusammengefasst.*
  Beispielsweise kürzen wir $\lambda x.\lambda y.\lambda z.yx$ zu $\lambda xyz.yx$.

In Bezug auf Typen fassen wir $\to$ als rechts-assoziativ auf; d.h. wir schreiben $t_1 \to t_2 \to t_3 \to t_4$ anstelle von $t_1 \to (t_2 \to (t_3 \to t_4))$. Schliesslich vereinbaren wir, dass fettgedruckte Typen (beispielsweise **int**) für Typkonstanten stehen.

Übung 1 α-Äquivalenz und β-Reduktion

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

   (a) $\lambda xy.xy =^?_\alpha \lambda uv.uv$

   (b) $\lambda xy.xy =^?_\alpha \lambda uv.vu$

   (c) $\lambda xy.xy =^?_\alpha \lambda yx.yx$

   (d) $(\lambda x.xx)(\lambda y.yy) =^?_\alpha (\lambda x.xx)(\lambda x.xx)$

   (e) $(\lambda x.x(\lambda y.yy)) =^?_\alpha (\lambda x.x(\lambda x.xx))$

   (f) $(\lambda x.x(\lambda y.yx)) =^?_\alpha (\lambda y.y(\lambda y.yy))$

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

   (a) $(\lambda xyz.xyz)(\lambda y.yy) \to^?_\beta \lambda yz.(\lambda y.yy)yz$

   (b) $(\lambda xyz.xyz)(yy) \to^?_\beta \lambda yz.(yy)yz$

   (c) $(\lambda xyz.xyz)(yy) \to^?_\beta \lambda uz.(yy)uz$

   (d) $(\lambda xyz.xyz)(yy) \to^?_\beta \lambda yz.(uu)yz$

   (e) $(\lambda xyz.xyz)(uu) \to^?_\beta \lambda xy.xy(uu)$

   (f) $(\lambda xyz.xy((\lambda u.ux)(yy)))uv \to^?_\beta (\lambda xyz.xy((yy)x))uv$

   (g) $(\lambda xyz.xy((\lambda u.ux)(yy)))uv \to^?_\beta (\lambda xuz.xu((yy)x))uv$

3. Die folgenden λ-Terme haben eindeutige β-Normalformen. Finden Sie diese und geben Sie jeweils eine vollständige Herleitung der Normalform an.
   Hinweis: Die Normalformen sind "eindeutig", wenn wir die Namen gebundener Variablen ignorieren (d.h. *eindeutig modulo α-Äquivalenz*).

   (a) $(\lambda xyz.x(zz)y)(\lambda uv.v)$
   (b) $(\lambda xy.(\lambda zu.y(uz)x)xx)uvu$

Übung 2 We Are Not Anonymous (Functions)

Funktionale Programmiersprachen wie zum Beispiel Haskell oder ML erweitern den λ-Kalül (bzw. getypte Fragmente des λ-Kalküls) um verschiedene Konstrukte. Insbesondere werden *Definitionen* verwendet, um Funktionen zu *benennen*. Beispielsweise können wir die folgenden drei Gleichungen angeben:

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

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 ("δ" wie "Definition") bezeichnet. Beispielsweise ist mit den obigen Definitionen die folgende Reduktion möglich:

$$(\lambda f.fu)\mathit{const} \to_\beta \mathit{const}\ u \to_\delta (\lambda x\ y\ .\ x)u \to_\beta \lambda y\ .\ u$$

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

   (a) $\mathit{flip}\ \ \mathit{const}\ \ \mathit{twice}$
   (b) $\mathit{twice}\ \ \mathit{flip}$

2. 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:

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

   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:

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

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

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

Übung 3 δ-Reduktion falsch gemacht

Der Präprozessor von C erlaubt das Definieren von *Makros* mit Variablen mittels der folgenden Syntax:

#define SOME_MACRO(X1,...,Xn) BODY

Die Expansion eines solchen Makros entspricht einer Anwendung der folgendenen simplen Reduktionsregel:

$$SOME\_MACRO(X1,...,Xn) \rightarrow_0 BODY$$

wobei *BODY* als Term ohne Variablenbindung, also als Term im Sinne des Kapitels über Termersetzungssysteme angesehen wird, so dass Substitution in *BODY* einfach Variablen durch Terme ersetzt. Das heißt, dass nun – im Gegensatz zu den δ-Regeln aus Übung 2.2 – keine komplexe Umbenennung gebundener Variablen stattfindet. Betrachten wir also das folgende Makro, das die Werte von zwei **int**-Variablen vertauscht:

#define SWAP_INT(x,y) **do**{**int** aux = x; x = y; y = aux;} **while**(0)

Zeigen Sie anhand eines Beispiels, dass *SWAP_INT* aufgrund der naiven Substitution in der Expansion von C-Makros auf subtile Weise nicht korrekt funktionsfähig ist.

Übung 4 Church-Kodierung

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.

1. Boolesche Wahrheitswerte werden als λ-Terme wie folgt definiert:

   $$\begin{aligned} true &= \lambda x\ y\ .\ x \\ false &= \lambda x\ y\ .\ y \\ if\_then\_else &= \lambda b\ x\ y\ .\ b\ x\ y \end{aligned}$$

   (a) Zeigen Sie, dass für alle λ-Terme s und t gilt:

   $$if\_then\_else\ \ true\ \ s\ \ t \rightarrow^*_{\beta\delta} s \qquad\qquad if\_then\_else\ \ false\ \ s\ \ t \rightarrow^*_{\beta\delta} t$$

   (b) Vervollständigen Sie die folgenden Funktionsdefinitionen, so dass sie (unter normaler Reduktion) boolesche Negation, Konjunktion und Disjunktion berechnen:

   $$\begin{aligned} not\ b &= ... \\ and\ b1\ b2 &= ... \\ or\ \ b1\ b2 &= ... \end{aligned}$$

   Notation: Von nun an schreiben wir "**if** s **then** t **else** u" anstelle von "$if\_then\_else\ \ s\ \ t\ \ u$".

2. *Paare* von Elementen können mit der folgenden Kodierung gebildet bzw. wieder zerlegt werden:

   $$\begin{aligned} pair\ a\ b &= \lambda select\ .\ \ select\ \ a\ b \\ fst\ \ p &= p\ (\lambda x\ y\ .\ x) \\ snd\ p &= p\ (\lambda x\ y\ .\ y) \end{aligned}$$

Schreiben Sie eine Funktion *swap*, die die beiden Komponenten eines gegebenen Paares austauscht. Das heißt, wir fordern, dass für alle λ- Terme s und t die folgenden Reduktionen gelten:

$$fst\,(swap\ (pair\ s\ t)) \to^*_{\beta\delta} t \qquad\qquad snd(swap\ (pair\ s\ t)) \to^*_{\beta\delta} s$$

3. 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:

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

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

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

(b) Definieren Sie die Addition, Multiplikation und Exponentiation natürlicher Zahlen bezüglich dieser Kodierung. Vervollständigen Sie also die folgenden Definitionen:

$$\begin{aligned} add\ \ n\ m &= ... \\ mult\ n\ m &= ... \\ exp\ \ n\ m &= ... \end{aligned}$$

derart, dass für alle x und y gilt:

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

(c) Definieren Sie eine Funktion *isZero*, so dass:

$$isZero\ \lceil 0 \rceil \to^*_{\beta\delta} true \qquad\qquad isZero\ \lceil n+1 \rceil \to^*_{\beta\delta} false$$

Notation: Von nun an schreiben wir $s + t$ und $s * t$ anstelle von $add\ s\ t$ und $mult\ s\ t$.

Übung 5 Subtraktion

Das Dekrementieren von Church-Numeralen ist erheblich komplizierter; angeblich benötigte Alonzo Church einige Jahre, um herauszubekommen, dass (und auf welche Weise) es möglich ist. Allerdings gab es zu dieser Zeit noch keine Computer, so dass Church auch kein Programmierer im klassischen Sinne war.

$$pred\ n = fst\ (n\ (\lambda p\ .\ pair\ (snd\ p)\ (succ\ (snd\ p)))\ (pair\ zero\ zero))$$

1. Zeigen Sie, dass $pred\ \lceil 3 \rceil \to^*_{\beta\delta} \lceil 2 \rceil$.
2. Erläutern Sie in eigenen Worten die Funktionsweise des durch *pred* implementierten Algorithmus.

3. Vervollständigen Sie die folgenden Funktionsdefinitionen:

$$\begin{array}{lll} \mathit{sub}\ n\ m = \ldots & // & n - m \\ \mathit{eq}\ \ n\ m = \ldots & // & n == m\,? \\ \mathit{le}\ \ n\ m = \ldots & // & n \leq m\,? \\ \mathit{lt}\ \ n\ m = \ldots & // & n < \ m\,? \end{array}$$

Notation: Von nun an schreiben wir $s - t$, $s == t$, $s \leq t$ und $s < t$ anstelle von $\mathit{sub}\ s\ t$, $\mathit{eq}\ s\ t$, $\mathit{le}\ s\ t$ und $\mathit{lt}\ s\ t$.

Übung 6 Rekursive Definitionen

In den meisten funktionalen Programmiersprachen sind *rekursive* Funktionsdefinitionen zulässig, das heißt, die definierte Funktion darf auf der rechten Seite einer solchen Funktionsdefinition vorkommen. Rekursive Funktionsdefinitionen entsprechen – wie in Übung 2 – δ-Reduktionen.

1. Wir betrachten die folgende rekursive Funktion:

$$\mathit{fact}\ \ n = \textbf{if}\ n \leq \lceil 1 \rceil\ \textbf{then}\ \lceil 1 \rceil\ \textbf{else}\ n * (\mathit{fact}\ \ (n - \lceil 1 \rceil))$$

   Zeigen Sie, dass $\mathit{fact}\ \lceil 4 \rceil \rightarrow^*_{\beta\delta} \lceil 24 \rceil$.

   Verwenden Sie, um die Derivationen abzukürzen, die Tatsache, dass $\lceil n \rceil * \lceil m \rceil \rightarrow^*_{\beta\delta} \lceil nm \rceil$ etc.

2. Schreiben Sie eine rekursive Funktion $\mathit{odd}\ \lceil n \rceil$, die *true* als Ergebnis liefert, wenn n ungerade ist, und *false* andernfalls.

3. Schreiben Sie eine rekursive Funktion *halve*, so dass:

   $(\lceil 2 \rceil * \mathit{halve}\ \lceil n \rceil) + \textbf{if}\ \mathit{odd}\ \lceil n \rceil\ \textbf{then}\ \lceil 1 \rceil\ \textbf{else}\ \lceil 0 \rceil\ \rightarrow^*_{\beta\delta} \lceil n \rceil$.

Übung 7 Wer braucht schon Rekursion?

Rekursive Definitionen sind komfortabel, aber nicht notwendig: Bereits im reinen λ-Kalkül ist es möglich, Fixpunkt-Kombinatoren zu definieren (wie z.B. den Y-Kombinator). Es sei *fix* ein beliebiger Fixpunkt-Kombinator, d.h. es gelte $\mathit{fix}\ \ f \rightarrow^*_{\beta\delta} f\,(\mathit{fix}\ f)$.

1. Die folgende Definition entspricht der Funktion *fact* aus Übung 6; jedoch wurde die Rekursion durch einen Fixpunkt ersetzt:

$$\mathit{fact'} = \mathit{fix}\ \ (\lambda \mathit{myself}\ n\ .\ \ \textbf{if}\ \ n \leq \lceil 1 \rceil\ \textbf{then}\ \lceil 1 \rceil\ \textbf{else}\ n * (\mathit{myself}\,(n - \lceil 1 \rceil)))$$

   Zeigen Sie, dass $\mathit{fact'}\ \lceil 3 \rceil \rightarrow^*_{\beta\delta} \lceil 6 \rceil$

2. Schreiben Sie unter Verwendung von *fix* nun nicht-rekursive Versionen aller übrigen Definitionen aus Übung 6.

Übung 8 Typprüfung simpler Terme

Wir lesen $\Gamma \vdash s : \alpha$ wie folgt: "Dem λ-Term s lässt sich unter der Annahme, dass eventuellen freien Variablen von s die durch Γ gegebenen Typen zugeordnet sind, der Typ α zuweisen". Zeigen Sie, dass die folgenden Aussagen zutreffen, indem Sie jeweils eine korrekte Typinferenz angeben.

1. $x : \mathbf{int}, add : \mathbf{int} \to \mathbf{int} \to \mathbf{int} \vdash \lambda y.add\ x\ (add\ x\ y) : \mathbf{int} \to \mathbf{int}$.
2. $length : \mathbf{string} \to \mathbf{int}, name : \mathbf{person} \to \mathbf{string} \vdash \lambda x.length\ (name\ x) : \mathbf{person} \to \mathbf{int}$
3. $\vdash \lambda f\ x\ y\ .\ f\ y\ x : (\mathbf{int} \to \mathbf{char} \to \mathbf{string}) \to (\mathbf{char} \to \mathbf{int} \to \mathbf{string})$
4. $\vdash \lambda f\ x\ y\ .\ f\ y\ x : (\alpha \to \beta \to \gamma) \to (\beta \to \alpha \to \gamma)$, für alle Typen α, β and γ.
5. $\vdash \lambda x\ y\ .\ x : \alpha \to \beta \to \alpha$, für alle Typen α, β and γ.

Übung 9 Untypisierbare Terme

Verwenden Sie das Inversionslemma um zu zeigen, dass:

1. $\nvdash \lambda x.xx : \alpha$, für jeden Typ α.
2. $y : \mathbf{char} \nvdash \lambda x.yx : \alpha$, für jeden Typ α.

Übung 10 Inferenz von Prinzipaltypen

Es sei t ein λ-Term und Γ ein Kontext; wir sagen, dass α der *Prinzipaltyp* (engl. *principal type*) von t ist, wenn i) $\Gamma \vdash t : \alpha$ und ii) jeder Typ β mit $\Gamma \vdash t : \beta$ eine *Instanz* von α ist, d.h. wenn jedes solche β sich durch Substitution von Typvariablen aus α erzeugen lässt. Beispielsweise ist $a \to b \to a$ (für Typvariablen a und b) der Prinzipaltyp von $\lambda x\ y\ .\ x$ (vergleiche: Übung 8.5). Leiten Sie den Prinzipaltyp der folgenden λ-Terme in dem jeweils gegebenen Kontext her:

1. $\Gamma = \emptyset$, $t = \lambda f\ g\ x\ .\ f\ (g\ x)$.
2. $\Gamma = \{add : \mathbf{int} \to \mathbf{int} \to \mathbf{int}, length : \mathbf{string} \to \mathbf{int}\}$, $t = \lambda x\ .\ add\ (length\ x)$

Übung 11 Programme sind Beweise?!

Das zur Typinferenz symmetrische Problem ist das Problem der *type inhabitation*, d.h. das Problem, einen λ-Term eines gegebenen Typs zu finden, falls ein solcher Term existiert. Im Folgenden bezeichnen p, q und r Typvariablen.

1. Finden Sie für jeden der folgenden Typen α einen λ-Term s, so dass $\vdash s : \alpha$.
   (a) $p \to p$
   (b) $p \to (q \to p)$
   (c) $(p \to (q \to r)) \to (p \to q) \to p \to r$
   (d) $((p \to q) \to r) \to q \to r$

2. Prüfen Sie (beispielsweise mittels Wahrheitstafeln), ob die Typen der vorherigen Teilaufgabe – als *aussagenlogische Formeln* interpretiert – aussagenlogische Tautologien sind. Ist das Ergebnis Ihrer Prüfung angesichts der Curry-Howard-Korrespondenz eine Überraschung?

3. Verwenden Sie die Curry-Howard-Korrespondenz, um zu zeigen, dass es keinen λ-Term *coerce* gibt, so dass $\vdash$ *coerce* $: a \to b$ (für Typvariablen a und b).

Übung 12 *Die unvermeidliche Ackermann-Übung (25 Punkte)*

Wir betrachten die folgende rekursive Funktion:

$$\begin{aligned} \mathit{ack}\ m\ n = \textbf{if}\ m = \lceil 0 \rceil\ \textbf{then}\ & n + \lceil 1 \rceil \\ \textbf{else}\ \ \textbf{if}\ n = \lceil 0 \rceil\ \textbf{then}\ & \mathit{ack}\ (m - \lceil 1 \rceil)\ \lceil 1 \rceil \\ \textbf{else}\ & \mathit{ack}\ (m - \lceil 1 \rceil)\ (\mathit{ack}\ m\ (n - \lceil 1 \rceil)) \end{aligned}$$

1. Zeigen Sie, dass $\mathit{ack}\ \lceil 1 \rceil\ \lceil 1 \rceil \to^*_{\beta\delta} \lceil 3 \rceil$.

   Hinweis: Sie können annehmen, dass die booleschen und arithmetischen Funktionen sich verhalten, wie zu erwarten ist (beispielsweise, dass $\lceil 5 \rceil - \lceil 3 \rceil \to^*_{\beta\delta} \lceil 2 \rceil$, etc).

2. Geben Sie einen λ-Term t an, so dass für jeden Fixpunkt-Kombinator *fix* der Term *fix* t dieselbe Funktion berechnet wie *ack*.

3. Es sei Γ der folgende Kontext (wobei wir die Konventionen von Übung 4 und Übung 5 beachten):

   $$\begin{aligned} \mathit{if\_then\_else} &: \textbf{bool} \to \textbf{nat} \to \textbf{nat} \to \textbf{nat}, & \mathit{add} &: \textbf{nat} \to \textbf{nat} \to \textbf{nat}, & \lceil 0 \rceil &: \textbf{nat}, \\ \mathit{eq} &: \textbf{nat} \to \textbf{nat} \to \textbf{bool}, & \mathit{sub} &: \textbf{nat} \to \textbf{nat} \to \textbf{nat}, & \lceil 1 \rceil &: \textbf{nat}. \end{aligned}$$

   Geben Sie den Prinzipaltyp Ihres Terms t aus der vorherigen Teilaufgabe an.

4. Finden Sie einen Typ α, so dass

   $$\Gamma, \mathit{fix} : \alpha \vdash \mathit{fix}\ \ t : \textbf{nat} \to \textbf{nat} \to \textbf{nat}$$

   Hinweis: Verwenden Sie das Weakening-Lemma um zu vermeiden, dass Sie erneut den Typ von t herleiten müssen!

Übung 13 *Quiz* *(15 Punkte)*

Entscheiden Sie, ob die folgenden Aussagen zutreffend sind, und begründen Sie Ihre Antworten!

1. Der ungetypte λ-Kalkül ist stark normalisierend.

2. Es existiert ein Fixpunkt-Kombinator *fix* (d.h. ein λ-Term *fix*, so dass für alle Terme f gilt: *fix* $f \to^*_{\beta\delta} f\,(\mathit{fix}\ f)$) für den der Typ $((a \to a) \to a \to a) \to a \to a$ hergeleitet werden kann.

3. Wenn ein λ-Term schwach normalisierend ist, so wird normale (d.h. leftmost-outermost-) Reduktion zu seiner Normalform führen.