Übung 7

Alexander Mattick Kennung: qi69dube

Kapitel 1

30. Juni 2020

1 aufgabe 3

Das problem der Subjektexpansion ist typischerweise, dass untypisierbare Terme durch beta-Reduktion gelöscht werden könnten.

Hier ist jedoch ist festgelegt, dass sowohl s als auch t typiserbar ist.

Sei $\Gamma[v \mapsto \alpha]$

$t = a : b$

$s = (\lambda v.v)a$

$$\dfrac{\dfrac{\dfrac{}{\Gamma[v \mapsto \alpha] \vdash v : \alpha}}{\Gamma \vdash \lambda v.v : \alpha \to \alpha} \qquad \dfrac{}{\Gamma \vdash a : \alpha}}{\Gamma \vdash (\lambda v.v)a}$$

Im Fall t ist das a an keine Weiteren typattribute gebunden (Es "weis" nichts mehr davon, dass es evtl. in einem Kontext s gestanden ist, in dem es relevant war ein α zu sein).

Bei s ist es jedoch, aufgrund des Kontexts gezwungenermaßen so, dass man $a : \alpha$ zuweist.

$S = \lambda xyz.xz(yz) : \alpha \to (\beta \to \gamma) \to (\alpha \to \beta \to (\alpha \to \gamma))$

- $PT(\emptyset, \lambda xyz.xz(yz), \alpha)$
  - $PT(x : a_1, \lambda yz.xz(yz), a_2)$
    - $PT(\{x : a_1, y : a_3\}, \lambda z.xz(yz), a_4)$
      - $PT(\underbrace{\{x : a_1, y : a_3, z : a_5\}}_{\Gamma}, xz(yz), a_6)$
        - $PT(\Gamma, xz, a_7 \to a_6)$
          - $PT(\Gamma, x, a_8 \to a_7 \to a_6)$
          - $PT(\Gamma, z, a_8)$
        - $PT(\Gamma, yz, a_7)$
          - $PT(\Gamma, y, a_9 \to a_7)$
          - $PT(\Gamma, z, a_9)$
      - $\{a_5 \to a_6 \doteq a_4\}$
    - $\{a_3 \to a_4 \doteq a_2\}$
  - $\{a_1 \to a_2 \doteq \alpha\}$

Der letzte schritt wurde bei jedem Teilbaum aus Formatierungsgründen weggelassen.

$\{a_1 \to a_2 \doteq \alpha, a_3 \to a_4 \doteq a_2, a_5 \to a_6 \doteq a_4, a_8 \to a_7 \to a_6 \doteq a_1, a_8 \doteq a_5, a_3 \doteq a_8 \to a_7, a_5 \doteq a_9\}$

elim $\{a_1 \to a_2 \doteq \alpha, a_3 \to a_4 \doteq a_2, a_9 \to a_6 \doteq a_4, a_8 \to a_7 \to a_6 \doteq a_1, a_8 \doteq a_9, a_3 \doteq a_8 \to a_7, a_5 \doteq a_9\}$

elim $\{(a_8 \to a_7 \to a_6) \to a_8 \to a_7 \to a_9 \to a_6 \doteq \alpha, a_3 \to a_4 \doteq a_2, a_9 \to a_6 \doteq a_4, a_8 \to a_7 \to a_6 \doteq a_1, a_8 \doteq a_9, a_3 \doteq a_8 \to a_7, a_5 \doteq a_9\}$

elim $\{(a_8 \to a_7 \to a_6) \to a_8 \to a_7 \to a_8 \to a_6 \doteq \alpha, a_3 \to a_4 \doteq a_2, a_9 \to a_6 \doteq a_4, a_8 \to a_7 \to a_6 \doteq a_1, a_8 \doteq a_9, a_3 \doteq a_8 \to a_7, a_5 \doteq a_9\}$

$\alpha \doteq (a_7 \to a_6) \to (a_9 \to a_8) \to (a_9 \to a_6)$

$K = true = \lambda xy.x : \alpha \to (\beta \to \alpha)$

$S(true) = \lambda yz.z$

2 4

1.

Nein:

$$\underbrace{((a \to a) \to a \to a)}_{1.Eingabe} \underbrace{\to a}_{2.Eingabe} \underbrace{\to a}_{ausgabe}$$

Die erste eingabe verlangt also irgendeine Funktion f mit 2 Argumenten $\underbrace{(a \to a)}_{1.Argument} \underbrace{\to a}_{2.Argument} \to a$

z.B. $\lambda fa.fa$.

Sprich wir haben eine funktion und eine anwendung als Argumente.

Daraus folgt, dass $fix\ f : a \to a$ als typ hat (man braucht noch ein a, dass man einsetzen kann, das $((a \to a) \to a \to a)$ ist f).

Wenn es ein fix gibt, dass diese typisierung besitzt, dann muss, wenn man $fix\ f : (a \to a)$ in fix einsetzt immernoch eine gültige typisierung entstehen. Sei $\Gamma = \{f = ((a \to a) \to a \to a), fix = ((a \to a) \to a \to a) \to a \to a\}$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : ((a \to a) \to a \to a)} \quad \dfrac{}{\Gamma \vdash fix : ((a \to a) \to a \to a) \to a \to a}}{\Gamma \vdash \underbrace{fix\ f : a \to a}_{erste\ typisierung}} \quad \dfrac{}{\Gamma \vdash f : (a \to a) \to (a \to a)}}{\Gamma \vdash \underbrace{f(fix\ f) : a \to a}_{zweite\ typisierung}}$$ Man kann

also beide (f (fix f) und fix f) gleich typisieren, also ist es kein widerspruch, somit existiert ein Term

2.

$\lceil n \rceil = \lambda fa.\underbrace{f \dots f}_{n}\ a$

ja: Jedes f wird auf eine a/f kombination angewandt:

$\underbrace{f(\underbrace{f(\underbrace{f(a)}_{erste\ Anwendung}}_{zweite\ Anwendung}))}_{Dritte\ Anwendung}$.

Damit dies funktioniert, muss f den gleichen Ein und Ausgabewert haben. Ebenso muss a den Eingabewert von f haben, damit die erste Anwendung funktioniert.

Somit erhalten wir $\lambda fa.\underbrace{f \dots f}_{n}\ a : \underbrace{(t \to t)}_{=f} \underbrace{\to t}_{=a} \to t$

3.

Nein:

Sei $n = 2$:

das erste K $\lambda xy.x$ muss im x den gleichen typen haben, wie a (sonst würde)$(\lambda fa.ffa)K = \lambda a.KKa$ bereits nicht typchecken, weil das erste Argument bereits nicht ausführbar wäre.

Der Typ vom ersten Argument von K ist somit "fixiert".

Das zweite Argument ist frei b:

$$\dfrac{\dfrac{\dfrac{}{\{a:\beta\} \vdash K : \gamma} \quad \dfrac{}{\{a:\beta\} \vdash K : \gamma \to \beta \to a}}{\{a:\beta\} \vdash KK : \beta \to a} \quad \dfrac{}{\{a:\beta\} \vdash a : \beta}}{\dfrac{\{a:\beta\} \vdash KKa : a}{\vdash \lambda a.KKa : \beta \to a}}$$

Die Beiden Typen, die K zugewiesen werden $\gamma \doteq \gamma \to \beta \to a$ sind durch occurs nicht unifizierbar. Somit gibt es keine solche typisierung.

Dies entsteht dadurch, dass jedes folgende K die übrigen argumente vom vorherigen K "mitnehmen" muss.

Da jedes argument jedoch eine Funktion ist die mehr und mehr argumente Erhält, kann es keine Typisierung im einfach getypten lambda-kalkül geben.

3 5

1.

- Cons True (Cons True Nil): boolean
- Cons True (Cons 35 Nil): Nicht typisierbar, da a und listentyp List a von Cons den gleichen Typ a besitzen müssen (währe hier Int/bool)
- Cons True: List Bool $\to$ List Bool, da als zweites argument noch ein List Bool zum anhängen erwartet wird, um die "echte" List zu erhalten
- Cons Nil (Cons (Cons 35 Nil) Nil): List (List Int), typchecked: hier haben die zwei äußere Nil den Typen Nil:List(List a) und das innere Nil bei der 35 den typ Nil:List Int.
- Cons Nil (Cons 35 Nil): List (List Int), gleicher grund, wie beim vorletzten: Typ des inneren Nil ist Int, Typ des äußeren Nil ist List Int.

2.

```
length::List a-> Nat
```

```
length Nil = 0
length (Cons x y) = 1+ length y
snoc::List a-> a->List a
snoc Nil x = Cons x Nil
snoc (Cons x y) z = Cons x (snoc y z)
reverse::List a-> List a
reverse (Cons x y)= snoc (reverse y) x
reverse Nil = Nil
drop::a->List a-> List a
drop x Nil = Nil
drop x (Cons _ y) = if x==z then drop x y else Cons z (drop x y)
elem::a->List a->Bool
elem x Nil =False
elem x (Cons z y) = if x==z then True else elem x y
maximum::List Nat->Nat
maximum Nil = 0
maximum (Cons x y) = if maximum y> x then maximum y else x
```