Tutorium 07: Typisierung

Paul Brinkmeier

03. Dezember 2019

Tutorium Programmierparadigmen am KIT

Heutiges Programm

Programm

- ÜBs 5 und 6
- ullet Typisierter λ -Kalkül
- Einführung in Prolog

• Zwischenschritte beim SKI-Kalkül müssen *nicht* angegeben werden

- Zwischenschritte beim SKI-Kalkül müssen nicht angegeben werden
- Nicht mehr relevante Aufgaben → kommt auf Vorlesung an
 - Alles, was in der Vorlesung angesprochen wird, ist relevant

- Zwischenschritte beim SKI-Kalkül müssen nicht angegeben werden
- Nicht mehr relevante Aufgaben → kommt auf Vorlesung an
 - Alles, was in der Vorlesung angesprochen wird, ist relevant
 - Bis auf Z-Folien

- Zwischenschritte beim SKI-Kalkül müssen nicht angegeben werden
- Nicht mehr relevante Aufgaben → kommt auf Vorlesung an
 - Alles, was in der Vorlesung angesprochen wird, ist relevant
 - Bis auf Z-Folien
- \bullet Freie Variablen sind relevant für $\alpha\text{-}\mathsf{Aquivalenz}$
 - Kommt aber so in Klausuraufgaben nicht vor

- Zwischenschritte beim SKI-Kalkül müssen nicht angegeben werden
- Nicht mehr relevante Aufgaben → kommt auf Vorlesung an
 - Alles, was in der Vorlesung angesprochen wird, ist relevant
 - Bis auf Z-Folien
- Freie Variablen sind relevant für α -Äquivalenz
 - Kommt aber so in Klausuraufgaben nicht vor
- "Standardbibliothek" für λ -Kalkül
 - Alles von den ÜBs
 - Sonstiges nur mit Quellenangabe
 - Bspw. ProPa Folie X, ProPa Klausur SS18 Seite Y

Übungsblatt 5

1.5 — β -Reduktion

Gegeben war:

$$(\lambda a.a) \; (\lambda b.b) \; ((\lambda c.c) \; ((\lambda d.d) \; (\lambda e.e) \; (\lambda f.f))) \; \lambda g.g \; ((\lambda h.h) \; (\lambda i.i))$$

- Vorgehensweise: Redexe markieren, Termliste durchsuchen
- Häufigster Fehler: $((\lambda h.h) (\lambda i.i))$ kann man in $\lambda g.g$ nicht reinziehen (Funktionsaufrufe sind linksassoziativ)

$$pair = \lambda a.\lambda b.\lambda f.f \ a \ b$$

$$pair \ c_{42} \ c_{100} \ \stackrel{2}{\Longrightarrow} \ \lambda f.f \ c_{42} \ c_{100}$$

- — Destructuring/Pattern Matching/Fallunterscheidung durch
 Aufruf einer Funktion mit den Elementen des Tupels
- Wie bei den Listen von letzter Woche

$$fst = \lambda p.p (\lambda a.\lambda b.a)$$

 $snd = \lambda p.p (\lambda a.\lambda b.b)$

 fst/snd ruft Tupel mit Funktion auf, die nur ihr erstes/zweites Argument zurückgibt

Geben Sie next an, sodass next (n, m) = (m, m + 1).

Geben Sie next an, sodass next (n, m) = (m, m + 1).

$$next = \lambda p.(p (\lambda n.\lambda m.pair m (succ m)))$$

In Haskell: next (n, m) = (m, succ m)

Geben Sie next an, sodass next (n, m) = (m, m + 1).

$$next = \lambda p.(p(\lambda n.\lambda m.pair\ m\ (succ\ m)))$$

In Haskell: next (n, m) = (m, succ m)

$$next = \lambda p.pair (snd p) (succ (snd p))$$

In Haskell: next p = (snd p, succ (snd p))

$$pred = \lambda n.fst (n next (pair c_0 c_0))$$

 $sub = \lambda m.\lambda n.n pred m$

- next wird n-mal auf (0,0) angewendet, erhöht immer das zweite Argument und speichert eine Kopie davon \rightsquigarrow im letzten Schritt ist das Tupel (n-1,n), fst liefert also n-1
- sub nimmt einfach n-mal den Vorgänger von m

Übungsblatt 6

Typsystem für λ -Terme

Für Variablen
$$t$$
:
$$\frac{\Gamma(t) = \sigma \quad \sigma \succeq \tau}{\Gamma \vdash t : \tau} \text{VAR}$$

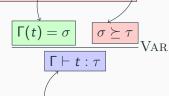
Für Aufrufe
$$f$$
 a:
$$\frac{\Gamma \vdash f : \phi \to \alpha \quad \Gamma \vdash a : \phi}{\Gamma \vdash f \ a : \alpha} APP$$

Für Funktionsterme
$$\lambda p.b$$
:
$$\frac{\Gamma, p : \pi \vdash b : \rho}{\Gamma \vdash \lambda p.b : \pi \to \rho} ABS$$

- Γ ist ein Typkontext
- D.h. *Liste* von Variable/Typ-Tupeln
- Bspw. x : char, y : int
- **Vorsicht**: Γ ist keine Menge, denn Γ hat eine Reihenfolge!

Var

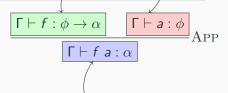
- "Der Typkontext Γ enthält einen Typ σ für t"
- " σ kann mit τ instanziiert werden"



- dann gilt:
- "Variable t hat im Kontext Γ den Typ au "
- $\sigma \succeq \tau \leadsto$ " σ hat τ s Struktur und ist (mind.) allgemeiner"
 - $\operatorname{int} \to \operatorname{int} \succeq \operatorname{int} \to \operatorname{int}$
 - $\forall \alpha. \alpha \to \alpha \succeq \text{int} \to \text{int}$
 - $\alpha \to \alpha \succeq \text{int} \to \text{int}$
 - int \rightarrow int $\not\succeq \forall \alpha.\alpha \rightarrow \alpha$

App

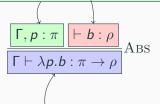
- \Box , f ist im Kontext Γ eine Funktion, die ϕ s auf α s abbildet \Box
- "a ist im Kontext Γ /ein Term des Typs ϕ "



- dann gilt:
- "a eingesetzt in f ergibt einen Term des Typs α "

Abs

- "Damit *b* als Funktion von *p* typisierbar ist..."
- "... müssen wir den Typ von p in den Kontext einfügen"



- dann gilt:
- " $\lambda p.b$ ist eine Funktion, die π s auf ρ s abbildet"

3 — λ -Terme und ihre Typen

Geben sie für jeden Typ der unten stehenden Tabelle all die Terme aus A - F an, die diesen Typ haben können.

- Zeigt, dass *D* den vierten Typ haben kann.
- $D = \lambda x. \lambda y. y (x y)$
- Typ: $((\alpha \to \beta) \to \alpha) \to (\alpha \to \beta) \to \beta$

3 — λ -Terme und ihre Typen

Geben sie für jeden Typ der unten stehenden Tabelle all die Terme aus A - F an, die diesen Typ haben können.

- Zeigt, dass D den vierten Typ haben kann.
- $D = \lambda x. \lambda y. y (x y)$
- Typ: $((\alpha \to \beta) \to \alpha) \to (\alpha \to \beta) \to \beta$
- Ansatz (unten links: leerer Kontext):

$$\frac{\dots \vdash \dots}{\vdash \lambda x. \lambda y. y (x y) : ((\alpha \to \beta) \to \alpha) \to (\alpha \to \beta) \to \beta} ABS$$

4 — Typ-Prüfung

- Sehr gute Übungsaufgabe zum Thema
- Abgegeben von ca. 3 Leuten :(
- Wenn ihr mir die Aufgabe aufs nächste Blatt (oder per Mail schreibt), korrigiere ich sie euch noch zur Übung

Einführung in Prolog

Prolog — Umgebung

- Prolog ist eine Programmiersprache, wenn auch eine "komische"
- → gut wird man durch Übung
- Zum Üben:
 - SWI-Prolog gängige Prolog-Umgebung
 - SWISH SWI-Prolog Web-IDE zum Testen
 - VIPR, VIPER PSE-Tools des IPD, auf der Seite der Ubung verlinkt

- Erste Computerprogramme: Pragmatischerweise imperativ
 - lat. imperare: befehlen
 - "Computer, tu dies, dann das, überspringe 5 Anweisungen"
 - ullet pprox Turing-Maschine
 - \rightsquigarrow einfacher zu implementieren als " λ -Maschine"
- Nächster Schritt: Strukturierte/Prozedurale Programmierung
 - "Go To Statement Considered Harmful" (Edsger Dijkstra)
 - ullet pprox if, while, for, Prozeduren
- "Große" Programme (≥ 10 kLoC) → Objektorientierte Prog.
 - Objekte + Design Patterns, um Komponenten abzugrenzen
 - Letztlich "syntactical sugar" für imperative Programme

- Imperative Sprachen:
 - Rein imperativ ≈ Programm ist Liste von Befehlen (MIMA)
 - Strukturiert ≈ Programm ist Liste von Prozeduren (C)
 - ullet Objektorientiert pprox Programm ist gegeben durch das Verhalten von Objekten (Java)
- Gegenstück?

- Imperative Sprachen:
 - Rein imperativ ≈ Programm ist Liste von Befehlen (MIMA)
 - Strukturiert ≈ Programm ist Liste von Prozeduren (C)
 - Objektorientiert ≈ Programm ist gegeben durch das Verhalten von Objekten (Java)
- Gegenstück? "Deklarative" Sprachen
 - lat. declarare: kenntlich machen, erklären
 - "Computer, so kannst du das Problem lösen; mach mal"
 - Bspw. fibs =

```
0:1: zipWith (+) (take 1 fibs) (take 2 fibs)
```

- Imperative Sprachen:
 - Rein imperativ ≈ Programm ist Liste von Befehlen (MIMA)
 - Strukturiert ≈ Programm ist Liste von Prozeduren (C)
 - Objektorientiert ≈ Programm ist gegeben durch das Verhalten von Objekten (Java)
- Gegenstück? "Deklarative" Sprachen
 - lat. declarare: kenntlich machen, erklären
 - "Computer, so kannst du das Problem lösen; mach mal"
 - Bspw. fibs =

```
0:1: zipWith (+) (take 1 fibs) (take 2 fibs)
```

• Irgendwie schwierig in Hardware zu machen

- Funktionale Programmierung:
 - Programm ist Baum von Funktionsaufrufen
 - "Teile-und-Herrsche": Problemlösung durch aufteilen in gelöste Teilprobleme
 - Unterste Ebene (Prelude) ist imperativ implementiert

- Funktionale Programmierung:
 - Programm ist Baum von Funktionsaufrufen
 - "Teile-und-Herrsche": Problemlösung durch aufteilen in gelöste Teilprobleme
 - Unterste Ebene (Prelude) ist imperativ implementiert
- Nächsthöhere Abstraktion:
 - Implementiere "Lösungs-Maschine" imperativ
 - Problem in Prädikatenlogik darstellen, Maschine löst das
 - Bspw. X ist Voraussetzung für PSE, löse nach X.
 - → logische Programmierung

Modulhandbuch

```
% modulhandbuch.pl
requires(pse, opruefung).
requires(pse, swt1).
requires (opruefung, la1).
requires (opruefung, gbi).
requires(opruefung, programmieren).
semester(pse, 3).
semester(swt1, 2).
semester(gbi, 1).
semester(programmieren, 1).
semester(la1, 1).
```