

Tutorium 07: Prolog-Basics

Paul Brinkmeier

03. Dezember 2019

Tutorium Programmierparadigmen am KIT

Heutiges Programm

- ÜBs 5 und 6
- Typisierter λ -Kalkül
- Einführung in Prolog
- Aufgaben zu Prolog

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

Fragen von letzter Woche

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

Fragen von letzter Woche

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

Fragen von letzter Woche

- Zwischenschritte beim SKI-Kalkül müssen *nicht* angegeben werden
- Nicht mehr relevante Aufgaben \rightsquigarrow 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

Fragen von letzter Woche

- Zwischenschritte beim SKI-Kalkül müssen *nicht* angegeben werden
- Nicht mehr relevante Aufgaben \rightsquigarrow 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)

$$\begin{aligned} \text{pair} &= \lambda a. \lambda b. \lambda f. f \ a \ b \\ \text{pair } c_{42} \ c_{100} &\xRightarrow{2} \lambda f. f \ c_{42} \ c_{100} \end{aligned}$$

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

$$\begin{aligned} \text{fst} &= \lambda p. p \ (\lambda a. \lambda b. a) \\ \text{snd} &= \lambda p. p \ (\lambda a. \lambda b. b) \end{aligned}$$

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

Geben Sie *next* an, sodass $\text{next}(n, m) = (m, m + 1)$.

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 $\text{next } (n, m) = (m, m + 1)$.

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

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

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

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

$$\begin{aligned} pred &= \lambda n. fst (n \ next (pair\ c_0\ c_0)) \\ sub &= \lambda m. \lambda n. n\ pred\ m \end{aligned}$$

- *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

Für Variablen t :

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

Für Aufrufe $f \ a$:

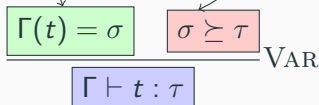
$$\frac{\Gamma \vdash f : \phi \rightarrow \alpha \quad \Gamma \vdash a : \phi}{\Gamma \vdash f \ a : \alpha} \text{APP}$$

Für Funktionsterme $\lambda p.b$:

$$\frac{\Gamma, p : \pi \vdash b : \rho}{\Gamma \vdash \lambda p.b : \pi \rightarrow \rho} \text{ABS}$$

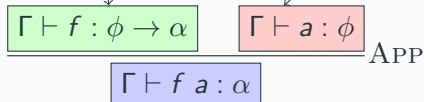
- Γ ist ein Typkontext
- D.h. *Liste* von Variable/Typ-Tupeln
- Bspw. $x : \text{char}, y : \text{int}$
- **Vorsicht:** Γ ist keine Menge, denn Γ hat eine Reihenfolge!

- „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 τ “
- $\sigma \succeq \tau \rightsquigarrow$ „ σ hat τ s Struktur und ist (mind.) allgemeiner“
 - $\text{int} \rightarrow \text{int} \succeq \text{int} \rightarrow \text{int}$
 - $\forall \alpha. \alpha \rightarrow \alpha \succeq \text{int} \rightarrow \text{int}$
 - $\alpha \rightarrow \alpha \not\succeq \text{int} \rightarrow \text{int}$
 - $\text{int} \rightarrow \text{int} \not\succeq \forall \alpha. \alpha \rightarrow \alpha$

- „ f ist im Kontext Γ eine Funktion, die ϕ s auf α s abbildet“
- „ a ist im Kontext Γ ein Term des Typs ϕ “



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

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

$$\frac{\boxed{\Gamma, p : \pi} \quad \boxed{\vdash b : \rho}}{\boxed{\Gamma \vdash \lambda p. b : \pi \rightarrow \rho}} \text{ABS}$$

- 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: $((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$

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: $((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$
- Ansatz (unten links: leerer Kontext):

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

- 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 ist eine Programmiersprache, wenn auch eine „komische“
- \rightsquigarrow 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 Übung](#) verlinkt

How did we get here?

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

How did we get here?

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

How did we get here?

- Imperative Sprachen:
 - Rein imperativ \approx Programm ist Liste von Befehlen (MIMA)
 - Strukturiert \approx Programm ist Liste von Prozeduren (C)
 - Objektorientiert \approx 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)`

How did we get here?

- Imperative Sprachen:
 - Rein imperativ \approx Programm ist Liste von Befehlen (MIMA)
 - Strukturiert \approx Programm ist Liste von Prozeduren (C)
 - Objektorientiert \approx 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

How did we get here?

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

How did we get here?

- 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 .
 - \rightsquigarrow logische Programmierung

```
% modulhandbuch.pl  
  
requires(pse, opruefung).  
requires(pse, swt1).  
  
semester(pse, 3).  
semester(swt1, 2).
```

- Legt semester/2 und requires/2 für opruefung und Abhängigkeiten an
- Schreibt reqT/2, was auch transitive Abhängigkeiten findet

Detektivrätsel

- Drei Verdächtige: Alice, Bob, Carl (leben in einer WG)
- ... leider auch die einzigen Zeugen
- Alice:
 - Bob war mit dem Opfer befreundet.
 - Carl und das Opfer waren verfeindet.
- Bob:
 - Ich war überhaupt nicht daheim!
 - Ich kenne den garnicht!
- Carl:
 - Ich bin unschuldig!
 - Wir waren zum Zeitpunkt der Tat alle in der WG.
- Überlegt, wie man dieses Problem mit Prolog darstellen und lösen könnte!