Übungsblatt 5

Alexander Mattick Kennung: qi69dube

Kapitel 1

20. April 2020

1 Überblick

- Ein programm, das ein Ergebnis liefert soll termineren. Ein programm, das kein Ergebnis liefert (e.g. OS), soll immer antwortfähig bleiben(reaktives System)
- Liefert das Programm das korrekte Ergebnis? (bzw bei reaktiven Systemen) verhält es sich richtig

$\Longrightarrow$ wir betrachten Semantik.

Planung:

- Termersetzun (technisch alle funk. Programmiersprache FOL termersetzung)
- λ-Kalkül (HOL-Termersetzung $\equiv$ Lisp, λ war lange zeit semantikfrei und Lisp ist es immernoch)
- (Ko-)Datentyp, (Ko-)Induktion
- reguläre Ausdrücke und minimierung von Automaten

Literatur:

- TES: Term Rewriting and all that (für die Leute dies genau wissen wollen), J.W. Klop term rewriting systems (kurz$\approx$ 80 seiten, kostenlos), Giesl: Termersetzungssysteme (Polynomordnung/Terminierung)
- λ-Kalkül: Lambda Calculi with Types (einfach, vom Jesus des Lambda, kostenlos), Nipkow: Lambda-Kalkül (anders als in der Vorlesung)
- (Ko-)Induktion: A Tutorial on (Co-)Algebras and (Co-)Induktion (lesbar geschrieben, rel kurz), Automat and Coinduction-An exercise in Coalgebra
- Reg. Ausdrücke/Automaten: Hopcraft/Ullman/Motwani (bibel), Pitts: Lecture notes on Regular Languages and Finite Automata (kostenlos)

2 Termersetzungssysteme

$\mathbb{N} \ni 0$, Implikation $\iff$ mit doppelstrich.

- Umformung von termen gemäß <u>gerichteter</u> Gleichungen (sukzessive, erschöpfend) Anwendungen:

- (algebraische) Spezifikation (nicht mehr 100% aktuell, andere sind beliebter)
- Programmverifikation

- automatisches Beweisen (Coq, Anabell)
- Computeralgebra (Gröbnerbasen/Buchbergeralgo[1])
- Programmierung: Turingvollständig & Grundlage der funktionalen Programmierung.

```
data Nat = Zero | Suc Nat
plus::Nat->Nat->Nat
plus Zero x = x
plus (Suc y) x = Suc (plus y x)
```

Auswertung:

$plus(Suc(SucZero))(SucZero) \rightarrow Suc(plus(SucZero)(SucZero)) \rightarrow Suc(Suc(plusZero(SucZero)) \rightarrow Suc(Suc(Suc(zero)))$

"Beweisen":

$(2+x)+y = 2+(x+y) \iff p(S(S(x)))y \rightarrow S(p(S(x)y)) \rightarrow S(S(pxy))$

Optimieren:

plus (plus x y) z = plus x (plus y z) assoziative Gesetz die Linke Seite braucht (nach obiger Suc-definition) 2x+y.

Die rechte Seite braucht x+y

(Weil nur das erste argument eine mehrfachauswertung der funktion bewirkt)

geht "plus x y = plus y x" auch?

NEIN, weil man hiermit in eine unendliche schleife geraten könnte! alternativ z.B. $y < x$ erzwingen, aber dann ist die frage: lohnt sich das? (hier nicht, weil der vergleich genau diese zeitdifferenz kostet).

Verifikation:

$p(S(SZ))x = p(SZ)(Sx)$ "Anforderung"

$p(S(SZ))x \rightarrow S(p(SZ)x) \rightarrow S(S(pZx)) \rightarrow S(Sx)$

$p(SZ)(Sx) \rightarrow S(pZ(Sx)) \rightarrow S(Sx)$

Beide werden in die gleiche Normalform reduziert (NF bezeichnet hier einen Zustand, der nicht mehr reduziert werden kann!)

Problem: Was ist wenn man nie eine NF erreicht? Wenn man ungleiche Normalformen erhält, könnte die Gleichung trotzdem gleich sein, wenn die Normalformbildung nicht deterministisch ist!

[1] https://de.wikipedia.org/wiki/Buchberger-Algorithmus