Entwicklung korrekter Programme in Coq

Thomas Strathmann
<thomas.strathmann@informatik.uni-oldenburg.de>

Überblick

- Teil 1
  - Was ist Coq?
  - Theoretische Grundlagen
  - Programmierung, Spezifikation, Beweise in Coq
- Teil 2
  - Vorstellung der Arbeit “Entwicklung eines korrekten Übersetzers für eine funktionale Programmiersprache in Coq”

Coq

- interaktiver, taktischer, typtheoretischer Theorembeweiser
- Basis: *Kalkül der induktiven Konstruktionen*
  - konstruktive Typtheorie + induktive Definitionen
  - abhängig getypter Lambda-Kalkül
  - Programmextraktion

Getypter Lambda-Kalkül

- Formalismus zur Beschreibung berechenbarer Funktionen
- Grundlage vieler funktionaler Sprachen
- Terme:
  - jeder Term hat einen Typ: x :T
  - Konstanten und Variablen
  - Funktionsanwendung: $(F\ G)$
  - Abstraktion: $\lambda x : T_1\ .\ M : T_2$

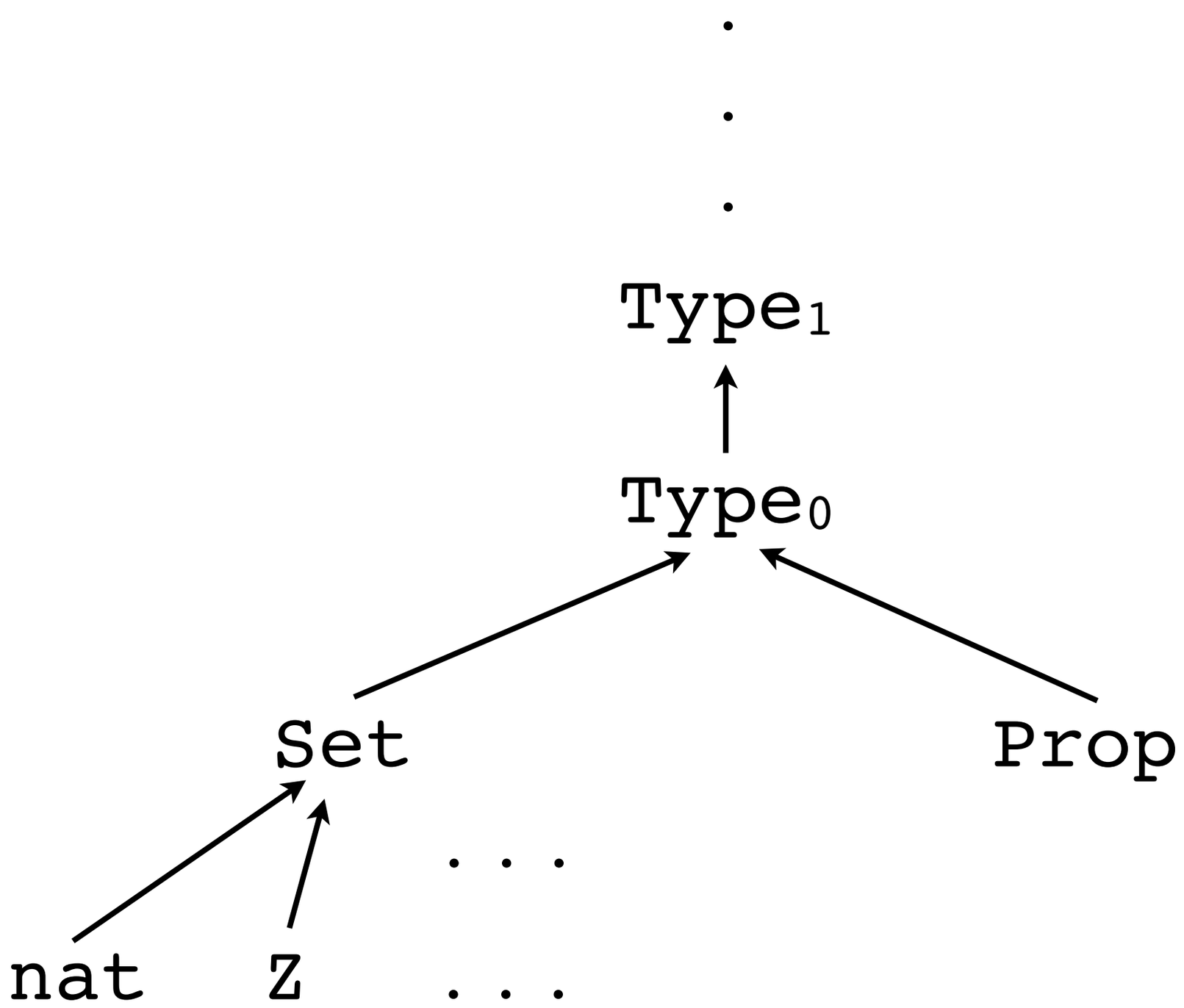
Konstruktive Typtheorie

- Konstruktive Semantik von Beweisen (Brouwer, Heyting, Kolmogorow)
  - Aussage ist eine Aufgabe; Beweis ist ein Verfahren zum Lösen der Aufgabe
- Typtheorie: Aussage ist Typ ihres Beweises
  - *P* ist wahr, gdw. man einen Term *t* : *P* konstruieren kann
    - *t* heißt Beweisterm für *P*

Curry-Howard Korrespondenz

- Beweisbarkeit der Aussage *P*
  - Ableitbarkeit von *P* in einem Kalkül
  - Existenz eines Terms vom Typ *P*
- Ableitungen in einem Kalkül natürlichen Schließens für konstruktive Logik entsprechen Termen des einfach getypten Lambda-Kalküls (Howard 1969)
  - *Beispiel*: Beweis für $P \Rightarrow Q$ ist ein Verfahren, das zu einem Beweis für P einen Beweis für Q liefert. Entspricht $(\lambda x : P \,.\, t : Q)$

Typhierarchie von Coq



Jedes Objekt in einer Typtheorie muss einen Typ haben. Deshalb müssen Typen selbst auch einen Typ besitzen. (Typen von Typen werden Sorten genannt.) Folglich gibt es in Coq eine unendliche Hierarchie von Typen.

`Set` und `Prop`

- Entsprechung zwischen Mengen (`Set`) und Propositionen (`Prop`)
  - Terme in `Set` sind Spezifikationen für Programme (d.h. als Funktion realisierbar)
    - Terme, deren Typ in `Set` ist, werden extrahiert
  - Terme in `Prop` sind logische Aussagen
    - Terme, deren Typ in `Prop` ist (= Beweise), werden bei Extraktion entfernt

Induktive Definitionen

- Induktive Definitionen modellieren (generalisierte) algebraische Datentypen und Relationen
- *Beispiel*: Natürliche Zahlen (Peano)

```
Inductive nat : Set :=
  O : nat
| succ : nat -> nat.
```

Rekursive Funktionen

- keine partiellen Funktionen
  - vollständiges Pattern Matching auf Konstruktoren induktiver Datentypen
- nur terminierende Funktionen (syntaktische Terminierungsprüfung)
  - Strukturelle Rekursion: ein Funktionsargument muss bei jedem rekursiven Aufruf strukturell kleiner werden; Aufruf nur mit Teilterm erlaubt

Strukturelle Induktion ist ein Spezialfall der fundierten Induktion (auch Noethersche Induktion genannt). Coq unterstützt auch fundierte Induktion/Rekursion für Fälle, in denen sich eine Funktion nicht strukturell rekursive ausdrücken kann. Die wesentliche Idee hierbei ist, Terminierung dadurch zu zeigen, dass ein Maß (im mathematischen Sinne eine Norm) der Argumente bei jedem rekursiven Aufruf streng monoton fallend ist, also eine absteigende Kette bildet.

Funktion: Beispiel

```
Fixpoint add (n m:nat) : nat :=
  match n with
    | O => m
    | succ n' => add n' (succ m)
  end.
```

Terminierung:

Rekursiver Aufruf mit strukturell kleinerem Argument `n'`

Beweise: Grundlagen

- Beweisvorgang eingeleitet durch Angabe eines *Beweisziels* (Goal)
  - *Beispiel*: `Goal forall n m:nat, n+m = m+n.`
- *Kontext* mit *Hypothesen* $H_1, \ldots, H_n$, aus denen das Ziel G durch Anwendung von *Taktiken* hergeleitet werden soll
  - $H_1 : P_1 \quad H_2 : P_2 \ \ldots \ H_n : P_n \vdash G \ : \ P$

Die interaktiven Beweise sind insofern imperativ als sie einen Zustand, den Kontext, verändern.

Beweise: Taktiken

- führen Beweisschritte aus (imperativ)
- Programmiersprache $\mathcal{L}_{tac}$
  - Kontrollfluss-Operationen: `try`, `repeat`, `orelse`, `fail`, `idtac` ...
    - Pattern Matching auf (Teiltermen von) Ziel und Hypothesen
- Verbesserte Wartung durch Taktiken, die robust gegenüber Änderungen der Formalisierung sind

Beweise: Beispiel

```
Lemma add_unit_right:
    ∀ n:nat, add n O = n.
Proof.
induction n as [|n'].
  Case "n = O". reflexivity.
  Case "n = succ n'".
    simpl.
    rewrite IHn.
    reflexivity.
Qed.
```

Relationen

- Relation $\rho \subseteq A \times B$ als Aussage auffassen, die wahr ist gdw. $A \ \rho \ B$
- *Beispiel*: Relation < auf natürlichen Zahlen

```
Inductive less_than : nat → nat → Prop :=
  lt_base : less_than O (succ O)
| lt_succ : ∀ n m : nat, less_than n m →
                    less_than (succ n)(succ m)
| lt_pred : ∀ n m : nat,
      less_than (succ n) m -> less_than n m.
```

Relationen (forts.)

```
Example lt1_4: less_than (succ O)
  (succ (succ (succ (succ O)))).
apply lt_pred.
apply lt_pred.
repeat apply lt_succ.
apply lt_O.
Qed.
```

Abhängige Typen

- Mit Termen indizierte Typfamilien
  - statische Typüberprüfung statt dynamischer Prüfung und Laufzeitfehlern
  - Funktionstypen mit Vor- und Nachbedingungen
  - Datentypen, deren Instanzen per constructionem Invariaten erfüllen
  - *Beispiel*: längenindizierte Listen

```
Inductive vector {A:Set} : nat → Set :=
  Vnil : vector O
| Vcons : A → ∀ n : nat, vector n → vector (succ n).
```


Abhängige Typen (dependent types) stammen aus der Typtheorie von Martin-Löf, die als Basis für die Formalisierung der gesamten konstruktiven Mathematik entwickelt wurde. Allerdings gab es die Idee der abhängigen Typen schon in der Untersuchung der Kombinatorlogik durch Curry.

Für die Haskellianer: Das Paper “Faking It” von Conor McBride beschreibt eine Codierung abhängiger Typen in Haskell.

Beispiel: Spezifikation

- Spezifkation der Funktion head:

```
forall (A:Type) (l:list A) :
  length l > 0 ->
    { l' : list A & { x : A | l = x :: l' } }.
```

Vorbedingung

Nachbedingung

- Eingabe: Liste `l` und Beweis für `length l>0`
- Ausgabe: `l'` und `x` sowie Beweis der Nachbedingung

Es handelt sich um einen abhängigen Funktionstyp, da er von einem Beweis der Vorbedingung abhängt. Dieser Beweis wird (weil sein Typ in Prop ist) bei der Extraktion entfernt.
Für die praktische Verwendung der Funktion schreibt man dann einen Wrapper, der nur das Head-Element x zurückliefert, nicht aber die Liste l'.

Beispiel: Beweis

```
Definition head (A:Type) (l:list A) :
  length l > 0 ->
    { l' : list A & { x : A | l = x :: l' } }.
intros.
induction l.
contradict H.
compute.
intros.
inversion H.
exists l.
exists a.
reflexivity.
Qed.
```

Ende Teil I

Fragen soweit?

Implementierung eines korrekten Übersetzers

- Formale Methoden wertlos, wenn Übersetzer fehlerhaft
- Compilerverifikation noch sehr aufwändig
  - *Beispiel*: CompCert: 42.000 Zeilen Coq (14% Compiler, 10% Semantik, 76% Beweis)
- rein-funktionale Sprachen haben einfacherere Semantik => besser geeignet für eine Diplomarbeit

Entscheidungen

- Quellsprache: Syntax und Semantik
  - Welche Sprache?
  - Termrepräsentation in Coq?
  - Welche Art Semantik? Wie formalisieren?
- Zielarchitektur
  - Maschinenmodell
  - Übersetzung

Vorgehen

- Formalisieren der Quellsprache
- Implementierung der Zielarchitektur
- Implementierung der Übersetzungsschritte
- Korrektheitsbeweis: Übersetzung erhält Semantik
  - interessante Punkte:
    - Automatisierung der Beweise
    - Auswirkung von Semantik und Termrepräsentation

Vielen Dank für die Aufmerksamkeit!

Fragen/Anregungen?