Grundlagen von Coq

Oktober 2021

Inhaltsverzeichnis

Vorwort

Im Laufe der letzten Jahrhunderte scheint sich in unserer Welt mit der Vertiefung der Erkenntnis und dem neu hinzugekommenen Wissen immer mehr Komplexität aufgetan zu haben. Insbesondere kommen in der Mathematik immer wieder verwickelte Sätze und Theorien hinzu, bei deren Beweisen ein unaufhörliches Wachstum der Länge zu beobachten ist. Eine ähnliche Schwierigkeit keimt in der Informatik auf, wo mit der wachsenden Vielfalt an Anforderungen die Länge von Programmen fortlaufend zunimmt.

Aus diesem Grund kommen wir letztendlich irgendwann in den Zugzwang, dieser Komplexität auf irgendeine Art Einhalt gebieten zu müssen.

Der folgende Text erkläutert, wie sich unter Zuhilfenahme von Computern mathematische Beweise verfizieren lassen. Grundlegend dafür ist der Curry-Howard-Isomorphismus, ein fundementaler Zusammenhang zwischen mathematischen Aussagen und Typsignaturen. Hierauf baut der Konstruktionenkalkül auf, mit dessen Erweiterung, dem Kalkül der induktiven Konstruktionen, man auf systematische Art Beweise in Programmterme umwandeln kann.

Wie läuft das ab? Zu jeder mathematischen Aussage gehört eine äquivalente Typsignatur. Findet man nun einen Programmterm, welcher einen Wert des Typs berechnet, so ist der Typ offenbar nichtleer. Diese Feststellung bedeutet aber nichts anderes, als dass der Beweis der ursprünglichen Aussage erbracht ist.

Damit diese Argumentation stichhaltig bleibt, darf der Term ausschließlich *reine* Funktionen enthalten. Das sind solche Funktionen, bei denen der Rückgabewert ausschließlich von den Argumenten abhängig ist. Die Berechnung läuft hierbei strikt funktional ab, – damit ist gemeint dass alle Daten unveränderlich sind und die Berechnung somit frei von Seiteneffekten ist. Außerdem muss jede Berechnung stets terminieren, darf sich also niemals in einer Endlosschleife verfangen. Mathematisch gesprochen bedeutet dies, dass jede auftretende Funktion *total* ist, dass also keine partiellen Funktionen vorkommen. Beide Forderungen gemeinsam, Reinheit und Totalität, bedeuten, dass wir es mit Funktionen im mathematischen Sinn zu tun haben.

Schließlich muss das Typsystem reichhaltig genug sein, um alle erdenklichen Aussagen kodieren zu können. Hierzu bedarf es der *parametrischen Polymorphie*, d. h. der Parametrisierung von Typen über Typparameter, die bei vielen modernen Programmiersprachen Grundlage für die generische Programmierung bildet. Darüber hinaus sind *abhängige Typen* notwendig, die die Parametrisierung so erweitern, dass als Typparameter nicht nur Typen, sondern auch Werte erlaubt sind.

Der vertrauenswürdige Programmkern, die *trusted computing base*, ist klein. Es ist dieser Flaschenhals, der uns ruhiger Schlafen lässt.

Ich hoffe der Text ist auch horizonterweiternd, wenn man nun nicht jeden Beweis mit diesem System verfizieren möchte. So handelt es sich hier um eine Berührregion zwischen Mathematik und Informatik.

1 Gallinas Typsystem

1.1 Programmieren in Gallina

2 Logik

2.1 Modus ponens

Wir wollen den Modus ponens

$$A \wedge (A \to B) \to B \tag{2.1}$$

beweisen. Der Typ zu dieser Aussage ist

$$A \times (A \to B) \to B. \tag{2.2}$$

Die Aussage ist wahr, wenn dieser Typ mindestens einen Wert enhält. Gesucht ist also eine Funktion, die einen Wert vom Typ A und eine Funktion vom Typ $A \to B$ nimmt und einen Wert vom Typ B liefert. Die kann man direkt angeben:

$$(a, f) \mapsto f(a). \tag{2.3}$$

Die freien Variablen müssen wir noch allquantifizieren, damit später im Programm keine ungebundenen Variablen mehr auftauchen. Das macht

$$\forall_{A,B}(A \times (A \to B) \to B). \tag{2.4}$$

Ein Wert dieses Typs ist entsprechend

$$(A, B) \mapsto (a, f) \colon A \times (A \to B) \mapsto f(a). \tag{2.5}$$

In Coq geschrieben lautet der Term:

```
Definition modus_ponens:
forall A B: Type, A*(A -> B) -> B :=
  fun (A B: Type) =>
    fun (t: A*(A -> B)) =>
      match t with (a, f) => f a end.
```

Allgemein gilt die äquivalente Umformung

$$\begin{aligned} A \wedge B \to C &\iff \overline{A \wedge B} \vee C \iff \overline{A} \vee \overline{B} \vee C \\ &\iff A \to \overline{B} \vee C \iff A \to B \to C. \end{aligned} \tag{2.6}$$

Dies entspricht dem Schönfinkeln der Funktion. Demnach kodiert der Typ

$$A \to (A \to B) \to B. \tag{2.7}$$

ebenfalls den Modus ponens. In Coq sind geschönfinkelte Formulierungen natürlicher. Das sieht man hier am Entfallen des match-Operators:

```
Definition modus_ponens:
forall A B: Type, A -> (A -> B) -> B :=
  fun (A B: Type) =>
    fun (a: A) =>
      fun (f: A -> B) => f a.
```

Nun sind in Coq Kurzschreibweisen erlaubt, die das Schönfinkeln syntaktisch rückgängig machen:

```
Definition modus_ponens:
forall A B: Type, A -> (A -> B) -> B :=
  fun (A B: Type) (a: A) (f: A -> B) => f a.
```

2.2 Formeln der Aussagenlogik

Betrachten wir $A \wedge B \to A$, der Typ dazu ist

$$A \times B \to B.$$

Ein Wert dieses Typs ist ja einfach die Projektion auf das linke Element, also

$$(a, b) \mapsto a.$$

Das Programm:

```
Definition left: forall A B: Type, A*B -> B :=
  fun (A B: Type) (t: A*B) =>
    match t with (a, b) => a end.
```

Zum Kommutativgesetz $A \wedge B \to B \wedge A$ gehört der Typ

$$A \times B \to B \times A. \tag{2.8}$$

Ein Wert dieses Typs ist die Funktion, welches linkes und rechtes Element vertauscht, das ist

$$(a, b) \mapsto (b, a).$$

Das Programm ist entsprechend:

```
Definition conjunction_commutativity:
forall A B: Type, A*B -> B*A :=
  fun (A B: Type) (t: A*B) =>
    match t with (a, b) => (b, a) end.
```

Zum Kommutativgesetz $A \vee B \to B \vee A$ gehört der Typ

$$A + B \to B + A. \tag{2.9}$$

Eine Funktion dieses Typs ergibt sich durch Fallunterscheidung. Wir finden

$$\begin{cases} (\text{left}, a) \mapsto (\text{right}, a), \\ (\text{right}, b) \mapsto (\text{left}, b). \end{cases} \tag{2.10}$$

Das Programm:

```
Definition disjunction_commutativity:
forall A B: Type, A + B -> B + A :=
  fun (A B: Type) (s: A + B) =>
    match s with
      | inl a => inr a
      | inr b => inl b
    end.
```

Betrachten wir noch $A \to A \vee B$ bzw. $A \to A + B$. Eine Funktion dieses Typs ist offenbar die Injektion mit Bild im linken Summand, das ist

$$a \mapsto (\text{left}, a). \tag{2.11}$$

Das Programm:

```
Definition injection_left:
forall A B: Type, A -> A + B :=
  fun (A B: Type) (a: A) => inl a.
```

Zeigen wir nun noch das Assoziativgesetz

$$A \wedge (B \wedge C) \leftrightarrow (A \wedge B) \wedge C. \tag{2.12}$$

Wir betrachten lediglich

$$A \times (B \times C) \to (A \times B) \times C, \tag{2.13}$$

denn die Gegenimplikation geht analog. Eine Funktion von diesem Typ ist

$$(a, (b, c)) \mapsto ((a, b), c). \tag{2.14}$$

Das Programm:

```
Definition conjunction_assoc1:
forall A B C: Type, A*(B*C) -> (A*B)*C :=
  fun (A B C: Type) (t: A*(B*C)) =>
    match t with (a, (b, c)) => ((a, b), c) end.
```

2.3 Umstieg auf Prop

Nun ersetzen wir `Type` durch `Prop`. Hierfür sind einige kleine Änderungen notwendig.

Betrachten wir nochmals $A \wedge B \to P$. Wir müssen $A \times B$ durch $A \wedge B$ ersetzen. Die Tupelkonstruktion (a, b) wird ersetzt durch conj $a\, b$. Insgesamt bekommt das Programm damit die Form:

```
Definition left: forall A B: Prop, A /\ B -> A :=
  fun (A B: Prop) (t: A /\ B) =>
    match t with conj a b => a end.
```

Und nun betrachten wir nochmals $A \vee B \to B \vee A$. Hier ersetzen wir $A + B$ durch $A \vee B$. Der Konstruktor `inl` wird ersetzt durch `or_introl` und `inr` entsprechend durch `or_intror`. Das Programm ist nun von der Gestalt:

```
Definition disjunction_commutativity:
forall A B: Prop, A \/ B -> B \/ A :=
  fun (A B: Prop) (s: A \/ B) =>
    match s with
      | or_introl a => or_intror a
      | or_intror b => or_introl b
    end.
```

Literatur

[1] Benjamin C. Pierce et al.: »*Software Foundations. Volume 1: Logical Foundations*«.