Agda

Programmieren und Beweisen

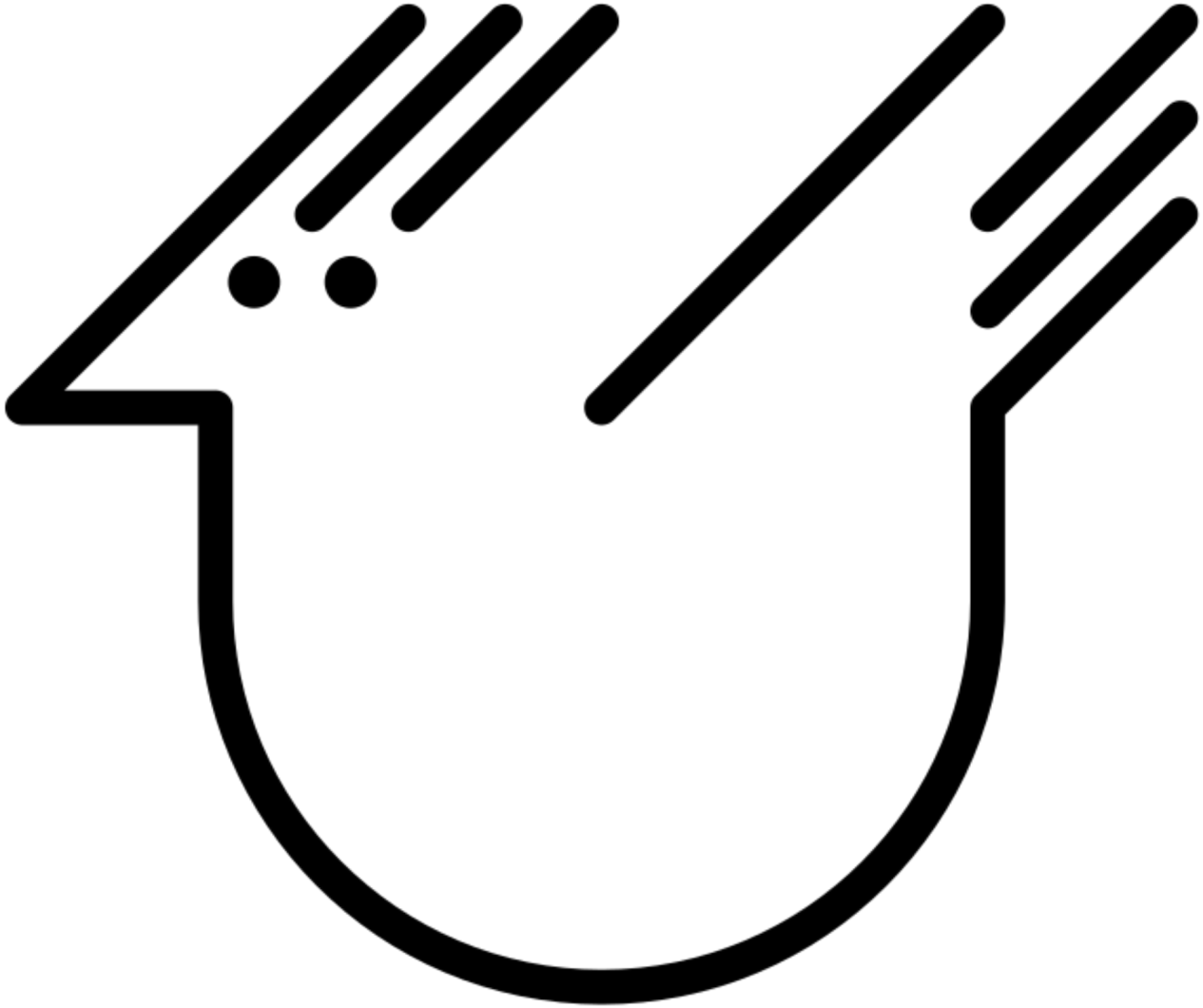
Nick Wagner

February 8, 2022

Übersicht

Einleitung

Hönan Agda



Allgemeines zu Agda

Was ist Agda?

- Funktionale Programmiersprache
- Syntax ähnlich zu Haskell
  (Verwendet den Glaskow Haskell Compiler)
- Stark typisiert
- Verwendet Dependent Types
- Implementation des Curry Howard Isomorphismus
- Ideal für das maschinengestützte Beweisen und Verifikation

Geschichte von Agda

- "Agda 1", 1999 Göteborg, Catarina Coquand
- Ehefrau von Thierry Coquand (Entwickler von Coq)
- "Agda 2", Ulf Norell, für PhD Thesis komplett neu geschrieben

Catarina Coquand

Ulf Norell

Curry Howard Isomorphismus

Die Auffassung von Typen als Logische Aussagen.

- Logische Aussagen können als Typen dargestellt werden.
- Programme können als Typen dargestellt werden.
- Beweise können als ausführbare Ausdrücke geschrieben werden.
- Beweissysteme und Berechnungsmodelle sind gleichbedeutend.

$$\begin{aligned} \text{Beweis} &\Leftrightarrow \text{Programm} \\ \text{zu beweisende Formel} &\Leftrightarrow \text{Typ des Programms} \end{aligned}$$

Totale Programmierung wird notwendig. Programme müssen...:

- ...korrekt sein.
- ...vollständig sein $\Leftrightarrow$ alle möglichen Fälle implementieren.
- ...terminieren $\Leftrightarrow$ Aussage entscheidbar.

Typ Checker

Type Checker:

- Löst Typen auf (bidirectional type inference) und vergleicht diese miteinander.

Termination Checker:

- Überprüft, ob das Programm terminiert.
- (bei Rekursion z b. durch Analyse der Parameter).
- IO-Funktionen und Endlosschleifen sind nicht nativ in Agda Realisierbar.
- Die Klasse der Agda-Programme ist nicht Turing-Mächtig.

Coverage Checker:

- Programme müssen vollständig sein.
  Ausnahmen sind: Löcher `"?"` und Absurd Patterns `"()"`
- Beim Patternmatching muss jeder mögliche Fall angegeben werden.

Ansonsten: Fehlermeldung zur Kompilierzeit!

Zusammenfassung

- Nicht alle Programme sind mit Agda realisierbar ☹
- Programmieren mit Agda ist kompliziert ☹
- Agda eignet sich zum Beweisen
- Alle Programme sind Fehlerfrei ☺(correct-by-construction)

Typen

Typisierung

- Agda ist stark typisiert.
- Der Typ muss immer angegeben werden (in Haskell nicht immer).
- Typen sind strikt positiv definit.
  - $(z_1 : C_1) \to ... \to (z_m : C_m) \to D$
  - D darf nicht in C_i vorkommen. Die Typreduktion muss terminieren!
- Man unterscheidet Funktionstypen, kleine und Große Datentypen.
  - kleine Datentypen: Bauen auf Datentyp `Set` auf
  - große Datentypen: Bauen auf kleinen und großen Datentypen auf

Typ-Definition

Datentyp:

- Typ-Spezifizierung mit Schlüsselwort **`data`** und **`where`**
- Block mit Konstruktoren (Einrücken Wichtig!)

Beispiel: Bool

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Funktionstyp:

- Typ-Spezifizierung
- Implementierung

Beispiel: Not

```
not : Bool → Bool
not true  = false
not false = true
```

Abhängige Typen

Abhängiger Datentyp

$$(x : A) \to B$$

Ein Typ, der von Elementen anderer Typen abhängt.

- x ist ein Wert vom Typ A
- Der Rückgabetyp B ist vom Wert x abhängig

Beispiel: Skalarprodukt zweier Ganzzahl-Vektoren

```
inner : {n : ℕ} → Vec ℕ n → Vec ℕ n → ℕ
inner [] [] = zero
inner (a :: as) (b :: bs) = a * b + inner as bs
```

Die Vektoren müssen die gleiche Länge `n` haben. Die Vektoren sind voneinander Abhängig.

Induktive Typen

- Nullstelliger Konstruktor = Basisfall
- Einstelliger Konstruktor = Induktionsschritt

Beispiel: Natürliche Zahlen

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Die Natürlichen Zahlen sind induktiv definiert. **`zero`** ist ein Element der Menge und **`suc`** eine Funktion, die für jedes Element dessen Nachfolger zur Menge hinzufügt.

Ziffern müssen noch definiert werden z.b. **`2 = suc (suc zero)`**
(Modul Data.Nat)

Induktive Familien

Induktive Familien sind induktive Typen mit zusätzlichem Index-Parameter. Für jeden Index enthält die Familie einen eigenen induktiven Typ.

Beispiel: Vektoren

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

`Vec A n` ist:

- mit `A` parametrisiert
- mit `n` indiziert

(Parameter werden vor dem Doppelpunkt geschrieben, wenn sie nicht in der Typspezifizierung auftauchen)

Agda Features

Modulsystem

Agda hat ein Modul System:

- Modulname = Dateiname
- Strukturiert Namespaces hierarchisch
- Aufruf über Pfad `MODULNAME.UNTERMODULNAME...`
- **`import`** importiert ein Modul
- **`open`** macht den Inhalt zugänglich
- **`using`**, **`as`** ,**`hide`** für mehr kontrolle

Beispiel: Modul-Import

```
module Data.Vec.Base where

open import Data.Bool.Base
open import Data.Nat.Base as ℕ using (zero)
```

weitere Schlüsselwörter: **`using`**, **`hiding`**, **`renaming`**, **`private`**

Pattern Matching

- Patternmatching ist sehr mächtig
- Lässt sich auf induktiven Typen verwenden
- Muss alle Fälle (Konstruktoren) abdecken
- Wildcards sind möglich
- "..." kann Parameter ersetzen, wenn diese immer gleich sind

Beispiel von vorhin: Skalarprodukt

```
inner : {n : ℕ} → Vec ℕ n → Vec ℕ n → ℕ
inner [] [] = zero
inner (a :: as) (b :: bs) = a * b + inner as bs
```

Pattern Matching

Mit Schlüsselwort **`with`** kann ein Ausdruck angegeben werden, dessen Ergebnis als zusätzliches Argument mittels | angehängt wird.

Beispiel: Liste und Filterfunktion

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A

filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
... | true  = x :: (filter p xs)
... | false =      (filter p xs)
```

Weitere Schlüsselworte: **`case`** (ähnlich zu Haskell), **`rewrite`**

Mixfix Operatoren

- Erlaubt Unterstriche im Funktionsnamen (ohne Leerzeichen), die mit Parametern aufgefüllt werden
- Beim Aufrufen werden Parameter mit Leerzeichen getrennt
- erlaubt intuitive Schreibweise

Beispiel: Addition

```
infixl 5 _+_

_+_ : ℕ → ℕ → ℕ
zero    + n = n
(suc m) + n = suc (m + n)
```

Jetzt kann `m + n` aufgerufen werden

Assoziativität: **`infix`**, **`infixl`**, **`infixr`**
Vorrang: Standard 20, höhere werte werden früher ausgewertet

Implizite Argumente

- Nicht alle abhängigen Parameter müssen bei Funktionsaufrufen angegeben werden
- Vorraussetzung: Typchecker muss diese ableiten können
- Werden in geschweiften Klammern geschrieben

Beispiel: ID Funktion implizit

```
id : {T : Set} → T → T
id x = x
```

Vergleich ID Funktion explizit

```
id : (T : Set) → T → T
id T x = x
```

Haskell Funktionen

- Agda Benutzt den Glasgow Haskell Compiler (GHC)
- Haskell Funktionen können mittels Pragmas importiert werden
- Keine Typ-Reduktionsregeln auf Haskell Funktionen

Beispiel: "Hello World"

```
open import Agda.Builtin.IO
open import Agda.Builtin.Unit
open import Agda.Builtin.String

postulate putStrLn : String → IO ⊤
{-# FOREIGN GHC import qualified Data.Text as T #-}
{-# COMPILE GHC putStrLn = putStrLn . T.unpack #-}

main : IO ⊤

main = putStrLn "Hello world"
```

Absurd Patterns

- Um keine Implementierung angeben zu müssen
- Ausnahme für Coverage Checker
- Nur erlaubt, wenn keine mögliche Belegung mit Parametern existiert.

Beispiel: Absurd Pattern

```
data ⊥ : Set where

data Even : ℕ → Set where
  isEven0 : Even 0
  isEven+2 : {n : ℕ} → Even n → Even (2 + n)

one-not-even : Even 1 → ⊥
one-not-even ()
```

Es gibt keine Ausgabe der Funktion, denn der Rückgabetyp ist eine leere Menge.

Praxis

Agda Mode

- Emac und Atom bieten ein Agda Interface.
- Agdapad ist ein online Editor (keine Installation notwendig)
- Unterstützung von Unicode Zeichen
- Incrementelles Programmieren
  (Programm wird typischerweise nach wenigen Eingaben neu kompiliert)
- Informationen zum aktuellen Stand des Programms werden bereitgestellt

Agdapad

Agdapad ist ein online Editor und Interpreter für Agda.

File Edit Options Buffers Tools Agda Help

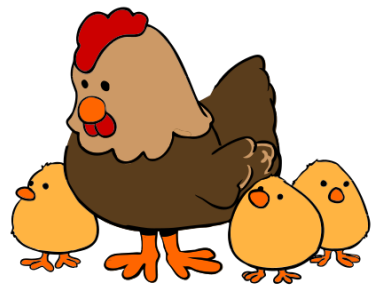
Save Undo

```
open import Data.Nat

data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)

inner : {n : ℕ} → Vec ℕ n → Vec ℕ n → ℕ
inner a b = { }0
```

U:**- hello.agda Top L10 <E> (Agda)

?0 : ℕ

U:%*- *All Goals* All L1 <M> (AgdaInfo)



https://agdapad.quasicoherent.io/

Agda Mode Funktionen

Tastenkombination	**Erklärung**
M+.	Zur Definition springen
M+,	Zurück springen
strg+c strg+c	(case) Case split
strg+c strg+l	(load) Agda laden
strg+c strg+d	(deduce) Typ des Ausdrucks herleiten
strg+c strg+a	(auto) Typ automatisch konstruieren
strg+c strg+n	(normalize) Ausdruck normalisieren
strg+c strg+,	Zeigt den erwarteten Typ eines Lochs
strg+c strg+SPC	Loch durch Ausdruck ersetzen
strg+c strg+r	(refine) mit rekursiven Ausdruck ersetzen
strg+c strg+x Strg+c	Kompilieren

Agda Mode Unicode-Zeichen

Zeichen	Agda Mode Notation
$\rightarrow$	\to
$\forall$	\forall
$\equiv$	\==
$\mathbb{N}$	\bN
$\langle$	\langle
$\rangle$	\rangle
$\blacksquare$	\qed
$\top$	\top
$\bot$	\bot

Beweise

Beweise

Ein Beweis zu einem Ausdruck ist ein ausführbares Programm bzw. eine Funktion. Wenn diese Funktion erfolgreich kompiliert, ist der Beweis:

- entscheidbar (Termination Checker)
- vollständig (Coverage Checker)
- korrekt (Type Checker)

Beweis als Agda Funktion:

- Variablen des Ausdrucks werden als Argumente übergeben.
- Der Rückgabetyp ist der Ausdruck selbst.

refl und cong

Um mathematische Ausdrücke zu beweisen, sind unter anderem folgende Eigenschaften relevant:

Äquivalenz

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

Kongruenz

```
cong : ∀ {a b} {A : Set a} {B : Set b}
       (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

$$x \equiv y \Rightarrow f(x) \equiv f(y)$$

(Aus Modul Relation.Binary.PropositionalEquality)

Direkter Beweis

Zu zeigen:

Ausdruck

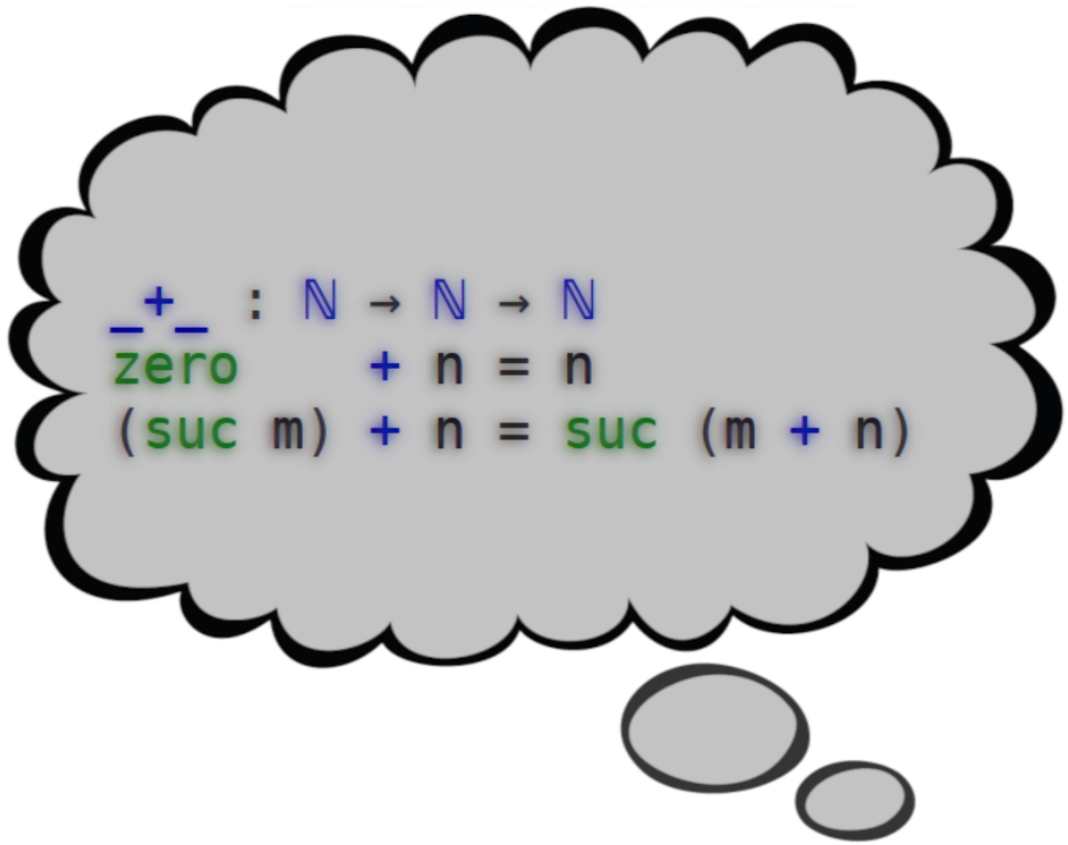
$$2 + 1 \equiv 3$$

Direkter Beweis:

- Über schrittweise Herleitung
- Keine Variablen $\Rightarrow$ keine Argumente
- Nur bekannte Umformungen dürfen verwendet werden
- Hier: $\mathbb{N}$, $\_+\_$, $\_\equiv\_$

Direkter Beweis

```
open import Relation.Binary.PropositionalEquality
open Relation.Binary.PropositionalEquality.≡-Reasoning
open import Data.Nat

proof : 2 + 1 ≡ 3
proof =
  begin
    2 + 1
  ≡⟨⟩
    suc (1 + 1)
  ≡⟨⟩
    suc (suc (0 + 1))
  ≡⟨⟩
    suc (suc 1)
  ≡⟨⟩
    3
  ∎
```



Direkter Beweis

Typchecker wertet den Typen intern soweit wie möglich aus:

$$
\begin{array}{rcl}
\texttt{2 + 1} & \equiv & \texttt{3} \\
 & \dots & \\
\texttt{suc (suc zero) + suc zero} & \equiv & \texttt{suc (suc (suc zero))} \\
 & \dots & \\
\texttt{suc (suc (suc zero))} & \equiv & \texttt{suc (suc (suc zero))} \\
 & \dots & \\
\texttt{3} & \equiv & \texttt{3}
\end{array}
$$

- Typ kann im Infofenster nachgeschaut werden
- Entspricht `refl`

Beweis kurz

```
proof : 2 + 1 ≡ 3
proof = refl
```

□

Beweis über Induktion

Zu zeigen:

Ausdruck

$$x + (y + z) \equiv (x + y) + z$$
$$x, y, z \in \mathbb{N}$$

Beweis über Induktion:

- Die natürlichen Zahlen sind induktiv definiert
  $\Rightarrow$ Beweis ebenfalls induktiv
- 3 Variablen $\Rightarrow$ Beweis hat 3 Argumente

Beweis über Induktion

```
proof : ∀ (x y z : ℕ) → (x + y) + z ≡ x + (y + z)
proof x y z = ?
```

- Programm soll schrittweise erstellt werden
  $\Rightarrow$ implementation durch ? ersetzt
- Programm laden:
  (strg+c strg+l)

```
proof : ∀ (x y z : ℕ) → (x + y) + z ≡ x + (y + z)
proof x y z = { }0
```

- Automatisches herleiten von `{ }0` schlägt fehl
  (strg+c strg+a)
- Es genügt z.z., dass der Ausdruck für alle `x` gilt

Beweis über Induktion

- Case Split auf **x**
- Automatisches herleiten von **{ }0** schlägt fehl (strg+c strg+a)
- Es genügt z.z., dass der Ausdruck für alle **x** gilt
- Case Split auf **x** (strg+c strg+c, x, enter)
- Erzeugt Patternmatching mit **zero** und **(suc x)**

```
proof : ∀ (x y z : ℕ) → (x + y) + z ≡ x + (y + z)
proof zero y z = { }0
proof (suc x) y z = { }1
```

Beweis über Induktion

- Automatisches Herleiten von `{ }0` ergibt `refl`
- Automatisches Herleiten von `{ }1` schlägt fehl.
  Das Info-Fenster zeigt den Typ:
  `suc x + y + z ≡ suc x + (y + z)`
- `cong` lässt sich anwenden
- Der Resultierende Ausdruck hat dieselbe Form wie ursprünglicher Ausdruck
- rekursiver Aufruf

```
proof : ∀ (x y z : ℕ) → (x + y) + z ≡ x + (y + z)
proof zero y z = refl
proof (suc x) y z = cong suc (proof x y z)
```

□

Beweis über Reflektion

Zu zeigen:

Ausdruck

6 ist gerade

Beweis über Reflektion:

- Definiere bfseries`Even` als Menge aller geraden Zahlen
- Wenn Zahl gerade, dann gibt es Herleitung aus Konstruktoren
- Agda kann einfache Herleitungen automatisch berechnen

Beweis über Reflektion

```
data Even : ℕ → Set where
  isEven0 : Even 0
  isEven+2 : {n : ℕ} → Even n → Even (2 + n)

proof : Even 6
proof = { }0
```

```
isEven+2 (isEven+2 (isEven+2 isEven0))
```

- Cursor im Loch platzieren und (strg+c strg+a)
- Loch wird durch Herleitungs-Kette ersetzt

Beweis geglückt!

Wenn Zahl sehr groß ⇒ Herleitung nicht möglich!

Beweis über Reflektion

Beweis für beliebige große Zahlen

```
data Wrong : Set where
data Right : Set where
  r : Right

even? : ℕ → Set
even? 0 = Right
even? 1 = Wrong
even? (suc (suc n)) = even? n

soundEven : {n : ℕ} → even? n → Even n
soundEven {0} r  = isEven0
soundEven {1} ()
soundEven {suc (suc n)} s = isEven+2 (soundEven s)

isEven8772 : Even 8772
isEven8772 = soundEven r
```

- **SoundEven** erzeugt automatisch einen Herleitungsbaum

Beweissysteme

Beweissystem

$$F : B \to A$$
$$A \subseteq \Gamma^*$$
$$B \subseteq \Sigma^*$$

- B ist eine Menge von Beweisen
- A ist eine Menge von Aussagen auf einer "Sprache" Γ^*
- Funktion F gibt automatisch ein $a \in A$ für ein $b \in B$ zurück.
- A ist durch die Domain gegeben

Wie Funktioniert ein Beweissystem?

- Die Grammatik und Wörter der Sprache Γ^* müssen spezifiziert werden.
- Gleichungen werden in ihre grammatikalischen Bestandteile zerlegt
- Ausdrücke werden in Normalform gebracht (Vektor-/Matrixform)
- Normalformen werden verglichen (refl, cong)

Beweissysteme für Monoide

Konstruktion eines Beweissystems für Monoide

Monoid

Ein Tripel $(M, \bigoplus, e)$

- Menge M
- innere zweistellige Verknüpfung $\bigoplus : M \times M \to M$
- $\bigoplus$ ist Assoziativ
- Ein Neutrales Element $e \in M$

(Eine Gruppe ohne inverses Element)

Beweissysteme für Monoide Gleichungen

Grammatik der Sprache Γ^* spezifizieren:

Ausdrücke

```
data Expr n : Set where
  _⊕_  : Expr n → Expr n → Expr n
  var  : Fin n → Expr n
  zero : Expr n
```

- Ausdrücke bestehen aus Variablen einer endlichen Menge ink. zero, oder aus zwei durch einen Operator verknüpfte Ausdrücke.
- `n` ist die Anzahl enthaltener Variablen.

Gleichungen

```
data Eqn n : Set where
  _==_ : Expr n → Expr n → Eqn n
```

- Gleichungen bestehen aus zwei durch == getrennte Ausdrücke.

Beweissysteme für Monoide Gleichungen

Normierung

```
norm : ∀ {n} → Expr n → Expr n
norm = reify ∘ eval
```

- **norm** wendet erst **reify** und dann **eval** auf dem Ausdruck an.
- **reify** verwandelt den Ausdrück in eine Darstellung, in der jedes Vorkommen einer Variablen in einem Vektor gespeichert wird. **eval** Verwandelt den Ausdruck zurück in die Form der Sprache Γ^*.

Normierung

```
NF : ℕ → Set
NF n = Vec ℕ n

eval  : ∀ {n} → Expr n → NF n
reify : ∀ {n} → NF n → Expr n
```

Beweissysteme für Monoide Gleichungen

Durch das normalisieren kann der Ausdruck bewiesen werden.
(analog zu dem direkten Beweis mittels schrittweisen Umformungen)

Für den Beweis fehlen noch viele weitere (Hilfs)-Funktionen.
Letztlich ist das Beweissystem eine Funktion, die:

- Den Ausdruck verlangt
- Eine Liste der Variablen verlangt
  (Für die Konstruktion des Vektors)

... Und daraus automatisch einen Beweis erstellt
Mit dem Beweissystem können alle monoiden Ausdrücke automatisch bewiesen werden.

Agda kann Aussage genau dann automatisch beweisen, wenn ein Beweissystem für deren Sprache existiert.

Leider gibt es noch nicht viele Module mit Beweissystemen.

Fazit

Fazit

- Abhängige Typen sind mächtiges Werkzeug
- Fehler können vermieden werden
- Intuitive Syntax dank Unicode
- Beweise können manuel und automatisch durchgeführt werden
- Einige praktische Funktionen fehlen
- Programmierung etwas anspruchsvoller
- Nutzergemeinschaft bislang im akademischen Sektor
- Wenigie Module

Literatur

[1] Abel, A. An introduction to dependent types and agda. *URl: http://www2. tcs. ifi. lmu. de/˜ abel/DepTypes. pdf (visited on 29/07/2014)(cit. on p. 11)* (2009).

[2] Blechschmidt, I. Agdapad.

[3] Bove, A., Dybjer, P., and Norell, U. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics* (2009), Springer, pp. 73–78.

[4] Breitner, J. Agda: Mit starken Typen abhängen.

[5] Curry, H. B. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America 20*, 11 (1934), 584.

[6] Dybjer, P. An Introduction to Programming and Proving in Agda.

[7] Kokke, W., Siek, J. G., and Wadler, P. Programming language foundations in agda. *Science of Computer Programming 194* (2020), 102440.

[8] Norell, U. Dependently typed programming in agda. In *International school on advanced functional programming* (2008), Springer, pp. 230–266.

[9] Stump, A. *Verified functional programming in Agda*. Morgan & Claypool, 2016.

[10] Team, T. A. Agda User Manual, Release 2.6.3, 2021.

[11] Van Der Walt, P., and Swierstra, W. Engineering proof by reflection in agda. In *Symposium on Implementation and Application of Functional Languages* (2012), Springer, pp. 157–173.

Diskussion

Fragen und Antworten