Dependent Types
Grundlagen und Beispiele

Johannes Waldmann, HTWK Leipzig

17. Mai 2018, OS FKPS, Leipzig

Definition, Beispiele, Motivation

- *Typ* = Menge von Werten.

```
data Nat : Set where
  zero : Nat ; succ : Nat -> Nat
data List : (A : Set) -> Set where
  nil : List A ; cons : A -> List A -> List A
```

- Ein *(daten-)abhängiger Typ* (dependent type, DT) ist ein Typ, der von Elementen anderer Typen abhängt.

```
data Vec : (n : Nat) -> (A : Set) -> Set
  nil :: Vec zero A
  cons : A -> Vec n A -> Vec (succ n) A
```

- Man verwendet DT, um Eigenschaften von Programmen zu spezifizieren und die Einhaltung dieser Eigenschaften exakt und soweit möglich automatisiert zu prüfen. Beispiel:

```
head : (A : Set) -> (n : Nat) -> Vec (succ n)
head (cons x _) = x
```


Programme und Logik

- die Curry-Howard-Isomorphie:
  - Typ ≈ Aussage (der Typ ist nicht leer)
  - Programm ≈ Beweis (konstruiert Objekt des Typs)
- Programm $\lambda x.\lambda y.x$ hat Typ $\forall AB : A \to B \to A$
  Aussage $A \Rightarrow B \Rightarrow A$ ist allgemeingültig
- Konjunktion:

```
data _&&_ (A B : Set) : Set where
  pair : A -> B -> A && B
```

- beweise Allgemeingültigkeit von $A \wedge B \to A$
- Disjunktion:

```
data _||_ (A B : Set) : Set where
  inl : A -> A || B ; inr : B -> A || B
```


Zur Geschichte von DT und Agda

- William A. Howard: *The Formulae-As-Types Notion of Construction*, 1969,
  Nachdruck in: Roger Hindley, Jonathan P. Seldin (Hrsg.): *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press 1980
- Peer Martin-Löf: *An Intuitionistic Theory of Types*, Techn. Report Univ. Stockholm, 1972
- Thierry Coquand, Lennart Augustsson: *ALF (A Logical Framework)*, ca. 1990, Göteborg
- Ulf Norell, *Towards a practical programming language based on dependent type theory*, Dissertation, Göteborg, 2007

Andere Systeme und Sprachen mit DT

- Coq (Calculus of Constructions)
  `https://coq.inria.fr/`.
  Theirry Coquand, Gerard Huet 1984; Christine Paulin 1991
- Lennart Augustsson: *Cayenne - a Language with Dependent Types*, ICFP 1998.
  ("Cayenne – hotter than Haskell")
- `https://www.idris-lang.org/`.
  Edwin Brady: *Idris - Systems Programming meets Full Dependent Types*, PLPV 2011.
- Simulation von DT in Haskell: generalized algebraic data types (2006)

Negation

- leerer Typ ≈ falsche Aussage

```
data False : Set where -- keine Konstruktoren!
```

- intuitionistische Form der Negation:

```
Not : Set -> Set ; Not A = A -> False
```

  Aussage ist falsch, wenn Widerspruch abgeleitet werden kann
- Vorsicht, `A || Not A` ist nicht allgemeingültig
- Beweise:

```
{ A B : Set }
   -> (not A || not B) -> not (A && B)
{ A B : Set }
  -> (not A && not B) -> not (A || B)
```


Beweise für Gleichungen

- Gleichheit wird modelliert durch Typkonstruktor

```
data _==_ { A : Set } -> A -> A -> Set where
  refl : (a : A) -> a == a
```

- Bsp: `refl 5 : (5 == 5)`
- symmetrisch, transitiv, kongruent
- Bsp: Kommutativität der (Peano-)Addition

```
plus-comm : (x : Nat) -> (y : Nat)
  -> plus x y == plus y x
```

  wird bewiesen durch Angabe einer Funktion mit diesem Typ

Induktion

- das Induktionsschema für Peano-Zahlen

```
fold : { R : Set }
  -> R -> (R -> R) -> Nat -> R
fold z s zero = z
fold z s (succ x) = s (fold z s x)
```

- verwenden zur Defition von Programmen (dabei `R = Nat`)

```
plus x y = fold y succ x
mult x y = fold zero (plus y) x
```

- *und* zur Konstruktion von Beweisen (dabei `R = Set`)

Stärkerer Typ für die Induktion

- bisher uniforme Induktion (`R` hängt nicht von Argument ab)

```
fold : { R : Set }
  -> R -> (R -> R) -> Nat -> R
```

- kann erweitert werden (bei gleicher Implementierung!) zu

```
fold : { R : Nat -> Set }
  -> R zero -> (R n -> R (succ n))
  -> (n : Nat) -> R n
```


Termination

- (Wdhlg.) `data False : Set` ohne Konstruktoren
- `f : (A : Set) -> False ; f x = f x`
  mit Curry-Howard (Typ = Aussage, Programm = Beweis) ist das ein Beweis für `False`?
- Lösung: `f` terminiert nicht, solche Def. sind in Agda nicht zugelassen
  (der Agda-Compiler verwendet einige syntaktische Kriterien, um Termination zu erkennen)
- in Agda sind alle Rechnungen *terminierend*
  $\Rightarrow$ die dadurch definierten Funktion *total*

Pattern Matching und Dependent Types

- ```
  data Vec (A: Set) : Nat -> Set where
    nil : Vec A zero
    cons : .. -> Vec A (succ n)
  head : {A : Set}->(n : Nat)-> Vec A (succ n) -
  head (cons x xs) = x
  ```
- diese Fkt `head` ist total (ordnet jedem Element ihres Def.-Bereiches `Vec A (succ n)` einen Wert zu)
- Bsp: auch diese Fkt ist total:

```
zipWith : (A -> B -> C)
  -> Vec A n -> Vec B n -> Vec C n
zipWith f nil nil = nil
zipWith f (cons x xs) (cons y ys) = ...
```

durch Pattern-Match im zweiten Argument wird Typ des dritten Arg. eingeschränkt

Gleichheit nach Definition und als Relation

- wir haben zwei Arten der Gleichheit benutzt
  - (definitionale Gleichheit)
    = in Funktionsdefinitionen (lhs = rhs)
  - (relationale Gleichheit)
    _==_ als Typkonstruktor, selbst definiert
- definitionalen Gleichungen orientieren (lhs $\to$ rhs),
  ergibt terminierendes Ersetzungssystem R
  alle (Typ)Ausdrücke in Beweisen werden vom Adga-System *normalisiert* (d.h., die R-Normalform wird ausgerechnet)
- relationale Gleichheiten müssen „von Hand“ (durch Angabe aller Schritte, d.h. Konstruktion von Beweisen) ungeformt werden
```