Einführung in Agda

https://tinyurl.com/bobkonf17-agda

Albert-Ludwigs-Universität Freiburg

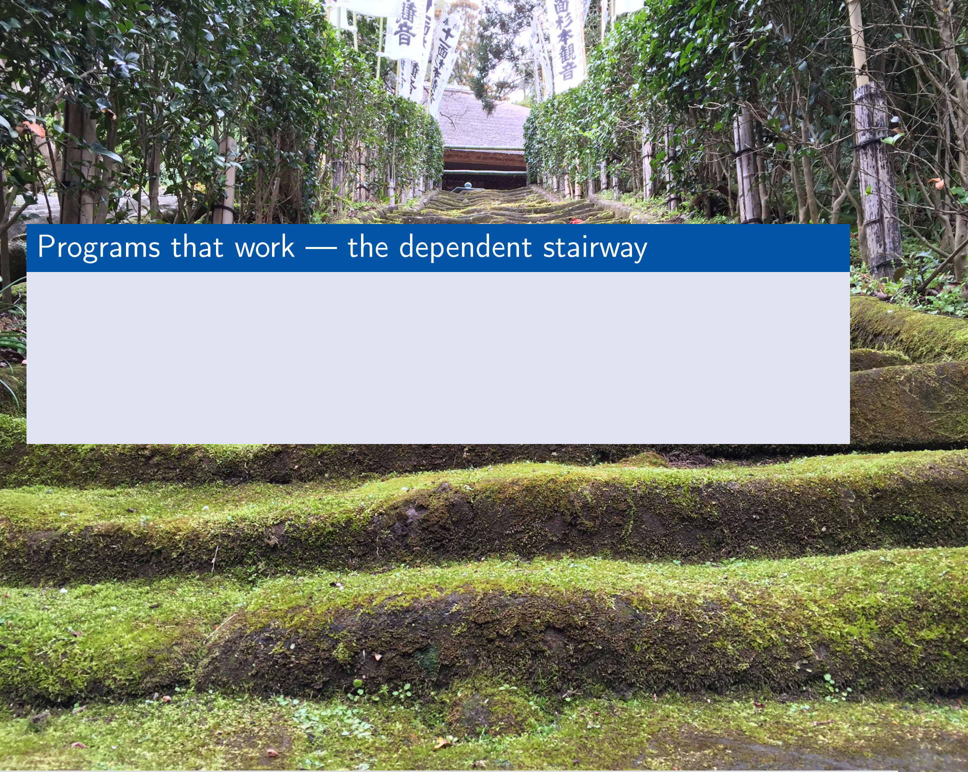
Peter Thiemann

University of Freiburg, Germany

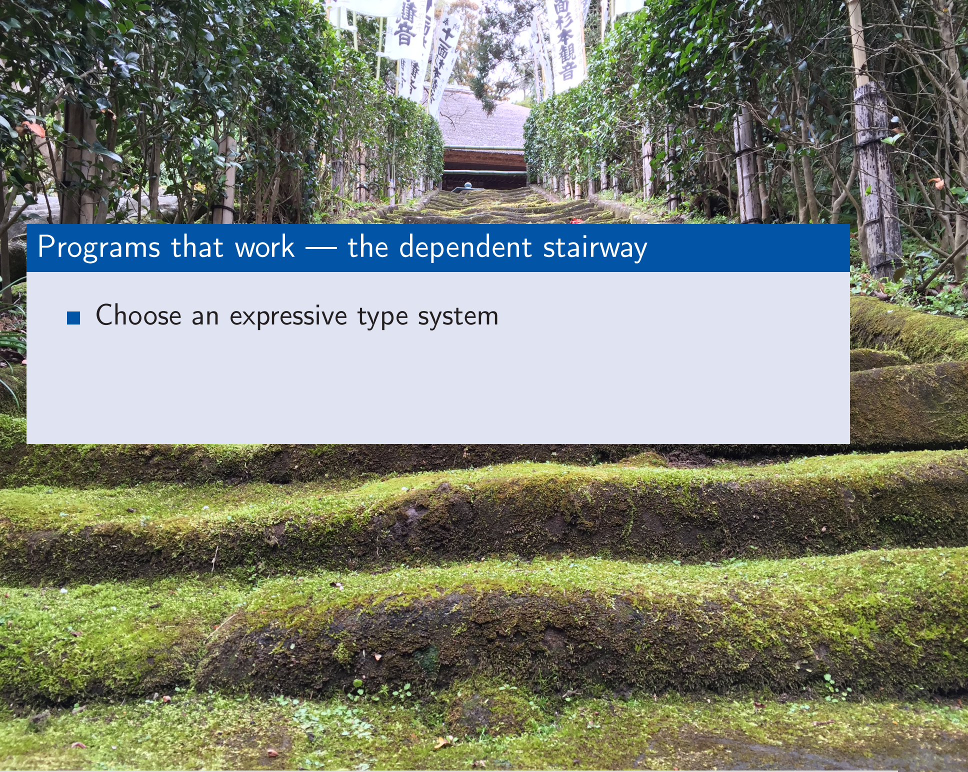
thiemann@informatik.uni-freiburg.de

24 Feb 2017


面杉本観音

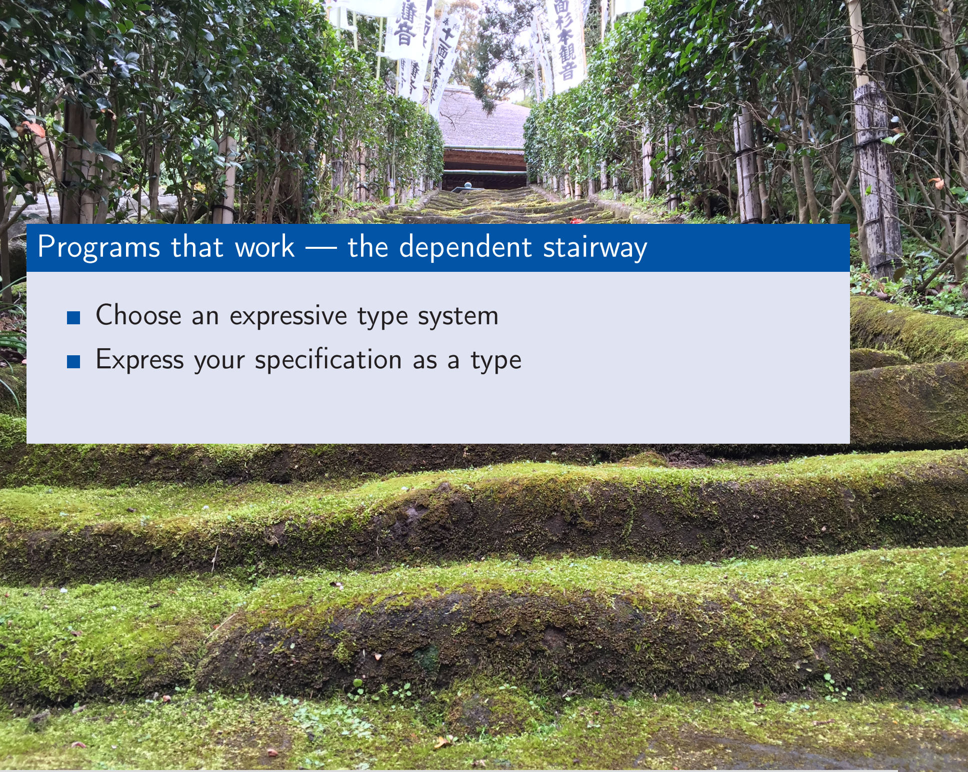


Programs that work — the dependent stairway



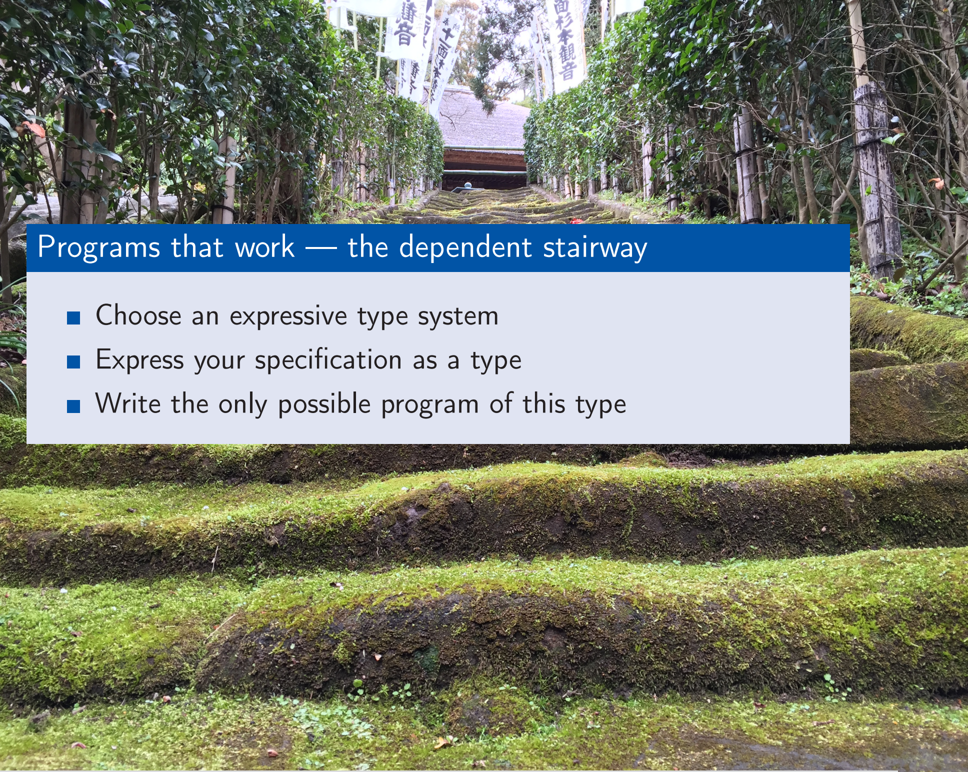
Programs that work — the dependent stairway

- Choose an expressive type system



Programs that work — the dependent stairway

- Choose an expressive type system
- Express your specification as a type



Programs that work — the dependent stairway

- Choose an expressive type system
- Express your specification as a type
- Write the only possible program of this type

Experienced Programmer

Why does it work?

The Curry-Howard Correspondence

Why does it work?

The Curry-Howard Correspondence

- Propositions as types

Why does it work?

The Curry-Howard Correspondence

- Propositions as types
- Proofs as programs

Why does it work?

The Curry-Howard Correspondence

- Propositions as types
- Proofs as programs

Central insight

Write program of this type

=

Find a proof for this proposition

In Agda

Remember Curry-Howard

- A type corresponds to a proposition
- **Elements** of the type are **proofs** for that proposition

In Agda

Remember Curry-Howard

- A type corresponds to a proposition
- **Elements** of the type are **proofs** for that proposition

The role of functions

A function $f : A \to B$ ...

In Agda

Remember Curry-Howard

- A type corresponds to a proposition
- **Elements** of the type are **proofs** for that proposition

The role of functions

A function $f : A \to B$ ...

- transforms an element of A to an element of B

In Agda

Remember Curry-Howard

- A type corresponds to a proposition
- **Elements** of the type are **proofs** for that proposition

The role of functions

A function $f : A \to B$ ...

- transforms an element of A to an element of B
- transforms a proof of A to a proof of B

In Agda

Remember Curry-Howard

- A type corresponds to a proposition
- **Elements** of the type are **proofs** for that proposition

The role of functions

A function $f : A \to B$ ...

- transforms an element of A to an element of B
- transforms a proof of A to a proof of B
- shows: if we have a proof of A, then we have a proof of B

In Agda

Remember Curry-Howard

- A type corresponds to a proposition
- **Elements** of the type are **proofs** for that proposition

The role of functions

A function $f : A \to B \ldots$

- transforms an element of A to an element of B
- transforms a proof of A to a proof of B
- shows: if we have a proof of A, then we have a proof of B
- **is** a proof of the logical implication $A \to B$

Plan

1 Prelude

2 Logic

3 Numbers

4 Vectors

5 Going further

```
- Truth
data ⊤ : Set where
  tt : ⊤
```

```
-- Truth
data ⊤ : Set where
  tt : ⊤
```

Explanation (cf. `data` in Haskell)

- -- Truth a comment
- data defines a new datatype
- ⊤ is the name of the type
- Set is its kind
- tt is the single element of ⊤

```
-- Conjunction
data _∧_ (P Q : Set) : Set where
  ⟨_,_⟩ : P → Q → (P ∧ Q)
```

```
-- Conjunction
data _∧_ (P Q : Set) : Set where
  ⟨_,_⟩ : P → Q → (P ∧ Q)
```

Explanation

- _∧_ the name of an infix type constructor
  the underlines indicate the positions of the arguments
- (P Q : Set) parameters of the type
- ⟨_,_⟩ data constructor with two parameters

```
- Disjunction
data _∨_ (P Q : Set) : Set where
  inl : P → (P ∨ Q)
  inr : Q → (P ∨ Q)
```

```
-- Disjunction
data _∨_ (P Q : Set) : Set where
  inl : P → (P ∨ Q)
  inr : Q → (P ∨ Q)
```

Explanation

- two data constructors
- everything covered

A first program in Agda

Specification

```
-- Conjunction is commutative
commConj1 : (P : Set) → (Q : Set) → (P ∧ Q) → (Q ∧ P)
```

A first program in Agda

Specification

```
-- Conjunction is commutative
```
$\text{commConj1} : (P : \text{Set}) \to (Q : \text{Set}) \to (P \wedge Q) \to (Q \wedge P)$

Explanation

- $(P : \text{Set})$ an argument of type Set with name P to be used later in the type
- $(P : \text{Set})$ and $(Q : \text{Set})$ declare that P and Q are types (propositions)
- $(P \wedge Q) \to (Q \wedge P)$ is the proposition we want to prove = the type of the program we want to write

A first program in Agda

UNI FREIBURG

Specification

```
- Conjunction is commutative
commComj1 : (P : Set) → (Q : Set) → (P ∧ Q) → (Q ∧ P)
```

Let's write it interactively

Should start with a screen like this

```
module LogicGaps where

-- Truth
data ⊤ : Set where
  tt : ⊤

-- Conjunction
data _∧_ (P Q : Set) : Set where
  ⟨_,_⟩ : (p : P) → (q : Q) → (P ∧ Q)

-- Disjunction
data _∨_ (P Q : Set) : Set where
  inl : (p : P) → (P ∨ Q)
  inr : (q : Q) → (P ∨ Q)

-- Conjunction is commutative
commConj1 : (P : Set) → (Q : Set) → P ∧ Q → Q ∧ P
commConj1 = { }0
```

```
?0 : (P Q : Set) → P ∧ Q → Q ∧ P
?1 : {P Q : Set} → P ∨ Q → Q ∨ P
?2 : {P Q R : Set} → (P ∨ Q) ∧ R → (P ∧ R) ∨ (Q ∧ R)
```

Variations on the specification

Fully explicit

```
-- Conjunction is commutative
commConj1 : (P : Set) → (Q : Set) → (P ∧ Q) → (Q ∧ P)
commConj1 P Q ⟨ p , q ⟩ = ⟨ q , p ⟩
```

- arguments P and Q are not used and Agda can infer them

Variations on the specification

Fully explicit

```
-- Conjunction is commutative
commConj1 : (P : Set) → (Q : Set) → (P ∧ Q) → (Q ∧ P)
commConj1 P Q ⟨ p , q ⟩ = ⟨ q , p ⟩
```

- arguments P and Q are not used and Agda can infer them

With inferred parameters

```
-- Conjunction is commutative
commConj2 : (P Q : Set) → (P ∧ Q) → (Q ∧ P)
commConj2 _ _ ⟨ p , q ⟩ = ⟨ q , p ⟩
```

- just put _ for inferred arguments

Variations on the specification

Implicit parameters

```
-- Conjunction is commutative
commConj : ∀ {P Q} → (P ∧ Q) → (Q ∧ P)
commConj ⟨ p , q ⟩ = ⟨ q , p ⟩
```

Variations on the specification

Implicit parameters

```
-- Conjunction is commutative
commConj : ∀ {P Q} → (P ∧ Q) → (Q ∧ P)
commConj ⟨ p , q ⟩ = ⟨ q , p ⟩
```

Explanation

- $\forall\ \{P\ Q\}$ is short for $\{P\ Q : \mathsf{Set}\}$
- $\{P\ Q : \mathsf{Set}\}$ indicates that P and Q are **implicit parameters**: they need not be provided and Agda tries to infer them
- Successful here, but we get an obscure error message if Agda cannot infer implicit parameters

A second program in Agda

Specification

```
-- Disjunction is commutative
commDisj : ∀ {P Q} → (P ∨ Q) → (Q ∨ P)
```

A second program in Agda

UNI FREIBURG

Specification

```
-- Disjunction is commutative
commDisj : ∀ {P Q} → (P ∨ Q) → (Q ∨ P)
```

Let's write it interactively

```
-- Falsity
data ⊥ : Set where

-- Negation
¬ : Set → Set
¬ P = P → ⊥
```

```
-- Falsity
data ⊥ : Set where

-- Negation
¬ : Set → Set
¬ P = P → ⊥
```

Explanation

- The type $\bot$ has **no** elements, hence no constructors
- Negation is defined by *reductio ad absurdum*: $P \to \bot$
  i.e., having a proof for P would lead to a contradiction

De Morgan's laws

Specification

```
-- DeMorgan's laws
demND1 : ∀ {P Q} → ¬ (P ∨ Q) → (¬ P ∧ ¬ Q)
demND2 : ∀ {P Q} → (¬ P ∧ ¬ Q) → ¬ (P ∨ Q)
```

De Morgan's laws

Specification

```
-- DeMorgan's laws
demND1 : ∀ {P Q} → ¬ (P ∨ Q) → (¬ P ∧ ¬ Q)
demND2 : ∀ {P Q} → (¬ P ∧ ¬ Q) → ¬ (P ∨ Q)
```

Interaction time

Plan

1 Prelude

2 Logic

3 Numbers

4 Vectors

5 Going further

Numbers in Agda

Surprise

- Numbers are not predefined in Agda
- We have to define them ourselves
- (But there is a library)

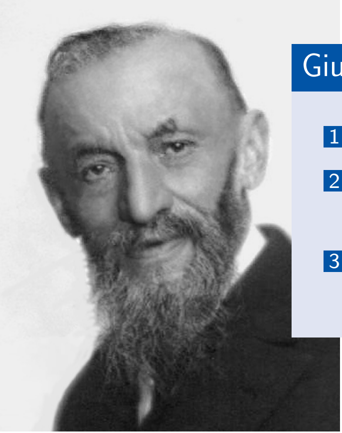
Numbers in Agda

Surprise

- Numbers are not predefined in Agda
- We have to define them ourselves
- (But there is a library)

Let's try

Peano's axioms[1]

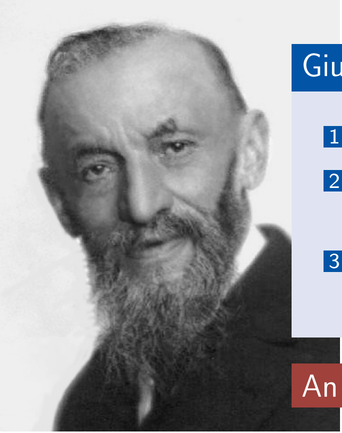


Giuseppe Peano says . . .

1. zero is a natural number
2. If n is a natural number, then suc n is also a natural number
3. All natural numbers can be (and must be) constructed from 1. and 2.

[1] Image Attribution: By Unknown - School of Mathematics and Statistics, University of St Andrews, Scotland [1], Public Domain, https://commons.wikimedia.org/w/index.php?curid=2633677

Peano's axioms[1]



Giuseppe Peano says ...

1. zero is a natural number
2. If *n* is a natural number, then suc *n* is also a natural number
3. All natural numbers can be (and must be) constructed from 1. and 2.

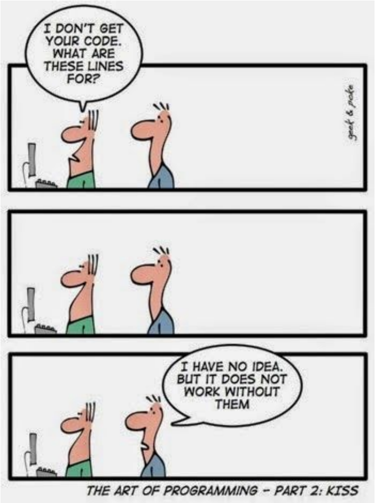
An inductive definition

[1] Image Attribution: By Unknown - School of Mathematics and Statistics, University of St Andrews, Scotland [1], Public Domain, `https://commons.wikimedia.org/w/index.php?curid=2633677`

Inductive definition in Agda

Natural numbers

$$
\begin{aligned}
&\text{data } \mathbb{N} : \text{Set where} \\
&\quad \text{zero} : \mathbb{N} \\
&\quad \text{suc} : \mathbb{N} \to \mathbb{N}
\end{aligned}
$$



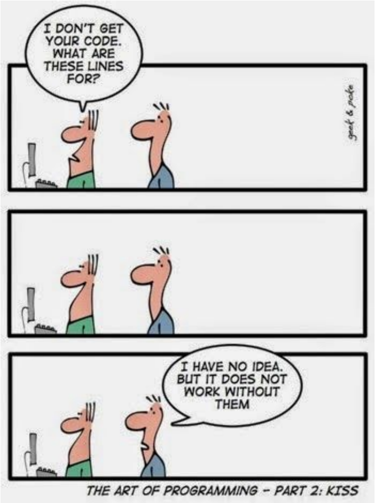
Inductive definition in Agda

Natural numbers

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

Explanation

- Defines zero and suc just like demanded by Peano
- Define functions on ℕ by induction and pattern matching on the constructors



Functional programming

Addition

$$\text{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\text{add zero } n = n$$
$$\text{add } (\text{suc } m)\ n = \text{suc } (\text{add } m\ n)$$

Functional programming

Addition

$$\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{add}\ \mathsf{zero}\ n = n$$
$$\mathsf{add}\ (\mathsf{suc}\ m)\ n = \mathsf{suc}\ (\mathsf{add}\ m\ n)$$

Subtraction

$$\mathsf{sub} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{sub}\ m\ \mathsf{zero} = m$$
$$\mathsf{sub}\ \mathsf{zero}\ (\mathsf{suc}\ n) = \mathsf{zero}$$
$$\mathsf{sub}\ (\mathsf{suc}\ m)\ (\mathsf{suc}\ n) = \mathsf{sub}\ m\ n$$

Why specify properties?

Deficiency of Testing

Testing shows the presence, not the absence of bugs.

E.W. Dijkstra

What can we specify?

- Properties of addition all require equality on numbers

What can we specify?

- Properties of addition all require equality on numbers

Next surprise

- Equality is not predefined in Agda
- We have to define it ourselves
- (But there is a library)

What can we specify?

- Properties of addition all require equality on numbers

Next surprise

- Equality is not predefined in Agda
- We have to define it ourselves
- (But there is a library)

Let's try

Inductive definition of equality

Equality on natural numbers

```
data _≡_ : ℕ → ℕ → Set where
  z≡z : zero ≡ zero
  s≡s : {m n : ℕ} → m ≡ n → suc m ≡ suc n
```

Explanation

- Unusual: datatype parameterized by two numbers
- The constructor s≡s takes a proof that $m \equiv n$ and thus becomes a proof that suc $m \equiv$ suc n

Properties of equality

Equality is ...

```
- reflexive
refl-≡ : (n : ℕ) → n ≡ n
- transitive
trans-≡ : {m n o : ℕ} → m ≡ n → n ≡ o → m ≡ o
- symmetric
symm-≡ : {m n : ℕ} → m ≡ n → n ≡ m
```

Properties of equality

Reflexivity

- Need to define a function that given some n returns a proof of (element of) $n \equiv n$
- Straightforward programming exercise
- Use pattern matching / induction
- Agda can do it automatically

Properties of equality

Reflexivity

- Need to define a function that given some n returns a proof of (element of) $n \equiv n$
- Straightforward programming exercise
- Use pattern matching / induction
- Agda can do it automatically

Interaction time

Properties of equality

Symmetry

- $m \equiv n \to n \equiv m$
- Symmetry can be proved by induction on m and n
- Introduces a new concept: absurd patterns
- Less cumbersome alternative:
  pattern matching on equality proof

Properties of equality

Symmetry

- $m \equiv n \to n \equiv m$
- Symmetry can be proved by induction on m and n
- Introduces a new concept: absurd patterns
- Less cumbersome alternative:
  pattern matching on equality proof

Interaction time

Properties of addition

Zero is neutral element of addition

$$\text{neutralAdd0l} : (m : \mathbb{N}) \to \text{add zero } m \equiv m$$
$$\text{neutralAdd0r} : (m : \mathbb{N}) \to \text{add } m \text{ zero} \equiv m$$

Properties of addition

Zero is neutral element of addition

$\text{neutralAdd0l} : (m : \mathbb{N}) \rightarrow \text{add zero } m \equiv m$
$\text{neutralAdd0r} : (m : \mathbb{N}) \rightarrow \text{add } m \text{ zero} \equiv m$

Addition is associative

$\text{assocAdd} : (m\ n\ o : \mathbb{N})$
$\quad \rightarrow \text{add } m\ (\text{add } n\ o) \equiv \text{add } (\text{add } m\ n)\ o$

Properties of addition

Zero is neutral element of addition

$\text{neutralAdd0l} : (m : \mathbb{N}) \to \text{add zero } m \equiv m$

$\text{neutralAdd0r} : (m : \mathbb{N}) \to \text{add } m \text{ zero} \equiv m$

Addition is associative

$\text{assocAdd} : (m\ n\ o : \mathbb{N})$
$\quad \to \text{add } m\ (\text{add } n\ o) \equiv \text{add } (\text{add } m\ n)\ o$

Addition is commutative

$\text{commAdd} : (m\ n : \mathbb{N}) \to \text{add } m\ n \equiv \text{add } n\ m$

Properties of addition

Proving . . .

- Neutral element and associativity are straightforward
- Commutativity is slightly more involved
- Requires an auxiliary function

Properties of addition

Proving . . .

- Neutral element and associativity are straightforward
- Commutativity is slightly more involved
- Requires an auxiliary function

Interaction time

Plan

1 Prelude

2 Logic

3 Numbers

4 Vectors

5 Going further

Vectors in Agda

Vectors with static bounds checks

- Flagship application of dependent typing
- All vector operations proved safe at compile time
- Key: define vector type indexed by its length

The vector type

```
data Vec (A : Set) : (n : ℕ) → Set where
  Nil  : Vec A zero
  Cons : {n : ℕ} → (a : A) → Vec A n → Vec A (suc n)
```

The vector type

```
data Vec (A : Set) : (n : ℕ) → Set where
  Nil  : Vec A zero
  Cons : {n : ℕ} → (a : A) → Vec A n → Vec A (suc n)
```

```
concat : ∀ {A m n}
  → Vec A m → Vec A n → Vec A (add m n)
concat Nil ys = ys
concat (Cons a xs) ys = Cons a (concat xs ys)
```

Trick #1

- Type of get depends on length of vector n and index m
- ... and a proof that $m < n$

Safe vector access
"avoid out of bound indexes"

Trick #1

- Type of get depends on length of vector n and index m
- ... and a proof that $m < n$

$$\text{get} : \forall \{A\ n\} \to \text{Vec}\ A\ n \to (m : \mathbb{N}) \to \text{suc}\ m \leq n \to A$$

Safe vector access
"avoid out of bound indexes"

Trick #1

- Type of get depends on length of vector n and index m
- ... and a proof that $m < n$

$$\text{get} : \forall \{A\ n\} \rightarrow \text{Vec}\ A\ n \rightarrow (m : \mathbb{N}) \rightarrow \text{suc}\ m \leq n \rightarrow A$$

Trick #2

- ... type restricts the index to $m < n$

Safe vector access
"avoid out of bound indexes"

Trick #1

- Type of get depends on length of vector n and index m
- … and a proof that $m < n$

$$\text{get} : \forall \{A\ n\} \rightarrow \text{Vec}\ A\ n \rightarrow (m : \mathbb{N}) \rightarrow \text{suc}\ m \leq n \rightarrow A$$

Trick #2

- … type restricts the index to $m < n$

$$\text{get1} : \forall \{A\ n\} \rightarrow \text{Vec}\ A\ n \rightarrow \text{Fin}\ n \rightarrow A$$

Finite set type

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Finite set type

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Explanation

- Overloading of constructors ok
- Fin zero $= \emptyset$ (empty set)
- Fin (suc zero) $= \{0\}$
- Fin (suc (suc zero)) $= \{0, 1\}$
- etc

Finite set type

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Explanation

- Overloading of constructors ok
- Fin zero $= \emptyset$ (empty set)
- Fin (suc zero) $= \{0\}$
- Fin (suc (suc zero)) $= \{0, 1\}$
- etc

Interaction time

Splitting a vector

We know this type already . . .

```
-- Pair
data _×_ (A B : Set) : Set where
  _,_ : (a : A) → (b : B) → (A × B)
```

```
-- split a vector in two parts
split : ∀ {A n} → Vec A n → (m : ℕ) → m ≤ n
  → Vec A m × Vec A (sub n m)
```

- Solution introduces a new feature: with matching
- This operation can also be defined with Fin . . .

Splitting a vector

We know this type already ...

```
-- Pair
data _×_ (A B : Set) : Set where
  _,_ : (a : A) → (b : B) → (A × B)
```

```
-- split a vector in two parts
split : ∀ {A n} → Vec A n → (m : ℕ) → m ≤ n
  → Vec A m × Vec A (sub n m)
```

- Solution introduces a new feature: with matching
- This operation can also be defined with Fin ...

Interaction time

Plan

1 Prelude

2 Logic

3 Numbers

4 Vectors

5 Going further

Going further

- http://learnyouanagda.liamoc.net/ nicely paced tutorial, some more background
- http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage definitive resource
- http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Othertutorials with a load of links to tutorials

Questions?