Generic Programming With Dependent Types: I
Generic Programming in Agda

Stephanie Weirich

University of Pennsylvania

March 24–26, 2010 – SSGIP
Why study generic programming in dependently-typed languages?
Why study generic programming in dependently-typed languages?

1. Dependently-typed languages are current research topic and likely component of next-generation languages.
Why study generic programming in dependently-typed languages?

1. Dependently-typed languages are current research topic and likely component of next-generation languages.

2. Generic programming is a *killer-app* for dependently-typed languages. It is a source of programs that are difficult to type check in other contexts.
Why study generic programming in dependently-typed languages?

1. Dependently-typed languages are current research topic and likely component of next-generation languages.

2. Generic programming is a killer-app for dependently-typed languages. It is a source of programs that are difficult to type check in other contexts.

3. TAKEAWAY: Dependent types are not just about program verification, they really do add to expressiveness.
Spring School Goals and Non-Goals

Goals:

Non-goals:

1. I won't argue that Agda is best tool for generic programming (it's not)
2. You won't be expert Agda programmers (I'm not)
3. I'm ignoring termination (with regards to Agda)
4. No interactive labs, sorry (try it at home!)
Spring School Goals and Non-Goals

Goals:

1. Introduction to Agda language and dependently-typed programming
Goals:

1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:

1. I won't argue that Agda is best tool for generic programming (it's not)
2. You won't be expert Agda programmers (I'm not)
3. I'm ignoring termination (with ago to Agda)
4. No interactive labs, sorry (try it at home!)
Goals:

1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:

1. I won't argue that Agda is best tool for generic programming (it's not)
2. You won't be expert Agda programmers (I'm not)
3. I'm ignoring termination (with ago to Agda)
4. No interactive labs, sorry (try it at home!)
Spring School Goals and Non-Goals

Goals:

1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:

1. I won’t argue that Agda is best tool for generic programming
Goals:

1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:

1. I won’t argue that Agda is best tool for generic programming (it’s not)
Spring School Goals and Non-Goals

Goals:
1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:
1. I won’t argue that Agda is best tool for generic programming (it’s not)
2. You won’t be expert Agda programmers
Spring School Goals and Non-Goals

Goals:
1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:
1. I won’t argue that Agda is best tool for generic programming (it’s not)
2. You won’t be expert Agda programmers (I’m not)
Spring School Goals and Non-Goals

Goals:
1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:
1. I won’t argue that Agda is best tool for generic programming (it’s not)
2. You won’t be expert Agda programmers (I’m not)
3. I’m ignoring termination
Spring School Goals and Non-Goals

Goals:

1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:

1. I won’t argue that Agda is best tool for generic programming (it’s not)
2. You won’t be expert Agda programmers (I’m not)
3. I’m ignoring termination (with flags to Agda)
Spring School Goals and Non-Goals

Goals:
1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:
1. I won’t argue that Agda is best tool for generic programming (it’s not)
2. You won’t be expert Agda programmers (I’m not)
3. I’m ignoring termination (with flags to Agda)
4. No interactive labs, sorry
Spring School Goals and Non-Goals

Goals:
1. Introduction to Agda language and dependently-typed programming
2. Extended examples of generic programming

Non-goals:
1. I won’t argue that Agda is best tool for generic programming (it’s not)
2. You won’t be expert Agda programmers (I’m not)
3. I’m ignoring termination (with flags to Agda)
4. No interactive labs, sorry (try it at home!)
Where to go for more information

3. References in the slides
4. All of the code from these slides, from my website http://www.seas.upenn.edu/~sweirich/ssgip/
What is Agda?

Agda has a dual identity:

1. A functional programming language with dependent types based on Martin-Löf intuitionistic type theory
2. A proof assistant, based on the Curry-Howard isomorphism

Historically derived from series of proof assistants and languages implemented at Chalmers. Current version (officially named Agda 2) implemented by Ulf Norell.¹

What is Agda?

Agda has a dual identity:

1. A functional programming language with dependent types based on Martin-Löf intuitionistic type theory
2. A proof assistant, based on the Curry-Howard isomorphism

Historically derived from series of proof assistants and languages implemented at Chalmers. Current version (officially named Agda 2) implemented by Ulf Norell.¹

We will focus exclusively on the first aspect.

Agda looks a bit like Haskell

- Define datatypes

```
data Bool : Set where
  true  : Bool
  false : Bool
```
Agda looks a bit like Haskell

- Define datatypes

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

- Define (infix) functions by pattern matching

```agda
    _∧_   : Bool → Bool → Bool
  true ∧ true  =  true
  _    ∧ _    =  false
```
Agda looks a bit like Haskell

- Define datatypes
  ```
  data Bool : Set where
  true : Bool
  false : Bool
  ```

- Define (infix) functions by pattern matching
  ```
  _∧_ : Bool → Bool → Bool
  true ∧ true = true
  _ ∧ _ = false
  ```

- Define mixfix/polymorphic functions
  ```
  if_then_else : ∀ {A} → Bool → A → A → A
  if true then e1 else e2 = e1
  if false then e1 else e2 = e2
  ```
Inductive datatypes

Datatypes can be inductive

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

... and used to define total recursive functions

```haskell
replicate : ∀ {A} → ℕ → A → List A
replicate zero x = []
replicate (suc n) x = x :: replicate n x
```

... and used to state properties about those functions

```haskell
replicate-spec : ∀ {A} → (x : A) → (n : ℕ) → length (replicate n x) ≡ n
```
Inductive datatypes

- Datatypes can be inductive

\[
data \mathbb{N} : \text{Set where}
\]
\[
\text{zero} : \mathbb{N}
\]
\[
\text{suc} : \mathbb{N} \rightarrow \mathbb{N}
\]

- ... and used to define total recursive functions

\[
\text{replicate} : \forall \{A\} \rightarrow \mathbb{N} \rightarrow A \rightarrow \text{List } A
\]
\[
\text{replicate} \text{ zero } x = []
\]
\[
\text{replicate} \ (\text{suc } n) \ x = (x :\! : \text{replicate } n \ x)
\]
Inductive datatypes

- Datatypes can be inductive

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

- ... and used to define total recursive functions

  ```haskell
replicate : ∀ {A} → ℕ → A → List A
replicate zero x = []
replicate (suc n) x = (x :: replicate n x)
  ```

- ... and used to state properties about those functions

  ```haskell
replicate-spec : ∀ {A} → (x : A) → (n : ℕ)
  → length (replicate n x) ≡ n
  ```
Polymorphic Length-indexed Vectors

Lists that know their length

```haskell
data Vec (A : Set) : ℕ → Set where
  []     : Vec A zero
  _∷_   : ∀ {n} → A → Vec A n → Vec A (suc n)
```
Polymorphic Length-indexed Vectors

Lists that know their length

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

Give informative types to functions

repeat : ∀ {A} → (n : ℕ) → A → Vec A n
repeat zero x = []
repeat (suc n) x = x :: repeat n x
Lengths eliminate bugs

List "zap"

\[
_\circ_ : \forall \{A, B\} \rightarrow \text{List} (A \rightarrow B) \rightarrow \text{List} A \rightarrow \text{List} B
\]

\[
\begin{align*}
\text{[]} & \circ \text{[]} = \text{[]} \\
(a :: As) & \circ (b :: Bs) = (a b :: As \circ Bs) \\
_ & \circ _ = \text{error}
\end{align*}
\]
Lengths eliminate bugs

**List "zap"**

\[ \_ \circ \_ : \forall \{ A B \} \rightarrow \text{List} (A \rightarrow B) \rightarrow \text{List} A \rightarrow \text{List} B \]
\[
\begin{array}{c}
\text{[]} \circ \text{[]} = \text{[]} \\
(a :: A) \circ (b :: B) = (a b :: A \circ B)
\end{array}
\]
\[
\_ \circ \_ = \text{error}
\]

**Vec "zap"**

\[ \_ \ast \_ : \forall \{ A B \text{ n} \} \rightarrow \text{Vec} (A \rightarrow B) \text{ n} \rightarrow \text{Vec} A \text{ n} \rightarrow \text{Vec} B \text{ n} \]
\[
\begin{array}{c}
\text{[]} \ast \text{[]} = \text{[]} \\
(a :: A) \ast (b :: B) = (a b :: A \ast B)
\end{array}
\]
Main thesis:
Dependently typed languages are not just for eliminating bugs, they enable Generic Programming.
Generic programming in Agda

Main thesis:
Dependently typed languages are not just for eliminating bugs, they enable Generic Programming

But what is generic programming?
Main thesis:
Dependently typed languages are not just for eliminating bugs, they enable Generic Programming

But what is generic programming? Lots of different definitions, but they all boil down to lifting data structures and algorithms from concrete instances to general forms.
### Generalizing programs

#### Specific cases

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>zerox</code></td>
<td><code>(Bool \rightarrow Bool) \rightarrow Bool \rightarrow Bool</code></td>
<td><code>zerox f x = x</code></td>
</tr>
<tr>
<td><code>onex</code></td>
<td><code>(Bool \rightarrow Bool) \rightarrow Bool \rightarrow Bool</code></td>
<td><code>onex f x = f x</code></td>
</tr>
<tr>
<td><code>twox</code></td>
<td><code>(Bool \rightarrow Bool) \rightarrow Bool \rightarrow Bool</code></td>
<td><code>twox f x = f (f x)</code></td>
</tr>
</tbody>
</table>
Generalizing programs

Specific cases

- \( \text{zerox} : (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool} \)
  
  \[ \text{zerox} \ f \ x = x \]

- \( \text{onex} : (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool} \)
  
  \[ \text{onex} \ f \ x = f \ x \]

- \( \text{twox} : (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool} \)
  
  \[ \text{twox} \ f \ x = f \ (f \ x) \]

Add extra argument

Generic function

- \( \text{nx} : \mathbb{N} \to (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool} \)
  
  \[ \text{nx} \ \text{zero} \ f \ x = x \]

  \[ \text{nx} \ (\text{suc} \ n) \ f \ x = \text{nx} \ n \ f \ (f \ x) \]
Parametric polymorphism

Specific cases

app-nat : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}
app-nat \ f \ x = f \ x

app-bool : (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \text{Bool}
app-bool \ f \ x = f \ x
Parametric polymorphism

Specific cases

\[
\text{app-nat} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}
\]
\[
\text{app-nat} f \ x = f \ x
\]

\[
\text{app-bool} : (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}
\]
\[
\text{app-bool} f \ x = f \ x
\]

New argument could be an implicit, parametric type

Generic function

\[
\text{app} : \forall \{A\} \to (A \to A) \to A \to A
\]
\[
\text{app} f \ x = f \ x
\]
Ad hoc polymorphism

\[
\begin{align*}
\text{eq-nat} & : \mathbb{N} \to \mathbb{N} \to \mathbb{B} \\
\text{eq-nat} \ z \ z & = \text{true} \\
\text{eq-nat} (\text{suc} \ n) (\text{suc} \ m) & = \text{eq-nat} \ n \ m \\
\text{eq-nat} \ _ \ _ & = \text{false} \\
\text{eq-bool} & : \mathbb{B} \to \mathbb{B} \to \mathbb{B} \\
\text{eq-bool} \ \text{false} \ \text{false} & = \text{true} \\
\text{eq-bool} \ \text{true} \ \text{true} & = \text{true} \\
\text{eq-bool} \ _ \ _ & = \text{false}
\end{align*}
\]
Ad hoc polymorphism

\[
\text{eq-nat} : \mathbb{N} \to \mathbb{N} \to \text{Bool}
\]
\[
\text{eq-nat} \; \text{zero} \; \text{zero} = \text{true}
\]
\[
\text{eq-nat} \; (\text{suc} \; n) \; (\text{suc} \; m) = \text{eq-nat} \; n \; m
\]
\[
\text{eq-nat} \; _\_ \; _ = \text{false}
\]

\[
\text{eq-bool} : \text{Bool} \to \text{Bool} \to \text{Bool}
\]
\[
\text{eq-bool} \; \text{false} \; \text{false} = \text{true}
\]
\[
\text{eq-bool} \; \text{true} \; \text{true} = \text{true}
\]
\[
\text{eq-bool} \; _\_ \; _ = \text{false}
\]

\[
\llbracket \_ \rrbracket : \text{Bool} \to \text{Set}
\]
\[
\llbracket \_ b \rrbracket = \text{if} \; b \; \text{then} \; \mathbb{N} \; \text{else} \; \text{Bool}
\]

\[
\text{eq-nat-bool} : (b : \text{Bool}) \to \llbracket \_ b \rrbracket \to \llbracket \_ b \rrbracket \to \text{Bool}
\]
\[
\text{eq-nat-bool} \; \text{true} = \text{eq-nat}
\]
\[
\text{eq-nat-bool} \; \text{false} = \text{eq-bool}
\]
General idea: “universes” for generic programming

Start with a “code” for types:

```
data Type : Set where
    nat : Type
    bool : Type
    pair : Type → Type → Type
```
General idea: “universes” for generic programming

- Start with a “code” for types:

  ```
  data Type : Set where
  nat : Type
  bool : Type
  pair : Type → Type → Type
  ```

- Define an “interpretation” as an Agda type

  ```
  \[
  \begin{align*}
  \mathcal{[} \_ \mathcal{]} : \ & \text{Type} \rightarrow \text{Set} \\
  \mathcal{[} \text{nat} \mathcal{]} & = \mathbb{N} \\
  \mathcal{[} \text{bool} \mathcal{]} & = \text{Bool} \\
  \mathcal{[} \text{pair} \ t_1 \ t_2 \mathcal{]} & = \mathcal{[} \ t_1 \mathcal{]} \times \mathcal{[} \ t_2 \mathcal{]}
  \end{align*}
  \]
  ```
General idea: “universes” for generic programming

- Start with a “code” for types:
  
  ```
  data Type : Set where
  nat : Type
  bool : Type
  pair : Type → Type → Type
  ```

- Define an “interpretation” as an Agda type
  
  ```
  [ ] : Type → Set
  [ nat ] = ℕ
  [ bool ] = Bool
  [ pair t₁ t₂ ] = [ t₁ ] × [ t₂ ]
  ```

- Then define generic function, dispatching on code
  
  ```
  eq : (t : Type) → [ t ] → [ t ] → Bool
  eq nat  x    y    = eq-nat x y
  eq bool x    y    = eq-bool x y
  eq (pair t₁ t₂) (x₁,x₂) (y₁,y₂) = eq t₁ x₁ y₁ ∧ eq t₂ x₂ y₂
  ```
Expressiveness

Patterns in both types and definitions

zeroApp : \( \forall \{ A \ B \} \rightarrow B \rightarrow A \rightarrow B \)
zeroApp \( f \ x = f \)

oneApp : \( \forall \{ A \ B \} \rightarrow (A \rightarrow B) \rightarrow A \rightarrow B \)
oneApp \( f \ x = f \ x \)

twoApp : \( \forall \{ A \ B \} \rightarrow (A \rightarrow A \rightarrow B) \rightarrow A \rightarrow B \)
twoApp \( f \ x = f \ x \ x \)
Expressiveness

Patterns in both types and definitions

\[
\begin{align*}
\text{zeroApp} & : \forall \{ A \ B \} \rightarrow B \rightarrow A \rightarrow B \\
\text{zeroApp} \ f \ x & = f \\
\text{oneApp} & : \forall \{ A \ B \} \rightarrow (A \rightarrow B) \rightarrow A \rightarrow B \\
\text{oneApp} \ f \ x & = f \ x \\
\text{twoApp} & : \forall \{ A \ B \} \rightarrow (A \rightarrow A \rightarrow B) \rightarrow A \rightarrow B \\
\text{twoApp} \ f \ x & = f \ x \ x
\end{align*}
\]

\[
\begin{align*}
\text{NAPP} & : \mathbb{N} \rightarrow \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \\
\text{NAPP} \ \text{zero} & \ A \ B \ = \ B \\
\text{NAPP} \ (\text{suc} \ n) \ A \ B & = A \rightarrow \text{NAPP} \ n \ A \ B \\
\text{nApp} & : \forall \{ A \ B \} \rightarrow (n : \mathbb{N}) \rightarrow \text{NAPP} \ n \ A \ B \rightarrow A \rightarrow B \\
\text{nApp} \ \text{zero} & \ f \ x \ = \ f \\
\text{nApp} \ (\text{suc} \ n) \ f \ x & = \text{nApp} \ n \ (f \ x) \ x
\end{align*}
\]
Key features of advanced generic programming

**Strong elimination**

- if \( b \) then \( \mathbb{N} \) else \( \text{Bool} \)
- Static case analysis on data to produce a type
Key features of advanced generic programming

<table>
<thead>
<tr>
<th>Strong elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>- if b then ( \mathbb{N} ) else Bool</td>
</tr>
<tr>
<td>- Static case analysis on data to produce a type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependent pattern matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f : (b : \text{Bool}) \rightarrow \text{if b then } \mathbb{N} \text{ else Bool} )</td>
</tr>
<tr>
<td>( f \text{ true } = 0 )</td>
</tr>
<tr>
<td>( f \text{ false } = \text{false} )</td>
</tr>
<tr>
<td>- Dynamic case analysis on data refines types</td>
</tr>
</tbody>
</table>
Key features of advanced generic programming

**Strong elimination**

- if b then $\mathbb{N}$ else Bool
- Static case analysis on data to produce a type

**Dependent pattern matching**

\[
f : (b : \text{Bool}) \rightarrow \text{if b then } \mathbb{N} \text{ else Bool}
\]

\[
f \text{true} = 0
\]

\[
f \text{false} = \text{false}
\]

- Dynamic case analysis on data refines types

**Overall**

Uniform extension of the notion of programmability from run-time to compile-time.
Proof checking vs. Programming

- Two uses for Agda are in conflict
Proof checking vs. Programming

- Two uses for Agda are in conflict
- Under the Curry-Howard isomorphism, only terminating programs are proofs.
Two uses for Agda are in conflict

Under the Curry-Howard isomorphism, only terminating programs are proofs. An infinite loop has any type, so can prove any property

\[
\text{anything} : \forall \{A\} \rightarrow A
\]
\[
\text{anything} = \text{anything}
\]
Proof checking vs. Programming

- Two uses for Agda are in conflict
- Under the Curry-Howard isomorphism, only terminating programs are proofs. An infinite loop has any type, so can prove any property

\[
\text{anything} : \forall \{A\} \rightarrow A
\]
\[
\text{anything} = \text{anything}
\]

- By default, Agda only accepts programs that it can show terminate.
Proof checking vs. Programming

To prove that all programs terminate, Agda makes strong restrictions on definitions

1. Predicative polymorphism
2. Structural recursive functions
3. Strictly-positive datatypes

Restrictions hinder *compile-time programmability*. 
Proof checking vs. Programming

To prove that all programs terminate, Agda makes strong restrictions on definitions

1. Predicative polymorphism
2. Structural recursive functions
3. Strictly-positive datatypes

Restrictions hinder *compile-time programmability*. So, we remove them with flags
Proof checking vs. Programming

To prove that all programs terminate, Agda makes strong restrictions on definitions

1. Predicative polymorphism --type-in-type
2. Structural recursive functions
3. Strictly-positive datatypes

Restrictions hinder compile-time programmability. So, we remove them with flags
Proof checking vs. Programming

To prove that all programs terminate, Agda makes strong restrictions on definitions:

1. Predicative polymorphism --type-in-type
2. Structural recursive functions --no-termination-check
3. Strictly-positive datatypes

Restrictions hinder *compile-time programmability*. So, we remove them with flags.
Proof checking vs. Programming

To prove that all programs terminate, Agda makes strong restrictions on definitions

1. Predicative polymorphism `--type-in-type`
2. Structural recursive functions `--no-termination-check`
3. Strictly-positive datatypes `--no-positivity-check`

Restrictions hinder *compile-time programmability*. So, we remove them with flags
Types are first-class data

They may be:

- Passed to functions, dependently or non-dependently
  
  \[
  f : \text{Set} \to \text{Set}
  \]
  
  \[
  f(x) = (x \to x)
  \]

  \[
  g : (A : \text{Set}) \to A \to A
  \]
  
  \[
  g_A(x) = x
  \]

- Returned as results, dependently or non-dependently

  \[
  h : \text{Bool} \to \exists (\lambda A \to A)
  \]
  
  \[
  h(x) = \begin{cases} (\text{Nat}, 0) & \text{if } x \text{ then} \\ (\text{Bool}, \text{true}) & \text{else} \end{cases}
  \]

- Stored in data structures

  \[
  (\text{Nat} :: \text{Bool} :: \text{Vec} \text{Nat} :: \text{List} \text{Set})
  \]
Types are first-class data

They may be:

- Passed to functions, dependently or non-dependently

\[ f : \text{Set} \rightarrow \text{Set} \]
\[ f \ x = (x \rightarrow x) \]
\[ g : (A : \text{Set}) \rightarrow A \rightarrow A \]
\[ g\ A\ x = x \]
Types are first-class data

They may be:

- Passed to functions, dependently or non-dependently

  \[ f : \text{Set} \rightarrow \text{Set} \]
  \[ f \, x \, = \, (x \rightarrow x) \]
  \[ g : (A : \text{Set}) \rightarrow A \rightarrow A \]
  \[ g \, A \, x \, = \, x \]

- Returned as results, dependently or non-dependently

  \[ h : \text{Bool} \rightarrow \exists (\lambda A \rightarrow A) \]
  \[ h \, x \, = \, \text{if } x \, \text{then } (\mathbb{N},0) \, \text{else } (\text{Bool},\text{true}) \]
Types are first-class data

They may be:

- Passed to functions, dependently or non-dependently

\[ f : \text{Set} \rightarrow \text{Set} \]
\[ f x = (x \rightarrow x) \]
\[ g : (A : \text{Set}) \rightarrow A \rightarrow A \]
\[ g A x = x \]

- Returned as results, dependently or non-dependently

\[ h : \text{Bool} \rightarrow \exists (\lambda A \rightarrow A) \]
\[ h x = \text{if } x \text{ then } (\mathbb{N},0) \text{ else } (\text{Bool},\text{true}) \]

- Stored in data structures

\[ (\mathbb{N} :: \text{Bool} :: \text{Vec} \mathbb{N} 3 :: []) : \text{List Set} \]
Both types and regular data may be inferred by the type checker (as implicit arguments) and symbolically evaluated at compile time.
Both types and regular data may be inferred by the type checker (as implicit arguments) and symbolically evaluated at compile time.

What is the difference between types and other sorts of data?
Both types and regular data may be inferred by the type checker (as implicit arguments) and symbolically evaluated at compile time.

What is the difference between types and other sorts of data?
- Types can be used to classify data.
Both types and regular data may be inferred by the type checker (as implicit arguments) and symbolically evaluated at compile time.

What is the difference between types and other sorts of data?
- Types can be used to classify data.
- Types are classified by Set
Both types and regular data may be inferred by the type checker (as implicit arguments) and symbolically evaluated at compile time.

What is the difference between types and other sorts of data?

- Types can be used to classify data.
- Types are classified by `Set`

i.e. if $\Gamma \vdash e : t$ then $\Gamma \vdash t : \text{Set}$ and $t$ is a type.
What about Set? What is its type?
What about \texttt{Set}? What is its type?

- Flag \texttt{--type-in-type} enables $\Gamma \vdash \texttt{Set} : \texttt{Set}$
What about \texttt{Set}? What is its type?

- Flag \texttt{--type-in-type} enables $\Gamma \vdash \texttt{Set} : \texttt{Set}$
- Convenient for polymorphic data structures:

\[
\begin{align*}
\text{head} & : \forall \{A \ n\} \to \texttt{Vec} \ A \ (\texttt{suc} \ n) \to A \\
\text{head} \ (x :: xs) & = x
\end{align*}
\]
What about \texttt{Set}? What is its type?

- Flag \texttt{--type-in-type} enables $\Gamma \vdash \texttt{Set} : \texttt{Set}$
- Convenient for polymorphic data structures:

\[
\begin{align*}
\text{head} : & \forall \{A \ n\} \rightarrow \text{Vec} \ A \ (\text{suc} \ n) \rightarrow A \\
\text{head} \ (x :: xs) &= x \\
\end{align*}
\]

\[
\begin{align*}
x : & \texttt{Set} \\
x &= \text{head} \ (\mathbb{N} :: \texttt{Bool} :: []) \\
\end{align*}
\]
What about termination?

Type soundness is independent of termination. So even if we don’t know that programs terminate, we still know that they will not crash.

\[ \forall \{ A \} \rightarrow \text{Vec} A (\text{succ} n) \rightarrow A \]

Have partial correctness: the program is correct up to termination. Caveats: Invalid proofs can also cause programs to diverge. (And can’t erase them either!) Implications are not to be trusted.
What about termination?

Type soundness is independent of termination. So even if we don’t know that programs terminate, we still know that they will not crash.
Knowing that we don’t need an additional case in head is independent of termination.

\[
\text{head} : \forall \{ A \ n \} \rightarrow \text{Vec} \ A \ (\text{suc} \ n) \rightarrow A \\
\text{head} \ (x :: xs) = x
\]
What about termination?

Type soundness is independent of termination. So even if we don’t know that programs terminate, we still know that they will not crash. Knowing that we don’t need an additional case in `head` is independent of termination.

\[
\text{head : } \forall \{A \ n\} \rightarrow \text{Vec} \ A \ (\text{suc} \ n) \rightarrow A \\
\text{head} \ (x :: xs) = x
\]

Have *partial correctness*: the program is correct up to termination. Caveats:

- Invalid proofs can also cause programs to diverge. (And can’t erase them either!)
- Implications are not to be trusted.
Coming next...

Two intensive examples of generic programming in Agda...
Two intensive examples of generic programming in Agda...

1 Kind-indexed type-directed programming
Coming next...

Two intensive examples of generic programming in Agda...

1. Kind-indexed type-directed programming
2. Arity-generic programming