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:

Spring School Goals and Non-Goals

Goals:

1. Introduction to Agda language and dependently-typed programming

Spring School Goals and Non-Goals

Goals:

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

Spring School Goals and Non-Goals

Goals:

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

Non-goals:

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

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)

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

1. Agda Wiki `http://wiki.portal.chalmers.se/agda/`
2. Stephanie Weirich and Chris Casinghino. Arity-generic type-generic programming. In *ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 15–26, January 2010
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]

---

[1] See Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

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.

[1] See Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Agda looks a bit like Haskell

- Define datatypes

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

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
```

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

$$
\begin{aligned}
\textbf{data}\ \mathbb{N} &: \mathsf{Set}\ \textbf{where} \\
\mathsf{zero} &: \mathbb{N} \\
\mathsf{suc} &: \mathbb{N} \longrightarrow \mathbb{N}
\end{aligned}
$$

Inductive datatypes

- Datatypes can be inductive

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

- ... and used to define total recursive functions

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

Inductive datatypes

- Datatypes can be inductive

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

- ... and used to define total recursive functions

```
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

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

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"

$$
\begin{array}{llll}
\_\odot\_ & : \forall \{A\ B\} \to \mathsf{List}\ (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B & & \\
[] & \odot\ [] & = & [] \\
(a :: As) & \odot\ (b :: Bs) & = & (a\ b :: As \odot Bs) \\
\_ & \odot\ \_ & = & \mathsf{error}
\end{array}
$$

Lengths eliminate bugs

List "zap"

$$
\begin{array}{lcl}
\_\odot\_ & : & \forall \{A\ B\} \rightarrow \mathsf{List}\ (A \rightarrow B) \rightarrow \mathsf{List}\ A \rightarrow \mathsf{List}\ B \\
[] \ \odot\ [] & = & [] \\
(a :: As) \odot (b :: Bs) & = & (a\ b :: As \odot Bs) \\
\_ \ \odot\ \_ & = & \mathsf{error}
\end{array}
$$

Vec "zap"

$$
\begin{array}{lcl}
\_\circledast\_ & : & \forall \{A\ B\ n\} \rightarrow \mathsf{Vec}\ (A \rightarrow B)\ n \rightarrow \mathsf{Vec}\ A\ n \rightarrow \mathsf{Vec}\ B\ n \\
[] \ \circledast\ [] & = & [] \\
(a :: As) \circledast (b :: Bs) & = & (a\ b :: As \circledast Bs)
\end{array}
$$

Generic programming in Agda

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?

Generic programming in Agda

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

$$\begin{aligned}
\mathsf{zerox} &: (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
\mathsf{zerox}\ f\ x &= x
\end{aligned}$$

$$\begin{aligned}
\mathsf{onex} &: (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
\mathsf{onex}\ f\ x &= f\ x
\end{aligned}$$

$$\begin{aligned}
\mathsf{twox} &: (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
\mathsf{twox}\ f\ x &= f\ (f\ x)
\end{aligned}$$

Generalizing programs

Specific cases

$$\begin{aligned}
\mathsf{zerox} &: (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
\mathsf{zerox}\ f\ x &= x \\
\mathsf{onex} &: (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
\mathsf{onex}\ f\ x &= f\ x \\
\mathsf{twox} &: (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
\mathsf{twox}\ f\ x &= f\ (f\ x)
\end{aligned}$$

Add extra argument

Generic function

$$\begin{aligned}
&\mathsf{nx} : \mathbb{N} \to (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
&\mathsf{nx}\ \mathsf{zero}\ f\ x = x \\
&\mathsf{nx}\ (\mathsf{suc}\ n)\ f\ x = \mathsf{nx}\ n\ f\ (f\ x)
\end{aligned}$$

Parametric polymorphism

Specific cases

$$
\begin{aligned}
\text{app-nat} &: (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \\
\text{app-nat}\ \mathsf{f}\ \mathsf{x} &= \mathsf{f}\ \mathsf{x}
\end{aligned}
$$

$$
\begin{aligned}
\text{app-bool} &: (\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool} \\
\text{app-bool}\ \mathsf{f}\ \mathsf{x} &= \mathsf{f}\ \mathsf{x}
\end{aligned}
$$

Parametric polymorphism

Specific cases

$$\begin{aligned} \text{app-nat} &: (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \\ \text{app-nat}\ f\ x &= f\ x \end{aligned}$$

$$\begin{aligned} \text{app-bool} &: (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool} \\ \text{app-bool}\ f\ x &= f\ x \end{aligned}$$

New argument could be an implicit, parametric type

Generic function

$$\begin{aligned} \text{app} &: \forall\ \{A\} \to (A \to A) \to A \to A \\ \text{app}\ f\ x &= f\ x \end{aligned}$$

Ad hoc polymorphism

$$
\begin{array}{llll}
\text{eq-nat} & : \mathbb{N} \to \mathbb{N} \to \text{Bool} & & \\
\text{eq-nat zero} & \text{zero} & = & \text{true} \\
\text{eq-nat (suc n)} & \text{(suc m)} & = & \text{eq-nat n m} \\
\text{eq-nat \_} & \text{\_} & = & \text{false} \\
\text{eq-bool} & : \text{Bool} \to \text{Bool} \to \text{Bool} & & \\
\text{eq-bool false} & \text{false} & = & \text{true} \\
\text{eq-bool true} & \text{true} & = & \text{true} \\
\text{eq-bool \_} & \text{\_} & = & \text{false}
\end{array}
$$

Ad hoc polymorphism

```
eq-nat  :  ℕ → ℕ → Bool
eq-nat zero    zero     =  true
eq-nat (suc n) (suc m)  =  eq-nat n m
eq-nat _       _        =  false

eq-bool  :  Bool → Bool → Bool
eq-bool false  false    =  true
eq-bool true   true     =  true
eq-bool _      _        =  false
```

```
⟦_⟧  :  Bool → Set
⟦ b ⟧  =  if b then ℕ else Bool

eq-nat-bool  :  (b : Bool) → ⟦ b ⟧ → ⟦ b ⟧ → Bool
eq-nat-bool true   =  eq-nat
eq-nat-bool false  =  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

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{Type} \to \mathsf{Set} \\
&[\![\ \mathsf{nat}\ ]\!] = \mathbb{N} \\
&[\![\ \mathsf{bool}\ ]\!] = \mathsf{Bool} \\
&[\![\ \mathsf{pair}\ \mathsf{t}_1\ \mathsf{t}_2\ ]\!] = [\![\ \mathsf{t}_1\ ]\!] \times [\![\ \mathsf{t}_2\ ]\!]
\end{aligned}
$$

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

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{Type} \to \mathsf{Set} \\
&[\![\,\mathsf{nat}\,]\!] = \mathbb{N} \\
&[\![\,\mathsf{bool}\,]\!] = \mathsf{Bool} \\
&[\![\,\mathsf{pair}\ t_1\ t_2\,]\!] = [\![\,t_1\,]\!] \times [\![\,t_2\,]\!]
\end{aligned}
$$

- Then define generic function, dispatching on code

$$
\begin{aligned}
&\mathsf{eq} : (t : \mathsf{Type}) \to [\![\,t\,]\!] \to [\![\,t\,]\!] \to \mathsf{Bool} \\
&\mathsf{eq}\ \mathsf{nat}\ x\ y = \mathsf{eq\text{-}nat}\ x\ y \\
&\mathsf{eq}\ \mathsf{bool}\ x\ y = \mathsf{eq\text{-}bool}\ x\ y \\
&\mathsf{eq}\ (\mathsf{pair}\ t_1\ t_2)\ (x_1,x_2)\ (y_1,y_2) = \mathsf{eq}\ t_1\ x_1\ y_1 \wedge \mathsf{eq}\ t_2\ x_2\ y_2
\end{aligned}
$$

Expressiveness

Patterns in both types and definitions

```
zeroApp : ∀ {A B} → B → A → B
zeroApp f x = f
oneApp  : ∀ {A B} → (A → B) → A → B
oneApp  f x = f x
twoApp  : ∀ {A B} → (A → A → B) → A → B
twoApp  f x = f x x
```

Expressiveness

Patterns in both types and definitions

```
zeroApp : ∀ {A B} → B → A → B
zeroApp f x = f

oneApp  : ∀ {A B} → (A → B) → A → B
oneApp  f x = f x

twoApp  : ∀ {A B} → (A → A → B) → A → B
twoApp  f x = f x x
```

```
NAPP : ℕ → Set → Set → Set
NAPP zero    A B = B
NAPP (suc n) A B = A → NAPP n A B

nApp : ∀ {A B} → (n : ℕ) → NAPP n A B → A → B
nApp zero    f x = f
nApp (suc n) f x = nApp n (f x) x
```

Key features of advanced generic programming

Strong elimination

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

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

$$
\begin{aligned}
&\mathsf{f} : (\mathsf{b} : \mathsf{Bool}) \to \mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathbb{N}\ \mathsf{else}\ \mathsf{Bool} \\
&\mathsf{f}\ \mathsf{true} = 0 \\
&\mathsf{f}\ \mathsf{false} = \mathsf{false}
\end{aligned}
$$

- Dynamic case analysis on data refines types

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

$$
\begin{aligned}
&\mathsf{f} : (\mathsf{b} : \mathsf{Bool}) \to \mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathbb{N}\ \mathsf{else}\ \mathsf{Bool} \\
&\mathsf{f}\ \mathsf{true} = 0 \\
&\mathsf{f}\ \mathsf{false} = \mathsf{false}
\end{aligned}
$$

- 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.

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

```
anything : ∀ {A} → A
anything = 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

```
anything : ∀ {A} → A
anything = 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:

Types are first-class data

They may be:

- Passed to functions, dependently or non-dependently

$$
\begin{aligned}
&\mathsf{f} \;:\; \mathsf{Set} \rightarrow \mathsf{Set} \\
&\mathsf{f}\,\mathsf{x} \;=\; (\mathsf{x} \rightarrow \mathsf{x}) \\
&\mathsf{g} \;:\; (\mathsf{A} \;:\; \mathsf{Set}) \rightarrow \mathsf{A} \rightarrow \mathsf{A} \\
&\mathsf{g}\,\mathsf{A}\,\mathsf{x} \;=\; \mathsf{x}
\end{aligned}
$$

Types are first-class data

They may be:

- Passed to functions, dependently or non-dependently

$$\begin{aligned} f &: \mathsf{Set} \to \mathsf{Set} \\ f\,x &= (x \to x) \\ g &: (A : \mathsf{Set}) \to A \to A \\ g\,A\,x &= x \end{aligned}$$

- Returned as results, dependently or non-dependently

$$\begin{aligned} h &: \mathsf{Bool} \to \exists\,(\lambda\,A \to A) \\ h\,x &= \mathsf{if}\ x\ \mathsf{then}\ (\mathbb{N},0)\ \mathsf{else}\ (\mathsf{Bool},\mathsf{true}) \end{aligned}$$

Types are first-class data

They may be:

- Passed to functions, dependently or non-dependently

$$
\begin{aligned}
&\mathsf{f} \ :\ \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{f}\,\mathsf{x} \ =\ (\mathsf{x} \to \mathsf{x}) \\
&\mathsf{g} \ :\ (\mathsf{A} \ :\ \mathsf{Set}) \to \mathsf{A} \to \mathsf{A} \\
&\mathsf{g}\,\mathsf{A}\,\mathsf{x} \ =\ \mathsf{x}
\end{aligned}
$$

- Returned as results, dependently or non-dependently

$$
\begin{aligned}
&\mathsf{h} \ :\ \mathsf{Bool} \to \exists\,(\lambda\ \mathsf{A} \to \mathsf{A}) \\
&\mathsf{h}\,\mathsf{x} \ =\ \mathsf{if}\ \mathsf{x}\ \mathsf{then}\ (\mathbb{N},0)\ \mathsf{else}\ (\mathsf{Bool},\mathsf{true})
\end{aligned}
$$

- Stored in data structures

$$
(\mathbb{N} :: \mathsf{Bool} :: \mathsf{Vec}\ \mathbb{N}\ 3 :: [])\ :\ \mathsf{List}\ \mathsf{Set}
$$

Type in Type

Both types and regular data may be inferred by the type checker (as implicit arguments) and symbolically evaluated at compile time.

Type in Type

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?

Type in Type

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.

Type in Type

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

Type in Type

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 $\Gamma \vdash t : \mathsf{Set}$ and t is a type.

Type in Type

What about Set? What is its type?

Type in Type

What about Set? What is its type?

- Flag `--type-in-type` enables $\Gamma \vdash \mathsf{Set} : \mathsf{Set}$

Type in Type

What about Set? What is its type?

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

```
head : ∀ {A n} → Vec A (suc n) → A
head (x :: xs) = x
```

Type in Type

What about Set? What is its type?

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

$$\begin{aligned}
&\mathsf{head} : \forall \{A\ n\} \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A \\
&\mathsf{head}\ (x :: xs) = x
\end{aligned}$$

$$\begin{aligned}
&x : \mathsf{Set} \\
&x = \mathsf{head}\ (\mathbb{N} :: \mathsf{Bool} :: [])
\end{aligned}$$

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.

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.

```
head : ∀ {A n} → Vec A (suc n) → A
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.

```
head : ∀ {A n} → Vec A (suc n) → A
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...

Coming next...

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