Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Advanced Functional Programming

2011-2012, period 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Jan 19, 2012

15. Dependently typed programming with Agda

15.1 Dependent functions

From functions to dependent functions

Normal functions

> $A \to B$

Domain (source) A, codomain (target) B. The target type B does not depend on the input value.

From functions to dependent functions

Normal functions

> $A \to B$

Domain (source) A, codomain (target) B. The target type B does not depend on the input value.

Dependent functions

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

Here, x is a name for the function argument, and B is a function from a term (x) to a type!

From functions to dependent functions

Normal functions

> $A \rightarrow B$

Domain (source) A, codomain (target) B. The target type B does not depend on the input value.

Dependent functions

> $(x : A) \rightarrow B\ x$

Here, x is a name for the function argument, and B is a function from a term (x) to a type!

Dependent types break down the barrier between terms and types.

Why?

- Can well-typed programs go wrong?
- error `"the impossible happened"`
- More precise specifications.
- Express properties about programs.

Why?

- Can well-typed programs go wrong?
- error `"the impossible happened"`
- More precise specifications.
- Express properties about programs.

Similar motivation as for **type-level programming** in Haskell. Haskell needs many extensions for this. Agda is conceptually simpler.

15.2 Agda

Agda

We are going to explore dependent types using **Agda**:

- An experimental dependently typed programming language.
- Actually Agda 2, the successor of Agda 1, a proof assistant.
- Developed at Chalmers University in Gothenburg, by Ulf Norell and others.
- Close to Haskell in many respects; also written in Haskell.
- Good enough to play with and run simple program, but not ready for production use.

Agda

We are going to explore dependent types using **Agda**:

- An experimental dependently typed programming language.
- Actually Agda 2, the successor of Agda 1, a proof assistant.
- Developed at Chalmers University in Gothenburg, by Ulf Norell and others.
- Close to Haskell in many respects; also written in Haskell.
- Good enough to play with and run simple program, but not ready for production use.

Notable features not directly related to dependent types:

- Quite flexible syntax.
- Interactive programming mode for Emacs.

Agda vs. others

There are other dependently typed languages, or systems that provide some form of dependent types:

- Cayenne, one of the first dependently typed programming languages, by Lennart Augustsson – no longer actively developed or maintained
- Coq, a well-known proof assistant that can be used as a programming language, developed by INRIA
- Epigram, a dependently typed system by Conor McBride – quite good ideas, but not very usable – a new version is in development
- Idris, an interesting new language by Edwin Brady, with a potentially good compiler and a relatively pragmatic approach
- ...

Agda vs. Haskell – quick overview

- No enforced naming conventions for identifiers.
- Unicode allowed and actively used.
- Use spaces to separate tokens.
- Type signatures for abstractions mandatory.
- No **case**, but with.
- Use : instead of :: for "is of type".
- Set replaces "kind" $*$.
- Polymorphism is type abstraction.
- Implicit arguments.
- No partial functions.
- More flexible module system.

No prelude

By default, Agda comes with absolutely nothing pre-loaded.

No prelude

By default, Agda comes with absolutely nothing pre-loaded.

Agda has essentially no built-in types except Set.

No prelude

By default, Agda comes with absolutely nothing pre-loaded.

Agda has essentially no built-in types except Set.

However, Agda offers syntactic sugar for a few types.

Modules

Every Agda module needs a header (cannot be omitted):

```
module Lecture where
```

15.3 Getting started

Datatypes

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

Datatypes

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

GADT syntax.

Convention to write types and type-variables with uppercase letters, constructors and functions with lowercase letters (different from Haskell).

Functions

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

Functions

$$
\begin{aligned}
&\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathsf{zero} \quad + \mathsf{n} = \mathsf{n} \\
&\mathsf{suc}\ \mathsf{m} + \mathsf{n} = \mathsf{suc}\ (\mathsf{m} + \mathsf{n})
\end{aligned}
$$

Type signatures are required.

Infix (and mix/distfix operators) can be defined by using underscores as placeholders.

There are **infix** statements for defining priorities like in Haskell.

Functions are defined via multiple lines as in Haskell, but there is no **case** statement.

Totality

Agda is (or tries to be) a **total** language. Functions terminate on every valid argument, and cannot fail:

- Pattern matching must be **exhaustive**. Non-exhaustive patterns are a compile-time error.
- Recursion must be **structural**.

Totality

Agda is (or tries to be) a **total** language. Functions terminate on every valid argument, and cannot fail:

- Pattern matching must be **exhaustive**. Non-exhaustive patterns are a compile-time error.
- Recursion must be **structural**.

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

Note that m is structurally smaller than suc m.

Lists

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

Lists

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

Double-colon and colon have reversed meaning compared to Haskell. (This is like they are used in ML and OCaml).

The type List has a parameter A of type Set, so

```
List : Set → Set
```

Functions on lists

We cannot define head and tail on lists – they are not total.

Functions on lists

We cannot define head and tail on lists – they are not total.

We can, however, define map:

$$
\begin{aligned}
&\mathsf{map} : (\mathsf{A} : \mathsf{Set}) \to (\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{B}) \to (\mathsf{List\ A} \to \mathsf{List\ B})\\
&\mathsf{map\ A\ B\ f}\ [\,] = [\,]\\
&\mathsf{map\ A\ B\ f}\ (\mathsf{x} :: \mathsf{xs}) = \mathsf{f\ x} :: \mathsf{map\ A\ B\ f\ xs}
\end{aligned}
$$

Functions on lists

We cannot define head and tail on lists – they are not total.

We can, however, define map:

$$\begin{aligned}
&\mathsf{map} : (\mathsf{A} : \mathsf{Set}) \to (\mathsf{B} : \mathsf{Set}) \to (\mathsf{A} \to \mathsf{B}) \to (\mathsf{List}\ \mathsf{A} \to \mathsf{List}\ \mathsf{B}) \\
&\mathsf{map}\ \mathsf{A}\ \mathsf{B}\ \mathsf{f}\ [\,] = [\,] \\
&\mathsf{map}\ \mathsf{A}\ \mathsf{B}\ \mathsf{f}\ (\mathsf{x} :: \mathsf{xs}) = \mathsf{f}\ \mathsf{x} :: \mathsf{map}\ \mathsf{A}\ \mathsf{B}\ \mathsf{f}\ \mathsf{xs}
\end{aligned}$$

Polymorphism is expressed by explicitly abstracting from types. Note that the type of map makes use of dependent functions!

Recursion is structural again.

Functions on lists

We cannot define head and tail on lists – they are not total.

We can, however, define map:

$$
\begin{aligned}
&\mathsf{map} : (\mathsf{A}\ \mathsf{B} : \mathsf{Set}) \rightarrow (\mathsf{A} \rightarrow \mathsf{B}) \rightarrow (\mathsf{List}\ \mathsf{A} \rightarrow \mathsf{List}\ \mathsf{B}) \\
&\mathsf{map}\ \mathsf{A}\ \mathsf{B}\ \mathsf{f}\ [\,] \qquad\quad = [\,] \\
&\mathsf{map}\ \mathsf{A}\ \mathsf{B}\ \mathsf{f}\ (\mathsf{x} :: \mathsf{xs}) = \mathsf{f}\ \mathsf{x} :: \mathsf{map}\ \mathsf{A}\ \mathsf{B}\ \mathsf{f}\ \mathsf{xs}
\end{aligned}
$$

Polymorphism is expressed by explicitly abstracting from types.
Note that the type of map makes use of dependent functions!

Recursion is structural again.

There is syntactic sugar to group arguments of the same type.

Functions on lists

We cannot define head and tail on lists – they are not total.

We can, however, define map:

```
map : {A B : Set} → (A → B) → (List A → List B)
map f []        = []
map f (x :: xs) = f x :: map f xs
```

Polymorphism is expressed by explicitly abstracting from types. Note that the type of map makes use of dependent functions!

Recursion is structural again.

There is syntactic sugar to group arguments of the same type.

Arguments that can be inferred from the context can be made **implicit**.

Folding lists

$$
\begin{aligned}
&\mathsf{foldr} : \{\mathsf{A}\ \mathsf{R} : \mathsf{Set}\} \to \mathsf{R} \to (\mathsf{A} \to \mathsf{R} \to \mathsf{R}) \to \mathsf{List}\ \mathsf{A} \to \mathsf{R} \\
&\mathsf{foldr}\ \mathsf{nil}\ \mathsf{cons}\ [\,] \qquad\quad = \mathsf{nil} \\
&\mathsf{foldr}\ \mathsf{nil}\ \mathsf{cons}\ (\mathsf{x} :: \mathsf{xs}) = \mathsf{cons}\ \mathsf{x}\ (\mathsf{foldr}\ \mathsf{nil}\ \mathsf{cons}\ \mathsf{xs})
\end{aligned}
$$

Once again, we have structural recursion.

Folding lists

$$
\begin{aligned}
&\text{foldr} : \{\text{A R} : \text{Set}\} \to \text{R} \to (\text{A} \to \text{R} \to \text{R}) \to \text{List A} \to \text{R}\\
&\text{foldr nil cons } [\,] \qquad\quad = \text{nil}\\
&\text{foldr nil cons } (\text{x} :: \text{xs}) = \text{cons x } (\text{foldr nil cons xs})
\end{aligned}
$$

Once again, we have structural recursion.

Functions defined using foldr are total (given the arguments to foldr are total).

Length of a list

$$\text{length} : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \mathbb{N}$$
$$\text{length} = \text{foldr zero } (\lambda\_\ n \rightarrow \text{suc zero} + n)$$

Lambda abstractions are as in Haskell.

Length of a list

```
length : {A : Set} → List A → ℕ
length = foldr zero (λ_ n → suc zero + n)
```

Lambda abstractions are as in Haskell.

We can enable some syntactic sugar for natural numbers

```
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

Length of a list

$$\text{length} : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \mathbb{N}$$
$$\text{length} = \text{foldr zero } (\lambda\_\ n \rightarrow \text{suc zero} + n)$$

Lambda abstractions are as in Haskell.

We can enable some syntactic sugar for natural numbers

```
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

$$\text{length} : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \mathbb{N}$$
$$\text{length} = \text{foldr } 0\ (\lambda\_\ n \rightarrow 1 + n)$$

Safe head and tail

More of the same:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A

safeHead : {A : Set} → List A → Maybe A
safeHead []        = nothing
safeHead (x :: xs) = just x

safeTail : {A : Set} → List A → Maybe (List A)
safeTail []        = nothing
safeTail (x :: xs) = just xs
```

15.4 Vectors

Vectors

Let us introduce proper dependent types:

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A 0
  _::_ : {n : ℕ} → A → Vec A n → Vec A (1 + n)
```

Vectors

Let us introduce proper dependent types:

$$\textbf{data}\ \mathsf{Vec}\ (\mathsf{A} : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \textbf{where}$$
$$[\,]\ : \mathsf{Vec}\ \mathsf{A}\ 0$$
$$\_{::}\_ : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (1 + \mathsf{n})$$

Agda distinguishes between **parameters** and **indices**:

- A is a parameter for the datatype and cannot change,
- the $\mathbb{N}$ is an index, and every constructor can target specific indices (like GADTs in Haskell can for types).

Vectors

Let us introduce proper dependent types:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\ (\mathsf{A} : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \textbf{where} \\
\quad [\,]\quad\ \ : \mathsf{Vec}\ \mathsf{A}\ 0 \\
\quad \_::\_ : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (1 + \mathsf{n})
\end{array}
$$

Agda distinguishes between **parameters** and **indices**:

- A is a parameter for the datatype and cannot change,
- the $\mathbb{N}$ is an index, and every constructor can target specific indices (like GADTs in Haskell can for types).

Agda allows us to overload constructors. We are reusing the list constructors.

Vectors

Let us introduce proper dependent types:

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A 0
  _::_ : {n : ℕ} → A → Vec A n → Vec A (1 + n)
```

Agda distinguishes between **parameters** and **indices**:

- A is a parameter for the datatype and cannot change,
- the $\mathbb{N}$ is an index, and every constructor can target specific indices (like GADTs in Haskell can for types).

Agda allows us to overload constructors. We are reusing the list constructors.

Note that $\mathbb{N}$ is not a kind, it's the **type** of natural numbers.

Vectors – contd.

$$\begin{array}{l} \textbf{data}\ \mathsf{Vec}\ (\mathsf{A} : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \textbf{where} \\ \quad [\,] \quad\ : \mathsf{Vec}\ \mathsf{A}\ 0 \\ \quad \_{::}\_ : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (1 + \mathsf{n}) \end{array}$$

Vectors – contd.

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A 0
  _::_ : {n : ℕ} → A → Vec A n → Vec A (1 + n)
```

Note that a term argument (the n) is implicit. The full type of _::_ is

```
_::_ : {A : Set} → {n : ℕ} → A → Vec A n → Vec A (1 + n)
```

Vectors – contd.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\ (\mathsf{A} : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \textbf{where} \\
\quad [\,] \quad : \mathsf{Vec}\ \mathsf{A}\ 0 \\
\quad \_{::}\_ : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (1 + \mathsf{n})
\end{array}
$$

Note that a term argument (the n) is implicit. The full type of _::_ is

$$
\_{::}\_ : \{\mathsf{A} : \mathsf{Set}\} \to \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (1 + \mathsf{n})
$$

Note that we use the **function** _+_ in the definition of Vec.

Vectors – contd.

$$\textbf{data}\ \mathsf{Vec}\ (\mathsf{A} : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \textbf{where}$$
$$[\,]\quad : \mathsf{Vec}\ \mathsf{A}\ 0$$
$$\_{::}\_ : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (1 + \mathsf{n})$$

Note that a term argument (the n) is implicit. The full type of _::_ is

$$\_{::}\_ : \{\mathsf{A} : \mathsf{Set}\} \to \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (1 + \mathsf{n})$$

Note that we use the **function** _+_ in the definition of Vec.

Recall that 1 is syntactic sugar for suc zero, so really the definition is

$$\textbf{data}\ \mathsf{Vec}\ (\mathsf{A} : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \textbf{where}$$
$$[\,]\quad : \mathsf{Vec}\ \mathsf{A}\ \mathsf{zero}$$
$$\_{::}\_ : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{A} \to \mathsf{Vec}\ \mathsf{A}\ \mathsf{n} \to \mathsf{Vec}\ \mathsf{A}\ (\mathsf{suc\ zero} + \mathsf{n})$$

Equality of types

Question

Are the two types

> Vec A (suc zero + n)
>
> Vec A (suc n)

the same?

Equality of types

Question

Are the two types

> Vec A (suc zero + n)
>
> Vec A (suc n)

the same?

Answer

Yes, because suc zero + n can be **symbolically reduced** to suc n by applying the **definition of _+_**. Agda considers types equal if and only if they (symbolically) reduce to the same term.

Equality of types – contd.

Followup question

Are the two types

> Vec A (n + suc zero)
>
> Vec A (suc n)

(note the difference to the situation before!) the same?

Equality of types – contd.

Followup question

Are the two types

> Vec A (n + suc zero)
>
> Vec A (suc n)

(note the difference to the situation before!) the same?

Answer

No, because _+_ is defined by pattern matching on the **first** argument. We do not know anything about n, so we cannot symbolically reduce n + suc zero. Agda cannot see that both types are equivalent, but we can help Agda by manually coercing the types (we will see that later).

Functions on vectors

Like in Haskell:

```
head : {A : Set} {n : ℕ} → Vec A (1 + n) → A
head (x :: xs) = x
tail : {A : Set} {n : ℕ} → Vec A (1 + n) → Vec A n
tail (x :: xs) = xs
map : {A B : Set} {n : ℕ} → (A → B) → Vec A n → Vec B n
map f []        = []
map f (x :: xs) = f x :: map f xs
```

Appending vectors

Easier than in Haskell – we just use $\_+\_$ again:

$$
\begin{aligned}
&\_{+\!\!+}\_ : \{A : \mathsf{Set}\}\ \{m\ n : \mathbb{N}\} \to \\
&\qquad \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (m + n) \\
&[\,]\qquad {+\!\!+}\ ys = ys \\
&(x :: xs)\ {+\!\!+}\ ys = x :: (xs\ {+\!\!+}\ ys)
\end{aligned}
$$

Appending vectors

Easier than in Haskell – we just use $\_+\_$ again:

$$
\begin{aligned}
&\_{+\!\!+}\_ : \{A : \mathsf{Set}\}\ \{m\ n : \mathbb{N}\} \to \\
&\qquad \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (m + n) \\
&[\,]\qquad\quad {+\!\!+}\ ys = ys \\
&(x :: xs)\ {+\!\!+}\ ys = x :: (xs\ {+\!\!+}\ ys)
\end{aligned}
$$

Verify that symbolic reduction is sufficient to typecheck this function!

Safe projection

Let us now try to write a total projection/indexing function for vectors.

Safe projection

Let us now try to write a total projection/indexing function for vectors.

Clearly,

> $\_!\_ : \{A : \mathsf{Set}\}\ \{n : \mathbb{N}\} \rightarrow \mathsf{Vec}\ A\ n \rightarrow \mathbb{N} \rightarrow A$

will not work.

Safe projection

Let us now try to write a total projection/indexing function for vectors.

Clearly,

$$\_!\_ : \{A : \mathsf{Set}\}\ \{n : \mathbb{N}\} \to \mathsf{Vec}\ A\ n \to \mathbb{N} \to A$$

will not work.

We need a type that represents natural numbers smaller than a certain bound.

Fin

The datatype Fin n contains the numbers from 0 to $n - 1$:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (1 + n)
  suc  : {n : ℕ} → Fin n → Fin (1 + n)
```

Fin

The datatype Fin n contains the numbers from 0 to $n - 1$:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (1 + n)
  suc  : {n : ℕ} → Fin n → Fin (1 + n)
```

Both constructors target Fin $(1 + n)$, so Fin 0 has no elements (as desired).

Fin

The datatype Fin n contains the numbers from 0 to $n - 1$:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (1 + n)
  suc  : {n : ℕ} → Fin n → Fin (1 + n)
```

Both constructors target Fin $(1 + n)$, so Fin 0 has no elements (as desired).

Fin 0	Fin 1	Fin 2	Fin 3	...
	zero	zero	zero	...
		suc zero	suc zero	...
			suc (suc zero)	...

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Safe projection – contd.

```
_!_ : {A : Set} {n : ℕ} → Vec A n → Fin n → A
[]          ! ()
(x :: xs) ! zero   = x
(x :: xs) ! suc n = xs ! n
```

Projecting from an empty list is impossible. We need the case, so that Agda can check for exhaustive patterns.

Safe projection – contd.

$$
\begin{array}{l}
\_!\_ : \{A : \mathsf{Set}\}\ \{n : \mathbb{N}\} \to \mathsf{Vec}\ A\ n \to \mathsf{Fin}\ n \to A \\
[\,] \qquad\quad !\ () \\
(x :: xs)\ !\ \mathsf{zero} \quad = x \\
(x :: xs)\ !\ \mathsf{suc}\ n = xs\ !\ n
\end{array}
$$

Projecting from an empty list is impossible. We need the case, so that Agda can check for exhaustive patterns.

However, there is no constructor of Fin 0 to use for the second argument, so we can use the **absurd pattern** $()$ without a right-hand side!

Safe projection – contd.

$$
\begin{array}{l}
\_!\_ : \{A : \mathsf{Set}\}\ \{n : \mathbb{N}\} \to \mathsf{Vec}\ A\ n \to \mathsf{Fin}\ n \to A \\
[\,]\qquad\quad !\ () \\
(x :: xs)\ !\ \mathsf{zero}\ \ = x \\
(x :: xs)\ !\ \mathsf{suc}\ n = xs\ !\ n
\end{array}
$$

Projecting from an empty list is impossible. We need the case, so that Agda can check for exhaustive patterns.

However, there is no constructor of Fin 0 to use for the second argument, so we can use the **absurd pattern** $()$ without a right-hand side!

Do not confuse absurd patterns with Haskell's unit type – these are two different concepts!

15.5 Equality

Agda's take on equality

$$\textbf{data}\ \_\equiv\_\ \{A : \mathsf{Set}\}\ (x : A) : A \to \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{refl} : x \equiv x$$

This is an equality between two terms of the same type.

Agda's take on equality

$$\textbf{data}\ \_\equiv\_\ \{A : \mathsf{Set}\}\ (x : A) : A \rightarrow \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{refl} : x \equiv x$$

This is an equality between two terms of the same type.

Much more versatile than Haskell's type-level equality.

Agda's take on equality

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

This is an equality between two terms of the same type.

Much more versatile than Haskell's type-level equality.

One of the A's is a parameter, one an index, because only the second one is restricted (to be equal to the first).

Using equality

Applying a function to equals results in equals:

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

Using equality

Applying a function to equals results in equals:

$$\begin{aligned}&\mathsf{cong} : \{A\ B : \mathsf{Set}\}\ \{x\ y : A\} \to\\&\qquad (f : A \to B) \to x \equiv y \to f\ x \equiv f\ y\\&\mathsf{cong}\ f\ \mathsf{refl} = \mathsf{refl}\end{aligned}$$

We can convert equals to equals in any context P:

$$\begin{aligned}&\mathsf{subst} : \{A : \mathsf{Set}\}\ \{x\ y : A\} \to\\&\qquad (P : A \to \mathsf{Set}) \to x \equiv y \to P\ x \to P\ y\\&\mathsf{subst}\ P\ \mathsf{refl}\ p = p\end{aligned}$$

Equality is symmetric and transitive

$$\mathsf{sym} : \{\mathsf{A} : \mathsf{Set}\}\ \{\mathsf{x}\ \mathsf{y} : \mathsf{A}\} \to \mathsf{x} \equiv \mathsf{y} \to \mathsf{y} \equiv \mathsf{x}$$
$$\mathsf{sym\ refl} = \mathsf{refl}$$

Equality is symmetric and transitive

$$\mathsf{sym} : \{\mathsf{A} : \mathsf{Set}\}\ \{\mathsf{x}\ \mathsf{y} : \mathsf{A}\} \to \mathsf{x} \equiv \mathsf{y} \to \mathsf{y} \equiv \mathsf{x}$$
$$\mathsf{sym}\ \mathsf{refl} = \mathsf{refl}$$

$$\mathsf{trans} : \{\mathsf{A} : \mathsf{Set}\}\ \{\mathsf{x}\ \mathsf{y}\ \mathsf{z} : \mathsf{A}\} \to$$
$$\mathsf{x} \equiv \mathsf{y} \to \mathsf{y} \equiv \mathsf{z} \to \mathsf{x} \equiv \mathsf{z}$$
$$\mathsf{trans}\ \mathsf{refl}\ \mathsf{refl} = \mathsf{refl}$$

Proving equalities

$$\mathsf{n{+}0{\equiv}n} : (\mathsf{n} : \mathbb{N}) \to \mathsf{n} + 0 \equiv \mathsf{n}$$
$$\mathsf{n{+}0{\equiv}n}\ 0 \qquad\qquad = \mathsf{refl}$$
$$\mathsf{n{+}0{\equiv}n}\ (\mathsf{suc}\ \mathsf{n}) = \textbf{?}$$

Proving equalities

$$\begin{aligned} &\text{n+0≡n} : (\text{n} : \mathbb{N}) \rightarrow \text{n} + 0 \equiv \text{n} \\ &\text{n+0≡n } 0 \qquad\quad = \text{refl} \\ &\text{n+0≡n } (\text{suc n}) = \textbf{?} \end{aligned}$$

The required type at the goal is

$$\text{suc n} + 0 \equiv \text{suc n}$$

which reduces to

$$\text{suc } (\text{n} + 0) \equiv \text{suc n}$$

Proving equalities

```
n+0≡n : (n : ℕ) → n + 0 ≡ n
n+0≡n 0        = refl
n+0≡n (suc n) = cong suc ?
```

The required type at the goal is

$$\mathsf{suc}\ \mathsf{n} + 0 \equiv \mathsf{suc}\ \mathsf{n}$$

which reduces to

$$\mathsf{suc}\ (\mathsf{n} + 0) \equiv \mathsf{suc}\ \mathsf{n}$$

After **refining** with cong suc, the goal type is

$$\mathsf{n} + 0 \equiv \mathsf{n}$$

Proving equalities

$$\mathsf{n{+}0{\equiv}n} : (\mathsf{n} : \mathbb{N}) \rightarrow \mathsf{n} + 0 \equiv \mathsf{n}$$
$$\mathsf{n{+}0{\equiv}n}\ 0 \qquad\qquad = \mathsf{refl}$$
$$\mathsf{n{+}0{\equiv}n}\ (\mathsf{suc\ n}) = \mathsf{cong\ suc}\ (\mathsf{n{+}0{\equiv}n\ n})$$

The required type at the goal is

$$\mathsf{suc\ n} + 0 \equiv \mathsf{suc\ n}$$

which reduces to

$$\mathsf{suc}\ (\mathsf{n} + 0) \equiv \mathsf{suc\ n}$$

After **refining** with cong suc, the goal type is

$$\mathsf{n} + 0 \equiv \mathsf{n}$$

Binary relations

Equality is an example of a binary relation:

$$\mathsf{Rel} : \mathsf{Set} \to \textbf{?}$$
$$\mathsf{Rel}\ \mathsf{A} = \mathsf{A} \to \mathsf{A} \to \mathsf{Set}$$

Binary relations

Equality is an example of a binary relation:

$$\mathsf{Rel} : \mathsf{Set} \to \textbf{?}$$
$$\mathsf{Rel}\ \mathsf{A} = \mathsf{A} \to \mathsf{A} \to \mathsf{Set}$$

What is the type of $\mathsf{A} \to \mathsf{A} \to \mathsf{Set}$?

Binary relations

Equality is an example of a binary relation:

$$\mathsf{Rel} : \mathsf{Set} \to \textbf{?}$$
$$\mathsf{Rel}\ \mathsf{A} = \mathsf{A} \to \mathsf{A} \to \mathsf{Set}$$

What is the type of $\mathsf{A} \to \mathsf{A} \to \mathsf{Set}$?

What is the type of Set? Not Set, but Set_1.

Binary relations

Equality is an example of a binary relation:

$$\mathsf{Rel} : \mathsf{Set} \to \mathsf{Set}_1$$
$$\mathsf{Rel}\ \mathsf{A} = \mathsf{A} \to \mathsf{A} \to \mathsf{Set}$$

What is the type of $\mathsf{A} \to \mathsf{A} \to \mathsf{Set}$?

What is the type of Set? Not Set, but Set_1.

And the type of Set_1 is Set_2 and so on.

Binary relations

Equality is an example of a binary relation:

$$\mathsf{Rel} : \mathsf{Set} \rightarrow \mathsf{Set}_1$$
$$\mathsf{Rel}\ \mathsf{A} = \mathsf{A} \rightarrow \mathsf{A} \rightarrow \mathsf{Set}$$

What is the type of $\mathsf{A} \rightarrow \mathsf{A} \rightarrow \mathsf{Set}$?

What is the type of Set? Not Set, but Set_1.

And the type of Set_1 is Set_2 and so on.

Note that Rel is like a type synonym in Haskell, without special syntax.

More abstractions

Properties like reflexivity, symmetry and transitivity are interesting for many relations, not just equality:

$$\mathsf{Reflexive} : \{\mathsf{A} : \mathsf{Set}\} \rightarrow \mathsf{Rel}\ \mathsf{A} \rightarrow \mathsf{Set}$$
$$\mathsf{Reflexive}\ \{\mathsf{A}\}\ \mathsf{R} = \{\mathsf{x} : \mathsf{A}\} \rightarrow \mathsf{R}\ \mathsf{x}\ \mathsf{x}$$

More abstractions

Properties like reflexivity, symmetry and transitivity are interesting for many relations, not just equality:

$$\mathsf{Reflexive} : \{\mathsf{A} : \mathsf{Set}\} \rightarrow \mathsf{Rel}\ \mathsf{A} \rightarrow \mathsf{Set}$$
$$\mathsf{Reflexive}\ \{\mathsf{A}\}\ \mathsf{R} = \{\mathsf{x} : \mathsf{A}\} \rightarrow \mathsf{R}\ \mathsf{x}\ \mathsf{x}$$

Note that we are matching on an implicit argument!

More abstractions

Properties like reflexivity, symmetry and transitivity are interesting for many relations, not just equality:

```
Reflexive : {A : Set} → Rel A → Set
Reflexive {A} R = {x : A} → R x x
```

Note that we are matching on an implicit argument!

```
Symmetric : {A : Set} → Rel A → Set
Symmetric {A} R = {x y : A} → R x y → R y x
Transitive : {A : Set} → Rel A → Set
Transitive {A} R =
  {x y z : A} → R x y → R y z → R x z
```

Using the abstractions

The type synonyms can for instance be used in the type signatures of sym and trans:

$$\mathsf{sym} : \{\mathsf{A} : \mathsf{Set}\} \rightarrow \mathsf{Symmetric}\ \{\mathsf{A}\}\ (\_\equiv\_\ \{\mathsf{A}\})$$
$$\mathsf{sym}\ \mathsf{refl} = \mathsf{refl}$$

$$\mathsf{trans} : \{\mathsf{A} : \mathsf{Set}\} \rightarrow \mathsf{Transitive}\ \{\mathsf{A}\}\ (\_\equiv\_\ \{\mathsf{A}\})$$
$$\mathsf{trans}\ \mathsf{refl}\ \mathsf{refl} = \mathsf{refl}$$

Using the abstractions

The type synonyms can for instance be used in the type signatures of sym and trans:

$$\text{sym} : \{\text{A} : \text{Set}\} \rightarrow \text{Symmetric}\ \{\text{A}\}\ (\_\equiv\_\ \{\text{A}\})$$
$$\text{sym refl} = \text{refl}$$

$$\text{trans} : \{\text{A} : \text{Set}\} \rightarrow \text{Transitive}\ \{\text{A}\}\ (\_\equiv\_\ \{\text{A}\})$$
$$\text{trans refl refl} = \text{refl}$$

Both the synonym and the relation are polymorphic – we need to fill in the type argument explicitly to make sure that they are unified.

Observations

- Term and type level are mixed.
- No duplication of concepts: in particular, type-level abstraction and application is the same as value-level abstraction and application.
- Dependent functions subsume polymorphism.
- Implicit arguments help to keep the programs concise, and are relatively orthogonal to the rest (unlike type classes in Haskell).
- Types become like theorems, and programs like proofs (Curry-Howard isomorphism).
- Interactive development becomes really helpful, certainly once we start writing proofs.

15.6 Induction

Associativity of addition

```
+-assoc : (m n o : ℕ) → (m + n) + o ≡ m + (n + o)
+-assoc zero    n o = refl
+-assoc (suc m) n o = cong suc (+-assoc m n o)
```

Associativity of addition

$$
\begin{aligned}
&+\text{-assoc} : (m\ n\ o : \mathbb{N}) \to (m + n) + o \equiv m + (n + o) \\
&+\text{-assoc zero} \quad\ \ n\ o = \text{refl} \\
&+\text{-assoc (suc m) n o} = \text{cong suc } (+\text{-assoc m n o})
\end{aligned}
$$

Is this a "fold" on natural numbers?

Associativity of addition

$$
\begin{aligned}
&+\text{-assoc} : (m\ n\ o : \mathbb{N}) \rightarrow (m + n) + o \equiv m + (n + o) \\
&+\text{-assoc zero}\quad n\ o = \text{refl} \\
&+\text{-assoc (suc m) } n\ o = \text{cong suc } (+\text{-assoc m n o})
\end{aligned}
$$

Is this a “fold” on natural numbers?

Not quite, because the result type of the recursive calls is different from the result type of the original call.

Fold on natural numbers

Recall the fold on natural numbers:

$$
\begin{aligned}
&\mathbb{N}\text{-Fold} : \{\,\mathsf{P} : \mathsf{Set}\,\} \rightarrow \\
&\qquad\qquad \mathsf{P} \rightarrow (\mathsf{P} \rightarrow \mathsf{P}) \rightarrow \\
&\qquad\qquad \mathbb{N} \rightarrow \mathsf{P} \\
&\mathbb{N}\text{-Fold pz ps zero} \qquad\; = \text{pz} \\
&\mathbb{N}\text{-Fold pz ps (suc n)} = \text{ps } (\mathbb{N}\text{-Fold pz ps n})
\end{aligned}
$$

Fold on natural numbers

Recall the fold on natural numbers:

$$
\begin{aligned}
&\mathbb{N}\text{-Fold} : \{\mathsf{P} : \mathsf{Set}\} \to \\
&\qquad\qquad \mathsf{P} \to (\mathsf{P} \to \mathsf{P}) \to \\
&\qquad\qquad \mathbb{N} \to \mathsf{P} \\
&\mathbb{N}\text{-Fold}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{zero} \quad\quad = \mathsf{pz} \\
&\mathbb{N}\text{-Fold}\ \mathsf{pz}\ \mathsf{ps}\ (\mathsf{suc}\ \mathsf{n}) = \mathsf{ps}\ (\mathbb{N}\text{-Fold}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{n})
\end{aligned}
$$

The result type P is constant, but in the case of associativity (and other properties), it cannot be.

Induction on natural numbers

We generalize $\mathbb{N}$-Fold to $\mathbb{N}$-Ind:

$$
\begin{aligned}
&\mathbb{N}\text{-Ind} : (\mathsf{P} : \mathbb{N} \to \mathsf{Set}) \to \\
&\qquad\quad \mathsf{P}\ 0 \to (\{\mathsf{n} : \mathbb{N}\} \to \mathsf{P}\ \mathsf{n} \to \mathsf{P}\ (\mathsf{suc}\ \mathsf{n})) \to \\
&\qquad\quad (\mathsf{n} : \mathbb{N}) \to \mathsf{P}\ \mathsf{n} \\
&\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{zero} \qquad\ = \mathsf{pz} \\
&\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ (\mathsf{suc}\ \mathsf{n}) = \mathsf{ps}\ (\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{n})
\end{aligned}
$$

Induction on natural numbers

We generalize $\mathbb{N}$-Fold to $\mathbb{N}$-Ind:

$$
\begin{aligned}
&\mathbb{N}\text{-Ind} : (\mathsf{P} : \mathbb{N} \to \mathsf{Set}) \to \\
&\qquad \mathsf{P}\ 0 \to (\{\mathsf{n} : \mathbb{N}\} \to \mathsf{P}\ \mathsf{n} \to \mathsf{P}\ (\mathsf{suc}\ \mathsf{n})) \to \\
&\qquad (\mathsf{n} : \mathbb{N}) \to \mathsf{P}\ \mathsf{n} \\
&\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{zero} \quad\ \ = \mathsf{pz} \\
&\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ (\mathsf{suc}\ \mathsf{n}) = \mathsf{ps}\ (\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{n})
\end{aligned}
$$

$$
\mathbb{N}\text{-Fold}\ \{\mathsf{P}\}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{n} = \mathbb{N}\text{-Ind}\ (\lambda \mathsf{x} \to \mathsf{P})\ \mathsf{pz}\ (\lambda\{\mathsf{n}\} \to \mathsf{ps})\ \mathsf{n}
$$

Induction on natural numbers

We generalize $\mathbb{N}$-Fold to $\mathbb{N}$-Ind:

$$
\begin{aligned}
&\mathbb{N}\text{-Ind} : (\mathsf{P} : \mathbb{N} \to \mathsf{Set}) \to \\
&\qquad \mathsf{P}\ 0 \to (\{\mathsf{n} : \mathbb{N}\} \to \mathsf{P}\ \mathsf{n} \to \mathsf{P}\ (\mathsf{suc}\ \mathsf{n})) \to \\
&\qquad (\mathsf{n} : \mathbb{N}) \to \mathsf{P}\ \mathsf{n} \\
&\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{zero} \quad\ \ = \mathsf{pz} \\
&\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ (\mathsf{suc}\ \mathsf{n}) = \mathsf{ps}\ (\mathbb{N}\text{-Ind}\ \mathsf{P}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{n})
\end{aligned}
$$

$$
\mathbb{N}\text{-Fold}\ \{\mathsf{P}\}\ \mathsf{pz}\ \mathsf{ps}\ \mathsf{n} = \mathbb{N}\text{-Ind}\ (\lambda \mathsf{x} \to \mathsf{P})\ \mathsf{pz}\ (\lambda\{\mathsf{n}\} \to \mathsf{ps})\ \mathsf{n}
$$

Note that $\mathbb{N}$-Ind corresponds to the proof principle we know as **induction** on natural numbers. The implementation is just as for the fold.

Using induction on natural numbers

```
n+0≡n : (n : ℕ) → n + 0 ≡ n
n+0≡n 0         = refl
n+0≡n (suc n) = cong suc n+0≡n
```

Using induction on natural numbers

```
n+0≡n : (n : ℕ) → n + 0 ≡ n
n+0≡n 0       = refl
n+0≡n (suc n) = cong suc n+0≡n
```

```
n+0≡n = ℕ-Ind (λn → n + 0 ≡ n)
              refl
              (λr → cong suc r)
```

Using induction on natural numbers – contd.

```
+-assoc : (m n o : ℕ) → (m + n) + o ≡ m + (n + o)
+-assoc zero     n o = refl
+-assoc (suc m) n o = cong suc (+-assoc m n o)
```

Using induction on natural numbers – contd.

$$
\begin{aligned}
&+\text{-assoc} : (m\ n\ o : \mathbb{N}) \to (m + n) + o \equiv m + (n + o)\\
&+\text{-assoc zero} \quad\quad n\ o = \text{refl}\\
&+\text{-assoc (suc m) n o} = \text{cong suc } (+\text{-assoc m n o})
\end{aligned}
$$

$$
\begin{aligned}
&+\text{-assoc} = \mathbb{N}\text{-Ind}\\
&\qquad (\lambda m \to (n\ o : \mathbb{N}) \to (m + n) + o \equiv m + (n + o))\\
&\qquad (\lambda n\ o \to \text{refl})\\
&\qquad (\lambda r\ n\ o \to \text{cong suc } (r\ n\ o))
\end{aligned}
$$

Induction on vectors

For vectors, we obtain a similar generalization from fold to induction principle:

$$
\begin{aligned}
&\_{+\!\!+}\_ : \{A : \mathsf{Set}\}\ \{m\ n : \mathbb{N}\} \rightarrow \\
&\qquad\quad \mathsf{Vec}\ A\ m \rightarrow \mathsf{Vec}\ A\ n \rightarrow \mathsf{Vec}\ A\ (m + n) \\
&[\,]\qquad\ \ {+\!\!+}\ ys = ys \\
&(x :: xs)\ {+\!\!+}\ ys = x :: (xs\ {+\!\!+}\ ys)
\end{aligned}
$$

Here, the result type of recursive calls is also dependent on the length of the vector.

Induction on vectors – contd.

$$
\begin{array}{l}
\text{Vec-Ind} : \\
\quad \{A : \text{Set}\} \rightarrow \\
\quad (P : \mathbb{N} \rightarrow \text{Set}) \rightarrow \\
\quad P\ [\,] \rightarrow \\
\quad (\{n : \mathbb{N}\}\ \{xs : \text{Vec A n}\}\ (x : A) \rightarrow P\ xs \rightarrow P\ (x :: xs)) \rightarrow \\
\quad \{n : \mathbb{N}\} \rightarrow (xs : \text{Vec A n}) \rightarrow P\ xs \\
\text{Vec-Ind P pn pc}\ [\,] \qquad\quad = \text{pn} \\
\text{Vec-Ind P pn pc}\ (x :: xs) = \text{pc x (Vec-Ind P pn pc xs)}
\end{array}
$$

Induction on vectors – contd.

$$
\begin{aligned}
&\text{Vec-Ind} : \\
&\quad \{A : \text{Set}\} \rightarrow \\
&\quad (P : \mathbb{N} \rightarrow \text{Set}) \rightarrow \\
&\quad P\ [\,] \rightarrow \\
&\quad (\{n : \mathbb{N}\}\ \{xs : \text{Vec}\ A\ n\}\ (x : A) \rightarrow P\ xs \rightarrow P\ (x :: xs)) \rightarrow \\
&\quad \{n : \mathbb{N}\} \rightarrow (xs : \text{Vec}\ A\ n) \rightarrow P\ xs \\
&\text{Vec-Ind}\ P\ pn\ pc\ [\,] \qquad\quad = pn \\
&\text{Vec-Ind}\ P\ pn\ pc\ (x :: xs) = pc\ x\ (\text{Vec-Ind}\ P\ pn\ pc\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&xs +\!\!+\ ys = \\
&\quad \text{Vec-Ind}\ (\lambda m \rightarrow \{n : \mathbb{N}\} \rightarrow \text{Vec}\ A\ n \rightarrow \text{Vec}\ A\ (m + n)) \\
&\qquad\qquad\quad (\lambda ys \rightarrow ys) \\
&\qquad\qquad\quad (\lambda x\ r\ ys \rightarrow x :: r\ ys)
\end{aligned}
$$

15.7 Curry-Howard

Curry-Howard isomorphism

Correspondence between propositions and types, and (constructive) proofs and programs.

Pairs

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

Dependent pairs

$$\textbf{data}\ \Sigma\ (A : Set)\ (B : A \to Set) : Set\ \textbf{where}$$
$$\quad \_,\_ : (x : A) \to B\ x \to \Sigma\ A\ B$$

The second component of the pair can depend on the value of the first.

Dependent pairs

$$\textbf{data}\ \Sigma\ (A : Set)\ (B : A \to Set) : Set\ \textbf{where}$$
$$\quad \_,\_ : (x : A) \to B\ x \to \Sigma\ A\ B$$

The second component of the pair can depend on the value of the first.

$$\text{List}\ A = \Sigma\ \mathbb{N}\ (\text{Vec}\ A)$$

Dependent pairs

$$\textbf{data}\ \Sigma\ (A : Set)\ (B : A \to Set) : Set\ \textbf{where}$$
$$\quad \_,\_ : (x : A) \to B\ x \to \Sigma\ A\ B$$

The second component of the pair can depend on the value of the first.

$$List\ A = \Sigma\ \mathbb{N}\ (Vec\ A)$$

$$A \times B = \Sigma\ A\ (const\ B)$$

15.8 Universes

Computing types

Agda's unit type:

```
data ⊤ : Set where
  tt : ⊤
```

Computing types

Agda's unit type:

$$\textbf{data}\ \top : \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{tt} : \top$$

Yet another way to define vectors:

$$\begin{aligned}
&\mathsf{Vec} : \mathsf{A} \to \mathbb{N} \to \mathsf{Set} \\
&\mathsf{Vec\ A\ zero} \quad\ \ = \mathsf{tt} \\
&\mathsf{Vec\ A\ (suc\ n)} = \mathsf{A}\ \times\ \mathsf{Vec\ A\ n}
\end{aligned}$$

Universe

A type of **codes** C together with an **interpretation** function $\mathsf{el} : \mathsf{C} \to \mathsf{Set}$ is called a universe.

Universe

A type of **codes** C together with an **interpretation** function $\mathsf{el} : \mathsf{C} \rightarrow \mathsf{Set}$ is called a universe.

The type $\mathbb{N}$ and the function Vec above are a simple example.

Reflecting types

```
data Code : Set where
  unit : Code
  bool : Code
  nat  : Code
  pair : Code → Code → Code

⟦_⟧ : Code → Set where
⟦unit⟧     = ⊤
⟦bool⟧     = Bool
⟦nat⟧      = ℕ
⟦pair x y⟧ = ⟦x⟧ × ⟦y⟧
```

Overloaded functions

$$
\begin{array}{lllll}
\mathsf{eq} : (\mathsf{c} : \mathsf{Code}) \to [\![\mathsf{c}]\!] \to [\![\mathsf{c}]\!] \to \mathsf{Bool} \\
\mathsf{eq\ unit} & \mathsf{tt} & \mathsf{tt} & = \mathsf{true} \\
\mathsf{eq\ bool} & \mathsf{true} & \mathsf{true} & = \mathsf{true} \\
\mathsf{eq\ bool} & \mathsf{false} & \mathsf{false} & = \mathsf{true} \\
\mathsf{eq\ nat} & \mathsf{zero} & \mathsf{zero} & = \mathsf{true} \\
\mathsf{eq\ nat} & (\mathsf{suc\ m}) & (\mathsf{suc\ n}) & = \mathsf{eq\ nat\ m\ n} \\
\mathsf{eq\ (pair\ x\ y)} & (\mathsf{a}, \mathsf{b}) & (\mathsf{c}, \mathsf{d}) & = \mathsf{eq\ x\ a\ c} \wedge \mathsf{eq\ y\ b\ d} \\
\mathsf{eq}\ \_ & \_ & \_ & = \mathsf{false}
\end{array}
$$

Several applications

For example:

- Computing the arguments of a function from a format string (printf).
- Computing the type, i.e., dimensions and color depth of an image from the image header.
- Computing the types of database entries from a database schema.
- . . .

The latter two are not possible in Haskell. Even the type-level programming trick does not work, because the input value is not statically known.

Datatype-generic programming

Idea: most datatypes are built from a limited number of concepts.

If we can express datatypes using such a limited number of concepts, we can write **data-type generic** functions and datatypes.

Datatype-generic programming

Idea: most datatypes are built from a limited number of concepts.

If we can express datatypes using such a limited number of concepts, we can write **data-type generic** functions and datatypes.

Examples:

- Haskell's derived classes
- Generic map, fold, unfold
- Traversals and queries
- tries and zippers
- ...

Conclusions