AGDA \\ SCALATAM

ERNESTO COPELLO (ECOPELLO@GMAIL.COM)

Created: 2019-05-06 Mon 20:30

1 AGDA

A Functional Dependently Typed Programming Language

Similar languages: [Coq], [Idris], [Lean]

1.1 WHAT IS AGDA GOOD FOR?

Utilize the capacity of the computers in a reliable way

- Joining programming and mathematics
- Formal definitions, theorems, proofs, and algorithms

1.2 ELIMINATION OF ERRORS

- No runtime errors
- Properties can be formalized and proved
- Automatic proof checking
- If compiles it works (no testing)

1.3 SAFE OPTIMIZATION

- Runtime checks like array bounds checks are eliminated
- Defensive coding is unnecessary

1.4 HIGH-LEVEL PROGRAMMING

- Formal specification can be given with the help of exact mathematical concepts like:
  - groups, rings, lattices, categories, and so on
- Programming with types as data (generic programming, universes)
- Embedded domain-specific languages can be defined with arbitrary precision

2 LANGUAGE INTRODUCTION

2.1 `Bool` SIMPLE ENUMERATION

```
\begin{code}
data Bool : Set where
  ff : Bool
  tt : Bool
\end{code}
```

$$
\begin{array}{cccccc}
 & & \mathsf{ff} & \mathsf{tt} & \\
 & & \uparrow & \uparrow & \\
Bool & = \{ & false, & true & \}
\end{array}
$$

2.2 FUNCTIONS (CODE)

- Unicode & Mixfix operators
- Pattern-matching with coverage checking
- Termination checking

```
\begin{code}
¬_ : Bool → Bool
¬ ff = tt
¬ tt = ff
\end{code}
```

2.3 PARAMETRIC POLYMORPHISM

```
\begin{code}
_if_then_else′_ : (A : Set) → Bool → A → A → A
A if tt then a else′ _ = a
A if ff then _ else′ b = b
\end{code}
```

2.3 PARAMETRIC POLYMORPHISM

```
\begin{code}
_if_then_else′_ : (A : Set) → Bool → A → A → A
A if tt then a else′ _ = a
A if ff then _ else′ b = b
\end{code}
```

{Implicit arguments} (derive arguments from context)

2.3 PARAMETRIC POLYMORPHISM

```
\begin{code}
_if_then_else′_ : (A : Set) → Bool → A → A → A
A if tt then a else′ _ = a
A if ff then _ else′ b = b
\end{code}
```

{Implicit arguments} (derive arguments from context)

```
\begin{code}
if_then_else_ : {A : Set} → Bool → A → A → A
if tt then a else _ = a
if ff then _ else b = b
\end{code}
```

2.4 ℕ INDUCTIVE TYPES

```
\begin{code}
data ℕ : Set where
  z : ℕ
  s : ℕ → ℕ
\end{code}
```

$$\begin{array}{ccccccc} & & \texttt{z} & \texttt{s z} & \texttt{s (s z)} & \\ \mathbb{N} & = \{ & 0, & 1, & 2, & \ldots\} \end{array}$$

2.5 SUM

```
\begin{code}
_+_ : ℕ → ℕ → ℕ
z     + m = m
(s n) + m = s (n + m)
\end{code}
```

2.6 `List` DATA TYPE POLYMORPHISM

```
\begin{code}
infixr 5 _∷_
data List (A : Set) : Set where
  []  : List A
  _∷_ : A → List A → List A
\end{code}
```

$$\begin{array}{lllll} & [] & [\mathsf{tt}] & [\mathsf{ff}, \mathsf{tt}] & \\ List\ Bool = \{ & [], & tt :: [], & ff :: tt :: [], & \ldots \} \\ & [] & [\mathsf{z}] & [\mathsf{s}\ \mathsf{z}, \mathsf{z}] & \\ List\ \mathbb{N} = \{ & [], & z :: [], & s\ z :: z :: [], & \ldots \} \end{array}$$

2.7 ANONYMOUS LAMBDA ABSTRACTIONS

```
\begin{code}
identity : {A : Set} → A → A
identity = λ x → x
\end{code}
```

```
\begin{code}
identity′ : {A : Set} → A → A
identity′ x = x
\end{code}
```

- Unique possible function for this type

"Abstraction Theorem" by J.Reynolds

"Theorems for free!" by P.Walder

3 ELEMENT INDEXED DATA TYPES

Dependently (*elements*) Typed Programming Language

3.1 `Fin n` IS A TYPE WITH N ELEMENTS.

```
\begin{code}
data Fin : ℕ → Set where
  zz : {n : ℕ} → Fin (s n)
  ss : {n : ℕ} → Fin n → Fin (s n)
\end{code}
```

$$\begin{aligned} Fin\, z &= \{ \quad \} \\ Fin\,(s\ z) &= \{ \quad zz \quad \} \\ Fin\,(s\,(s\ z)) &= \{ \quad zz, \quad ss\ zz \quad \} \end{aligned}$$

3.2 `Vec A n` IS A TYPE OF VECTORS WITH `n` ELEMENTS OF TYPE `A`.

```
\begin{code}
data Vec (A : Set) : ℕ → Set where
  []  : Vec A z
  _∷_ : {n : ℕ} → A → Vec A n → Vec A (s n)
\end{code}
```

$$\forall A, Vec\, A\, z \quad = \{ \qquad \}$$

$$Vec\, Bool\, (s\; z) \quad = \{ \quad [ff], \quad [tt] \quad \}$$

3.3 HEAD

```
\begin{code}
head : {A : Set}{n : ℕ} -> Vec A (s n) -> A
head (x ∷ xs) = x
\end{code}
```

3.3 HEAD

```
\begin{code}
head : {A : Set}{n : ℕ} -> Vec A (s n) -> A
head (x ∷ xs) = x
\end{code}
```

- No empty case, the type specification excludes it

3.3 HEAD

```
\begin{code}
head : {A : Set}{n : ℕ} -> Vec A (s n) -> A
head (x ∷ xs) = x
\end{code}
```

- No empty case, the type specification excludes it
- Same defintion in Haskell, but its use is restricted!

3.3 HEAD

```
\begin{code}
head : {A : Set}{n : ℕ} -> Vec A (s n) -> A
head (x ∷ xs) = x
\end{code}
```

- No empty case, the type specification excludes it
- Same defintion in Haskell, but its use is restricted!

```
\begin{code}
use : {A : Set}{n : ℕ} -> Vec A n -> A
use vector = head vector
\end{code}
```

3.3 HEAD

```
\begin{code}
head : {A : Set}{n : ℕ} -> Vec A (s n) -> A
head (x ∷ xs) = x
\end{code}
```

- No empty case, the type specification excludes it
- Same defintion in Haskell, but its use is restricted!

```
\begin{code}
use : {A : Set}{n : ℕ} -> Vec A n -> A
use vector = head vector
\end{code}
```

```
Typing error!
.n != (s (_n_50 vector)) of type ℕ
when checking that the expression vector has type
Vec .A (s (_n_50 vector))
```

3.4 n^{th} ELEMENT OF A

```
\begin{code}
_!_ : {n : ℕ} {A : Set} → Vec A n → Fin n → A
[]        ! ()  -- n=0 is an impossible case, as Fin z is empty
(x ∷ xs) ! zz   = x
(x ∷ xs) ! ss n = xs ! n
\end{code}
```

- No bounds checks required in runtime

3.5 VECTORS CONCATENATION

```
\begin{code}
_++_ : {m n : ℕ} {A : Set} → Vec A m → Vec A n → Vec A (m + n)
[]        ++ ys = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)
\end{code}
```

- Result size is correct by definition
- No new elements can be added (by Parameticity)

4 CURRY-HOWARD ISOMORPHISM ≈ PROPOSITIONS-AS-TYPES

Propositions can be codified as **Sets**

A proof of a proposition is an element of the **Set** which codifies it

4.1 ABSURDITY $\bot$ AND TRUTH $\top$

```
\begin{code}
data ⊥ : Set where
\end{code}
```

$$\bot = \{\ \}$$

```
\begin{code}
data ⊤′ : Set where
  ⊤' : ⊤′
\end{code}
```

$$\top' = \{\top'\}$$

4.2 IMPLICATION $\Rightarrow$

Introduction

```
\begin{code}
_⇒_ : (A : Set) → (B : Set) → Set
A ⇒ B = A → B
\end{code}
```

$$\dfrac{\begin{matrix} A \\ \vdots \\ B \end{matrix}}{A \Rightarrow B}$$

Elimination

```
\begin{code}
modus-ponens : {A B : Set} → A → (A → B) → B
modus-ponens a f = f a
\end{code}
```

$$\dfrac{A \qquad A \Rightarrow B}{B}$$

4.3 CONJUNCTION $\wedge$

Introduction

```
\begin{code}
data _∧_ (A B : Set) : Set where
  ⟨_,_⟩ : A → B → A ∧ B
\end{code}
```

$$\frac{A \qquad\qquad B}{A \wedge B}$$

Elimination

```
\begin{code}
fst : {A B : Set} → A ∧ B → A
fst ⟨ a , _ ⟩ = a

snd : {A B : Set} → A ∧ B → B
snd ⟨ _ , b ⟩ = b
\end{code}
```

$$\frac{A \wedge B}{A}$$

$$\frac{A \wedge B}{B}$$

4.4 DISJUNCTION $\vee$

Introduction

```
\begin{code}
data _∨_ (A B : Set) : Set where
  left  : A → A ∨ B
  right : B → A ∨ B
\end{code}
```

$$\frac{A}{A \vee B}$$

$$\frac{B}{A \vee B}$$

Elimination

```
\begin{code}
case : {A B C : Set}
     → A ∨ B → (A → C) → (B → C)
     → C
case (left a)  f _ = f a
case (right b) _ g = g b
\end{code}
```

$$\dfrac{A \vee B \qquad \begin{array}{c} A \\ \vdots \\ \dfrac{C}{C} \end{array} \qquad \begin{array}{c} B \\ \vdots \\ \dfrac{C}{C} \end{array}}{C}$$

5.5

4.5 LOGICAL QUANTIFIERS ∀, ∃

```
\begin{code}
∀′ : (A : Set) → (P : A → Set) → Set
∀′ A P = (a : A) → P a
\end{code}
```

$$\dfrac{\dfrac{\begin{matrix} a \text{ not free} \\ \vdots \end{matrix}}{P\,a}}{\forall a, P\,a} \qquad \dfrac{\forall a, P\,a}{P\,a}$$

```
\begin{code}
data ∃′ {A : Set}(P : A → Set) : Set where
  exists : (a : A) → P a → ∃′ P

witness : {A : Set}{P : A → Set} → ∃′ P → A
witness (exists a _) = a
\end{code}
```

$$\dfrac{\begin{matrix} \vdots \\ P\,a \end{matrix}}{\exists a, P\,a}$$

4.6 CLASSIC LOGIC EXAMPLE

We can **program** $\equiv$ **prove** the highschool exercise

```
\begin{code}
data Man : Set where
  person : ℕ → Man

socratesIsMortal :  {socrates : Man}
                    {isMortal : Man → Set}
                 →  ∀′ Man isMortal
                 →  isMortal socrates
socratesIsMortal {socrates} all = all socrates
\end{code}
```

4.7 BASIC PROOF

Sum definition

```
\begin{code}
open import Relation.Binary.PropositionalEquality as PE
open PE.≡-Reasoning

-- trivial by evaluation rules
proof1 : (n : ℕ) → z + n ≡ n
proof1 n = refl

-- inductive proof
proof2 : (n : ℕ) → n + z ≡ n
proof2 z     = refl
proof2 (s n) =   s n + z
               ≡⟨ refl ⟩                     -- trivial by ev. rule
                 s (n + z)
               ≡⟨ cong s (proof2 n) ⟩        -- cong : (f : A → B) → x ≡
                 s n                         --             → f x ≡ f y
             ∎
\end{code}
```

5 BACKENDS

- Javascript
- Haskell
- Epic Compiler
- Ruby