Agda – A brief overview

Eef Uijen
Alwyn Stiles

Index

- Introduction to Agda
- Agda as a programming language
- Agda as a proof checker

Introduction to Agda

What is Agda?

- Functional programming language
  - Dependent types
  - Strong typing
- Proof assistant

Agda vs Coq

- Both are based on intuitionistic type theory, and satisfy the Curry-Howard correspondence
- Coq is impredicative universe Prop, and difference between Prop/Set, Agda has Set = Set_0, Set_1,Set_2,... Russell style $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \cdots$
- Coq has tactical theorem proving, Agda does not

Agda vs Haskell

- Who knows Haskell?

Agda vs Haskell

- Who knows Haskell?
- Agda ≈ Haskell + Harmonious Support for Dependent Types[1]
  - Syntax inspired by Haskell

[1] = https://github.com/alhassy/AgdaCheatSheet

Agda vs Haskell

- Who knows Haskell?
- Agda ≈ Haskell + Harmonious Support for Dependent Types[1]
  - Syntax inspired by Haskell
- Curry-Howard correspondence does not hold for Haskell
  - Partial general recursive functions
  - Non-strictly positive data types

$\forall (a : \mathrm{Type}) \to a$

```
faulty :: a
faulty = let x = x in x
```

```
data Empty
data EmptyFun = Fun (EmptyFun->Empty)
selfApp :: EmptyFun -> Empty
selfApp (Fun f) = f (Fun f)
faulty :: Empty
faulty = selfApp (Fun selfApp)
```

[1] = https://github.com/alhassy/AgdaCheatSheet

Agda as a Programming Language

Programming in Agda

- “Agda is primarily being developed as a programming language and not as a proof assistant.”[2]
- Compiler backends
  - GHC
  - UHC
  - Javascript

[2] = Ana Bove, Peter Dybjer, Ulf Norell, A Brief Overview of Agda – A Functional Language with Dependent Types, 2009.

Demo

- Hello World
- Pattern Matching
- Basic Data Types
- Mis Fix operators
- Dependent Data Types
- Example I/O Program

Is Agda Turing complete?

No

Agda is not Turing complete

- Agda is a total functional programming language
  - No partial functions
  - Programs always terminate
- Turing completeness
  - Program might not terminate

Is Agda Turing complete?

No, but yes

(Kind of)

Agda is total(ish)

- “Agda and other languages based on type theory are total languages in the sense that a program e of type T will always terminate with a value in T. No runtime error can occur and no nonterminating programs can be written (unless explicitly requested by the programmer).”[3]
- TERMINATING and NON_TERMINATING Pragmas[4]
  - Bypass termination checker

[3] = https://agda.readthedocs.io/en/v2.6.0.1/getting-started/what-is-agda.html
[4] = https://agda.readthedocs.io/en/v2.6.0.1/language/termination-checking.html

Agda as a
Proof Assistant

Recall: What is a monoid?

A **monoid** is an algebraic structure $(S, \circ)$ which satisfies the following properties:

$$
\begin{array}{llll}
(\mathrm{S0}) & : & \text{Closure} & \forall a, b \in S : a \circ b \in S \\
(\mathrm{S1}) & : & \text{Associativity} & \forall a, b, c \in S : a \circ (b \circ c) = (a \circ b) \circ c \\
(\mathrm{S2}) & : & \text{Identity} & \exists e_S \in S : \forall a \in S : e_S \circ a = a = a \circ e_S
\end{array}
$$

The element e_S is called the identity element.

Some proof assistance from Agda - short demo

- Addition in $\mathbb{N}$ is associative
- Proving an equation in $\mathbb{N}$, and why it is tedious

We can do better

A general procedure for equational reasoning in commutative monoids

We can do better

A general procedure for equational reasoning in commutative monoids

We define monoid expressions by the number of variables

```
data Expr n : Set where
  var  : (i : Fin n) → Expr n
  _⊕_  : (a b : Expr n) → Expr n
  zero : Expr n
```

We can do better

A general procedure for equational reasoning in commutative monoids

We normalize and compare the expressions

```
norm : ∀ {n} → Expr n → Expr n
norm = reify ∘ eval

eval : ∀ {n} → Expr n → NF n
eval (var i) = unit i
eval (a ⊕ b) = zipWith _+_ (eval a) (eval b)
eval zero    = const 0

reify : ∀ {n} → NF n → Expr n
reify nf = poly nf vars
```

We can do better

A general procedure for equational reasoning in commutative monoids

Building the equation itself, we prove the equality by normalizing both sides. It can be proven that normalizing the equation keeps it semantically equal. (soundness)

```
data Eqn n : Set where
  _==_ : (a b : Expr n) → Eqn n

eqn₁ = build 4 λ a b c d →
          (a ⊕ b) ⊕ (c ⊕ d) == (a ⊕ c) ⊕ (b ⊕ d)
```

We can do better

A general procedure for equational reasoning in commutative monoids

The proof function uses a module from the standard library

```
prove : ∀ {n}(eq : Eqn n) ρ → [| normEqn eq |] ρ → [| eq |] ρ
prove (a == b) ρ prf =
  begin
    [ a ] ρ
      ≈⟨ sound a ρ ⟩
    [ norm a ] ρ
      ≈⟨ prf ⟩
    [ norm b ] ρ
      ≈⟨ sym (sound b ρ) ⟩
    [ b ] ρ
```