Agda I: Introduction

CAS CS 400: Type Theory and Mechanized Reasoning

January 29, 2023

Outline

Administrivia

Homework 1 will be assigned on Thursday.

We're doing Agda instead of Lean (unless there is strong dissent).

Enrollment is still low. I will keep you updated.

Today

See the basics of Agda as compare it to OCaml.

Tour the features that make Agda powerful (and strange).

Beginning looking at dependent types, and what can be done with them.

Compose a couple small Agda programs.

Outline

About Agda

Agda is a functional programming language developed by Ulf Norell out of Chamlers University.

It supports dependent types, the language feature that will be at the center of this course.

This means it can be used as a proof assistant in the framework of higher-order intuitionistic logic.

About Agda

Agda is a functional programming language developed by Ulf Norell out of Chamlers University.

It supports dependent types, the language feature that will be at the center of this course.

This means it can be used as a proof assistant in the framework of higher-order intuitionistic logic.

Agda is a functional programming language developed by Ulf Norell out of Chamlers University.

It supports dependent types, the language feature that will be at the center of this course.

This means it can be used as a proof assistant in the framework of higher-order intuitionistic logic.

An Aside: Our Motivation

I want to start off by thinking of Agda as a functional language with a new feature (i.e., dependent types).

I don't want to start off by getting too deep into the implications of this. For now, let's think of this as an experimentation period.

Caveats.

- Agda is not the same thing as dependent type theory
- Agda has its drawbacks

An Aside: Our Motivation

I want to start off by thinking of Agda as a functional language with a new feature (i.e., dependent types).

I don't want to start off by getting too deep into the implications of this. For now, let's think of this as an experimentation period.

Caveats.

- Agda is not the same thing as dependent type theory
- Agda has its drawbacks

Installing Agda

- The Agda Wiki
- Installation instructions (Agda Docs)
- Agda Pad (Online Emacs Playground)
- Agda Mode (VSCode)
- Agda Standard Library
  - GitHub Repository
  - Installation Guide

After today's meeting, I am happy to help with setup.

Similarities with OCaml (Overview)

- (anonymous) functions, (mutual) recursion
- strong typing, polymorphism, type synonyms
- inductive data types, record types, pattern matching
- let-expressions, where-blocks
- (parameterized) modules

(The syntax is maybe more Haskell-like)

Dissimilarities with OCaml (Overview)

- first-class types
- all functions are *total*
- implicit arguments
- dependent types
- unicode and mixfix operators(!)

Inductive Data Types

In OCaml:

```
type nat
= Zero
| Succ of nat
```

In Agda:

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

Polymorphic Inductive Data Types

In OCaml:

```
type 'a list
= Nil
| Cons of 'a * list 'a
```

In Agda:

```
data List (a : Set) : Set where
  nil : List a
  cons : a → List a → List a
```

- a is called a parameter for the type List

Polymorphic Inductive Data Types

In OCaml:

```
type 'a list
= Nil
| Cons of 'a * list 'a
```

In Agda:

```
data List (a : Set) : Set where
  nil : List a
  cons : a → List a → List a
```

- a is called a parameter for the type List

The type Set

Types are first-class values and Agda. For example, we can *implement* type synonyms.

```
IndexType : Set
IndexType = Nat
```

Set is the type of *types* in Agda (Agda is strongly typed, everything has to have a type).

Question. What is the type of List?

The type Set

Types are first-class values and Agda. For example, we can *implement* type synonyms.

```
IndexType : Set
IndexType = Nat
```

Set is the type of *types* in Agda (Agda is strongly typed, everything has to have a type).

Question. What is the type of List?

The type Set

Types are first-class values and Agda. For example, we can *implement* type synonyms.

```
IndexType : Set
IndexType = Nat
```

`Set` is the type of *types* in Agda (Agda is strongly typed, everything has to have a type).

Question. What is the type of `List`?

(Recursive) Functions and Pattern Matching

In OCaml:

```
let rec concat r l =
  match r with
  | Nil -> l
  | Cons x xs -> Cons x (concat xs l)
```

In Agda:

```
concat : List Nat → List Nat → List Nat
concat nil l = l
concat (cons x xs) l = cons x (concat xs l)
```

- This is much more Haskell-like
- **all** functions **must** have type signatures

(Recursive) Functions and Pattern Matching

In OCaml:

```
let rec concat r l =
  match r with
  | Nil -> l
  | Cons x xs -> Cons x (concat xs l)
```

In Agda:

```
concat : List Nat → List Nat → List Nat
concat nil l = l
concat (cons x xs) l = cons x (concat xs l)
```

- This is much more Haskell-like
- all functions **must** have type signatures

(Recursive) Functions and Pattern Matching

In OCaml:

```
let rec concat r l =
  match r with
  | Nil -> l
  | Cons x xs -> Cons x (concat xs l)
```

In Agda:

```
concat : List Nat → List Nat → List Nat
concat nil l = l
concat (cons x xs) l = cons x (concat xs l)
```

- This is much more Haskell-like
- **all** functions **must** have type signatures

Let-Expressions

Let-expressions in OCaml:

```
let squared_distance x1 y1 x2 y2 =
  let x_diff = x1 - x2 in
  let y_diff = y1 - y2 in
  x_diff * x_diff + y_diff * y_diff
```

Let-expressions in Agda:

```
squared-distance : ℤ → ℤ → ℤ → ℤ → ℤ
squared-distance x1 y1 x2 y2 =
  let x-diff = x1 - x2
      y-diff = y1 - y2
  in
  x-diff * x-diff + y-diff * y-diff
```

- You may need to include type annotations on let-defined names.

Let-Expressions

Let-expressions in OCaml:

```
let squared_distance x1 y1 x2 y2 =
  let x_diff = x1 - x2 in
  let y_diff = y1 - y2 in
  x_diff * x_diff + y_diff * y_diff
```

Let-expressions in Agda:

```
squared-distance : ℤ → ℤ → ℤ → ℤ → ℤ
squared-distance x1 y1 x2 y2 =
  let x-diff = x1 - x2
      y-diff = y1 - y2
  in
  x-diff * x-diff + y-diff * y-diff
```

- You may need to include type annotations on let-defined names.

Where–Blocks

Where–blocks in Agda:

```
squared-distance : ℤ → ℤ → ℤ → ℤ → ℤ
squared-distance x1 y1 x2 y2 =
  x-diff * x-diff + y-diff * y-diff where
    x-diff = x1 - x2
    y-diff = y1 - y2
```

- This is more Haskell–y

Named Arguments and Dependency

Labeled Arguments in OCaml:

```
let rec concat ~left_arg:r l =
  match r with
  | Nil -> l
  | Cons x xs -> Cons x (concat xs l)
```

Named Arguments in Agda:

```
concat : (left_arg : List Nat) → List Nat → List Nat
concat nil l = l
concat (cons x xs) l = cons x (concat xs l)
```

- These are *not* the same. **The small difference:** Labeled arguments are used in function calls.
- **The big difference:** named argument in the type can be used in *other parts of the type.*
- **This is a fundamental feature of Agda**

Named Arguments and Dependency

Labeled Arguments in OCaml:

```
let rec concat ~left_arg:r l =
  match r with
  | Nil -> l
  | Cons x xs -> Cons x (concat xs l)
```

Named Arguments in Agda:

```
concat : (left_arg : List Nat) → List Nat → List Nat
concat nil l = l
concat (cons x xs) l = cons x (concat xs l)
```

- These are *not* the same. **The small difference:** Labeled arguments are used in function calls.
- **The big difference:** named argument in the type can be used in *other parts of the type.*
- **This is a fundamental feature of Agda**

Named Arguments and Dependency

Labeled Arguments in OCaml:

```
let rec concat ~left_arg:r l =
  match r with
  | Nil -> l
  | Cons x xs -> Cons x (concat xs l)
```

Named Arguments in Agda:

```
concat : (left_arg : List Nat) → List Nat → List Nat
concat nil l = l
concat (cons x xs) l = cons x (concat xs l)
```

- These are *not* the same. **The small difference:** Labeled arguments are used in function calls.
- **The big difference:** named argument in the type can be used in *other parts of the type.*
- This is a fundamental feature of Agda

Named Arguments and Dependency

Labeled Arguments in OCaml:

```
let rec concat ~left_arg:r l =
  match r with
  | Nil -> l
  | Cons x xs -> Cons x (concat xs l)
```

Named Arguments in Agda:

```
concat : (left_arg : List Nat) → List Nat → List Nat
concat nil l = l
concat (cons x xs) l = cons x (concat xs l)
```

- These are *not* the same. **The small difference:** Labeled arguments are used in function calls.
- **The big difference:** named argument in the type can be used in *other parts of the type.*
- **This is a fundamental feature of Agda**

Example: Polymorphism

In OCaml:

```
let snoc x l =
  match l with
  | Nil -> Cons x Nil
  | Cons y ys -> Cons y (snoc x ys)
```

In Agda:

```
snoc : (a : Set) → a → List a → List a
snoc a x nil = cons x nil
snoc a x (cons y ys) = cons y (snoc a x ys)
```

- OCaml functions are polymorphic by assumption. This has to do with the way type inference is done in OCaml.
- In Agda we can use named arguments to *implement* polymorphism.

Example: Polymorphism

In OCaml:

```
let snoc x l =
  match l with
  | Nil -> Cons x Nil
  | Cons y ys -> Cons y (snoc x ys)
```

In Agda:

```
snoc : (a : Set) → a → List a → List a
snoc a x nil = cons x nil
snoc a x (cons y ys) = cons y (snoc a x ys)
```

- OCaml functions are polymorphic by assumption. This has to do with the way type inference is done in OCaml.
- In Agda we can use named arguments to *implement* polymorphism.

Example: Polymorphism

In OCaml:

```
let snoc x l =
  match l with
  | Nil -> Cons x Nil
  | Cons y ys -> Cons y (snoc x ys)
```

In Agda:

```
snoc : (a : Set) → a → List a → List a
snoc a x nil = cons x nil
snoc a x (cons y ys) = cons y (snoc a x ys)
```

- OCaml functions are polymorphic by assumption. This has to do with the way type inference is done in OCaml.
- In Agda we can use named arguments to *implement* polymorphism.

Implicit Arguments

The previous example is a bit unsatisfying: have to give the type explicitly as an argument.

```
l : List Nat
l = snoc Nat zero (cons (succ zero) nil)
```

But the since the argument doesn't play a role in the computation, we can make it *implicit*:

```
snoc : {a : Set} → a → List a → List a
snoc x nil = cons x nil
snoc x (cons y ys) = cons y (snoc x ys)

l : List Nat
l = snoc zero (cons (succ zero) nil)
```

Please be advised: implicit arguments are tricky

Example: Another way of writing List

```
data List : Set → Set where
  nil : {a : Set} → List a
  cons : {a : Set} → a → List a → List a
```

- This definition is equivalent to this previous one.
- Formally, this is an indexed typed instead of a parametrized type.
- In general, note that *inductive data types can define more than just types.*

Example: Another way of writing List

```
data List : Set → Set where
  nil : {a : Set} → List a
  cons : {a : Set} → a → List a → List a
```

- This definition is equivalent to this previous one.
- Formally, this is an indexed typed instead of a parametrized type.
- In general, note that *inductive data types can define more than just types.*

Example: Another way of writing List

```
data List : Set → Set where
  nil : {a : Set} → List a
  cons : {a : Set} → a → List a → List a
```

- This definition is equivalent to this previous one.
- Formally, this is an indexed typed instead of a parametrized type.
- In general, note that *inductive data types can define more than just types.*

Example: Another way of writing List

```
data List : Set → Set where
  nil : {a : Set} → List a
  cons : {a : Set} → a → List a → List a
```

- This definition is equivalent to this previous one.
- Formally, this is an indexed typed instead of a parametrized type.
- In general, note that *inductive data types can define more than just types.*

Example: Generalized Algebraic Data Types

In OCaml:

```
type _ t =
  | Int : int t
  | Bool : bool t
```

In Agda:

```
data t : Set → Set where
  | nat : t Nat
  | bool : t Bool
```

- We can *implement* GADTs because of types are first-class values.
- *Note.* polymorphic ADT versus GADT is *exactly* being parameterized by type versus being indexed by a type.

Example: Generalized Algebraic Data Types

In OCaml:

```
type _ t =
  | Int : int t
  | Bool : bool t
```

In Agda:

```
data t : Set → Set where
  | nat : t Nat
  | bool : t Bool
```

- We can *implement* GADTs because of types are first-class values.
- *Note.* polymorphic ADT versus GADT is *exactly* being parameterized by type versus being indexed by a type.

Example: Generalized Algebraic Data Types

In OCaml:

```
type _ t =
  | Int : int t
  | Bool : bool t
```

In Agda:

```
data t : Set → Set where
  | nat : t Nat
  | bool : t Bool
```

- We can *implement* GADTs because of types are first-class values.
- *Note.* polymorphic ADT versus GADT is *exactly* being parameterized by type versus being indexed by a type.

Let’s do a demo.

Syntactic Conveniences: Unicode and Mixfix Operators

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

if_then_else_ : Bool → ℕ → ℕ → ℕ
if true then n else m = n
if false then n else m = m
```

We'll try not to depend on this too much, but it's *very nice* when you start writing more complex programs.

Modules

Agda has a module system similar to that of OCaml's. For now we will just use:

- `import Module.name` to bring the module into view
- `open Module.name` to bring the functions in the module into view
- `open import Module.name` to do both
- `open import Module.name using (f1; f2)` to bring a restricted list of functions into view

Totality

All functions in Agda are total, meaning we can’t write partial functions.

```
-- These won't pass type-checking

foo : Bool → Bool
foo true = false

bar : ℕ → ℕ
bar x = bar x
```

- Pattern matches must be complete.
- Recursive calls must be one structurally smaller values
- *An aside.* It is impossible to write a function that checks if a function is total.

Totality

All functions in Agda are total, meaning we can't write partial functions.

```
-- These won't pass type-checking

foo : Bool → Bool
foo true = false

bar : ℕ → ℕ
bar x = bar x
```

- Pattern matches must be complete.
- Recursive calls must be one structurally smaller values
- *An aside.* It is impossible to write a function that checks if a function is total.

Totality

All functions in Agda are total, meaning we can’t write partial functions.

```
-- These won't pass type-checking

foo : Bool → Bool
foo true = false

bar : ℕ → ℕ
bar x = bar x
```

- Pattern matches must be complete.
- Recursive calls must be one structurally smaller values
- *An aside.* It is impossible to write a function that checks if a function is total.

Totality

All functions in Agda are total, meaning we can't write partial functions.

```
-- These won't pass type-checking

foo : Bool → Bool
foo true = false

bar : ℕ → ℕ
bar x = bar x
```

- Pattern matches must be complete.
- Recursive calls must be one structurally smaller values
- *An aside.* It is impossible to write a function that checks if a function is total.

Let's do a demo.

Outline

At a High Level

```
data NatBox : ℕ → Set where
  box : (n : ℕ) → NatBox n
```

Values can parameterize and index types. They can appear at the type level.

This comes from our ability for *named parameters* to appear in other parts of the type.

Example: Non-Empty Lists

Let's do a demo.

An Aside: Predicates

A *predicate* is a way of delineating a subset of objects by a property. For example, non-emptiness for lists.

```
data NonEmpty : {a : Set} → List a → Set where
  isNonEmpty : {a : Set} →
    (x : a) →
    (xs : List a) →
    NonEmpty (cons x xs)
```

In Agda, a predicate over a type `T` is just a function of type `T → Set`.

Question. How does this relate to our discussion about induction?

Outline

Summary

Agda is a dependently types functional programming language. It behaves a lot like OCaml, but it has this bizarre features:

- types are first-class values
- named parameters can be used throughout a type

Dependent Types can be used to inject “non-type” values into types. Many modern type features can be *implemented* in terms of dependent types.