Introduction to Agda

Lecture 11 of CSE 3100
Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

“Program testing can be used to show the presence of bugs, but never to show their absence!”

– Edsger W. Dijkstra

Lecture plan

- A brief overview of formal verification, dependent types, and Agda
- Syntax differences between Agda and Haskell
- Interactive programming in Agda
- Types as first-class values
- Total functional programming

When testing is just not enough

Question. In what situations might testing not be enough to ensure software works correctly?

When testing is just not enough

Question. In what situations might testing not be enough to ensure software works correctly?

- … failure is very costly (e.g. spacecraft, medical equipment, self-driving cars)
- … the software is difficult to update (e.g. embedded software)
- … it is security-sensitive (e.g. banking, your private chats)
- … errors are hard to detect or not apparent until much later (e.g. compilers, concurrent systems)

Formal verification

Formal verification is a collection of techniques for proving correctness of programs with respect to a certain formal specification.

These techniques often rely on ideas from formal logic and mathematics to ensure a very high degree of trustworthiness.

Forms of formal verification

- Model checking systematically explores all possible executions of a program.
- Deductive verification uses external or internal tools to analyse and prove correctness of a program.
- Lightweight formal methods *automatically* verify a class of properties such as type safety or memory safety.

Why dependent types?

Dependent types are a form of deductive verification that is embedded in the programming language.

Advantages.

- No different syntax to learn or tools to install
- Tight integration between IDE and type system
- Express invariants of programs in their types
- Use same syntax for programming and proving

Formally verifying a program should not be more difficult than writing the program in the first place!

The Agda language

Agda is a purely functional programming language similar to Haskell.

Unlike Haskell, it has full support for dependent types.

It also supports interactive programming with help from the type checker.

Building your own Agda

An important goal of Agda is to experiment with new language features.

As a consequence, many common language features are not built into Agda, but they can be defined.

Example. `if/then/else` is not built into Agda but can be defined as a function.

While we could import these from the standard library, here we will instead build them ourselves from the ground up.

Hello, Agda

Installing Agda

Binary release. (Linux/WSL/VM)

```
sudo apt install agda
```

From source. (Cabal/Stack)

`cabal install Agda` or

`stack install Agda`

Via the VS Code plugin.

Install the `agda-mode` plugin and enable the Agda Language Server in the settings.

Installing an editor for Agda

The following editors have support for Agda:

- **VS Code**: Install the `agda-mode` plugin
- **Emacs**: Plugin is distributed with Agda (run `agda-mode setup`)
- **Atom**: `https://atom.io/packages/agda-mode`
- **Vim**: `https://github.com/derekelkins/agda-vim`

A first Agda program

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

This program:

- Defines a datatype Greeting with one constructor hello.
- Defines a function greet of type Greeting that returns hello.

Loading an Agda file

You can load an Agda file by pressing `Ctrl+c` followed by `Ctrl+l`.

Once the file is loaded (and there are no errors), other commands become available:

`Ctrl+c Ctrl+d` Infer type of an expression.

`Ctrl+c Ctrl+n` Evaluate an expression.

Syntax of Agda vs. Haskell

Basic syntax differences

Typing uses a single colon:
b : Bool instead of b :: Bool.

Naming has fewer restrictions: any name can start with small or capital letter, and symbols can occur in names.

Whitespace is required more often: `1+1` is a valid function name, so you need to write `1 + 1` instead.

Infix operators are indicated by underscores:
`_+_` instead of `(+)`

Unicode syntax

Agda allows unicode characters in its syntax:

- $\rightarrow$ can be used instead of `->`
- λ can be used instead of `\`
- Other symbols can also be used as (parts of) names of functions, variables, or types: $\times$, Σ, $\top$, $\bot$, $\equiv$, $\langle$, $\rangle$, $\circ$, ...

Entering unicode

Editors with Agda support will replace LaTeX-like syntax (e.g. `\to`) with unicode:

→	`\to`
λ	`\lambda`
×	`\times`
Σ	`\Sigma`
⊤	`\top`
⊥	`\bot`
≡	`\equiv`
…	

Quiz question

Question. Which is NOT a valid name for an Agda function?

1. `1+1=2`
2. `foo bar`
3. `λ→×Σ`
4. `if_then_else_`

Declaring new datatypes

To declare a datatype in Agda, we need to give the full type of each constructor:

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

We also need to specify that Bool itself has type Set (see later).

Defining functions by pattern matching

Just as in Haskell, we can define new functions by pattern matching:

```
not : Bool → Bool
not true  = false
not false = true

_||_ : Bool → Bool → Bool
false || false = false
_     || _     = true
```

The type of natural numbers

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

one   = suc zero
two   = suc one
three = suc two
four  = suc three
five  = suc four
```

Builtin support for numbers

Writing numbers with zero and suc is annoying and inefficient. We can enable Agda's support for machine integers as follows:

```
{-# BUILTIN NATURAL Nat #-}
```

Agda will then convert between numerals and zero/suc representation automatically:

```
one'   = 1
two'   = 2
three' = 3
```

Functions on natural numbers

```
isEven : Nat → Bool
isEven zero          = true
isEven (suc zero)    = false
isEven (suc (suc x)) = isEven x

_+_ : Nat → Nat → Nat
zero    + y = y
(suc x) + y = suc (x + y)
```

Priority of infix operators

You can specify the priority and associativity of an operator with infixl or infixr:

```
infixl 10 _+_
infixl 20 _*_

myNumber = 1 + 2 * 3 + 4
- Parsed as ((1 + (2 * 3)) + 4
- With infixr, it would be parsed as
- 1 + ((2 * 3) + 4) instead.
```

Integers in Agda

Question. How would you define a type of integers in Agda?

Integers in Agda

Question. How would you define a type of integers in Agda?

Answer. Here is one possibility:

```
data Int : Set where
  pos  : Nat → Int
  zero : Int
  neg  : Nat → Int
```

where pos n represents the number $1 + n$ and neg n represents $-(1 + n)$.

Interactive programming in Agda

Holes in programs

A hole is a part of a program that is not yet complete. A hole can be created by writing `?` or `{!!}` and loading the file (`Ctrl+c Ctrl+l`).

New commands for files with holes:

`Ctrl+c Ctrl+,`	Give information about the hole
`Ctrl+c Ctrl+c`	Case split on a variable
`Ctrl+c Ctrl+space`	Give a solution for the hole

Demo session

Exercise. Define the following Agda functions:

- $\mathsf{maximum} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$
- $\_*\_ : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$
- $\_\leq\_ : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool}$

Summary of interactive commands

`Ctrl+c Ctrl+l`	**L**oad the file
`Ctrl+c Ctrl+d`	**D**educe type of an expression
`Ctrl+c Ctrl+n`	**N**ormalise an expression
`Ctrl+c Ctrl+,`	Get information about the hole
`Ctrl+c Ctrl+c`	**C**ase split on a variable
`Ctrl+c Ctrl+space`	Give a solution for the hole

These commands will become more and more useful, so start using them now!

Types as first-class values

The type `Set`

In Agda, types such as Nat and (Bool → Bool) are themselves expressions of type Set.

We can pass around and return values of type Set just like values of any other type.

Example. Defining a type alias as a function:

```
MyNat : Set
MyNat = Nat

myFour : MyNat
myFour = suc (suc (suc (suc zero)))
```

Polymorphic functions in Agda

We can define polymorphic functions as functions that take an argument of type Set:

```
id : (A : Set) → A → A
id A x = x
```

For example, we have `id Nat zero : Nat` and `id Bool true : Bool`.

Hidden arguments

To avoid repeating the type at which we apply a polymorphic function, we can declare it as a hidden argument using curly braces:

$$\mathsf{id} : \{A : \mathsf{Set}\} \to A \to A$$
$$\mathsf{id}\ x = x$$

Now we have $\mathsf{id}\ \mathsf{zero} : \mathsf{Nat}$ and $\mathsf{id}\ \mathsf{true} : \mathsf{Bool}$.

If/then/else as a function

We can define if/then/else in Agda as follows:

```
if_then_else_ : {A : Set} →
  Bool → A → A → A
(if true  then x else y) = x
(if false then x else y) = y
```

This is an example of a mixfix operator.

Example usage.

```
test : Nat → Nat
test x = if (x ≤ 9000) then 0 else 42
```

Polymorphic datatypes

Just like we can define polymorphic functions, we can also define polymorphic datatypes by adding a parameter $(A : \mathsf{Set})$:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
infixl 5 _::_
```

Note. Agda does not have built-in support for list syntax $[1, 2, 3]$. Instead, we have to write $1 :: 2 :: 3 :: []$.

A tuple type in Agda

Agda does not have a builtin type of tuples (x, y), but we can define the product type $A \times B$:

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

fst : {A B : Set} → A × B → A
fst (x , y) = x

snd : {A B : Set} → A × B → B
snd (x , y) = y
```

No pattern matching on `Set`

It is not allowed to pattern match on arguments of type Set:

```
-- Not valid Agda code:
sneakyType : Set → Set
sneakyType Bool = Nat
sneakyType Nat  = Bool
```

One reason for this is that Agda (like Haskell) erases all types during compilation.

Total functional programming

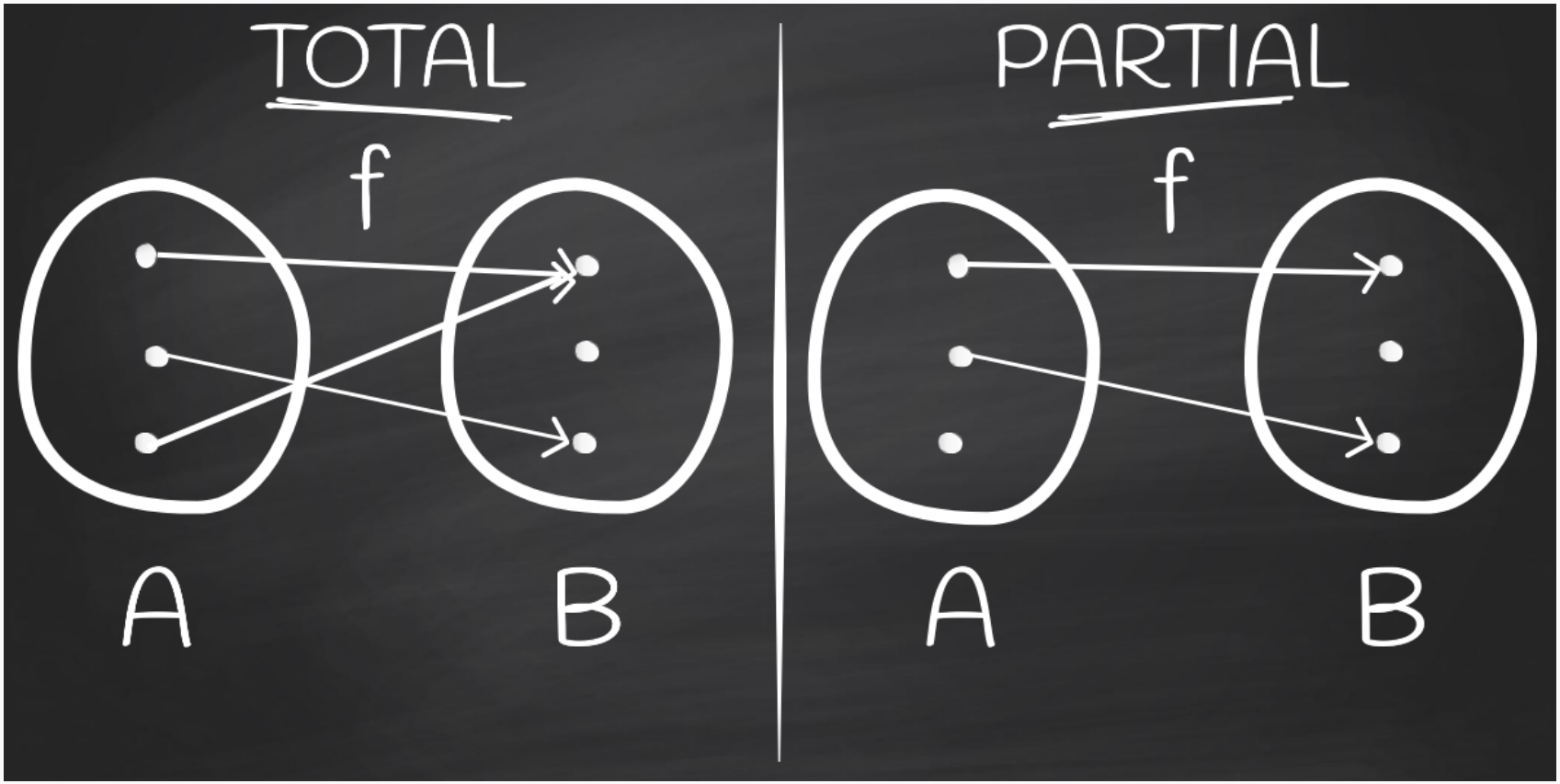
Total functional programming

In contrast to Haskell, Agda is a total language:

- **NO** runtime errors
- **NO** incomplete pattern matches
- **NO** non-terminating functions

So functions are true functions in the mathematical sense: evaluating a function call always returns a result in finite time.

Totality in mathematics[1]



[1]Source: https://kowainik.github.io/posts/totality

Why should we care about totality?

Some reasons to write total programs:

- Better guarantees of correctness
- Spend less time debugging infinite loops
- Easier to refactor without introducing bugs
- Less need to document valid inputs

Totality is also crucial for working with dependent types and using Agda as a proof assistant (see coming lectures).

Coverage checking

Agda performs a coverage check to ensure all definitions by pattern matching are complete:

```
pred : Nat → Nat
pred (suc x) = x
```

Incomplete pattern matching for pred.
Missing cases: pred zero

Termination checking

Agda performs a termination check to ensure all recursive definitions are terminating:

```
inf : Nat → Nat
inf x = 1 + inf x
```

```
Termination checking failed for the following
functions: inf
Problematic calls: inf x
```

To solve or not to solve the halting problem

Question. Isn't it impossible to determine whether a function is terminating? Or does Agda solve the halting problem?

To solve or not to solve the halting problem

Question. Isn't it impossible to determine whether a function is terminating? Or does Agda solve the halting problem?

Answer. No, Agda only accepts functions that are structurally recursive, and rejects all other functions.

Structural recursion

Agda only accepts functions that are structurally recursive: the argument of each recursive call must be a subterm of the argument on the left of the clause.

```
f : Nat → Nat
f (suc (suc x)) = f zero
f (suc x)       = f (suc (suc x))
f zero          = zero
```

The function f is terminating but not structurally recursive, so it is rejected.

Quiz question

Question. Which of these are possible to be defined in Agda?

- A data type with infinitely many elements
- A function that loops forever
- A function that pattern matches on types
- A function with infinitely many cases

Discussion question

Question. Is it possible to implement a function of type $\{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{Nat} \to A$ in Agda?

What's next?

Next lecture: Dependent types

To do:

- Read the lecture notes:
  - This lecture: section 1 of Agda lecture notes
  - Next lecture: section 2 of Agda lecture notes
- Install Agda on your computer
- Start on Agda exercises on Weblab