Introduction to Advanced Functional Programming

In this module you will learn the functional programming language Agda. We assume that you have learned Haskell before attending this module (the prerequisites also include data structures and algorithms, theories of computation, and mathematical and logical foundations of computer science).

There are three main distinguishing features of Agda compared to Haskell.

1. Agda has a richer set of types, including *dependent types*.

2. You can write Agda code to *prove* properties of your Agda definitions.

   More importantly, you can *write* such properties in Agda.

3. In Agda all computations must terminate.

Plan of this introductory handout

In this handout we give a taste of Agda, anticipating things that we will learn in future handouts.

1. We will show you some examples of Agda code.

   Hopefully there will be some things you will understand without further explanation.

   But there will be lots of things that require explanation.

2. After giving the examples, we will explain them.

The idea of this handout is *not* to fully explain the examples below, but instead to give you a *flavour* of what Agda is like. The full explanation of what follows, and more, is provided in future handouts.

Main ideas and questions

How can we think and reason about programs? How can we write better programs?

This handout is both an Agda file and a Markdown file

The file for this handout is `introduction.lagda.md`. It is in Markdown format and includes Agda code that can be understood directly by Agda. See the Literate Programming section of the Agda documentation.

Initial examples of types in Agda

We begin with some examples you are familiar from Haskell. Notice that the syntax is similar but slightly different.

```
data Bool : Type where
 true false : Bool

data Maybe (A : Type) : Type where
 nothing : Maybe A
 just    : A → Maybe A

data Either (A B : Type) : Type where
 left  : A → Either A B
 right : B → Either A B

data ℕ : Type where
 zero : ℕ
 suc  : ℕ → ℕ

{-# BUILTIN NATURAL ℕ #-}

data List (A : Type) : Type where
 []   : List A
 _::_ : A → List A → List A

infixr 10 _::_

data BinTree (A : Type) : Type where
 empty : BinTree A
 fork  : A → BinTree A → BinTree A → BinTree A

data RoseTree (A : Type) : Type where
 fork : A → List (RoseTree A) → RoseTree A
```

The pragma `BUILTIN NATURAL` is to get syntax sugar to be able to write 0,1,2,3,... rather than the more verbose

- zero
- suc zero
- suc (suc zero)
- suc (suc (suc zero))
- ⋯

We pronounce `suc` as "successor".

Examples definitions using the above types in Agda

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

_&&_ : Bool → Bool → Bool
true  && y = y
false && y = false

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

infixr 20 _||_
infixr 30 _&&_

if_then_else_ : {A : Type} → Bool → A → A → A
if true  then x else y = x
if false then x else y = y

_+_ : ℕ → ℕ → ℕ
zero  + y = y
suc x + y = suc (x + y)

_*_ : ℕ → ℕ → ℕ
zero  * y = 0
suc x * y = x * y + y

infixl 20 _+_
infixr 30 _*_

sample-list₀ : List ℕ
sample-list₀ = 1 :: 2 :: 3 :: []

sample-list₁ : List Bool
sample-list₁ = true || false && true :: false :: true :: true :: []

sample-list₂ : List (Either Bool ℕ)
sample-list₂ = left true :: right 2 :: right 17 :: left false :: []

length : {A : Type} → List A → ℕ
length []        = 0
length (x :: xs) = 1 + length xs

_++_ : {A : Type} → List A → List A → List A
[] ++ ys        = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

```
infixl 20 _++_

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

[_] : {A : Type} → A → List A
[ x ] = x :: []

reverse : {A : Type} → List A → List A
reverse []        = []
reverse (x :: xs) = reverse xs ++ [ x ]

rev-append : {A : Type} → List A → List A → List A
rev-append []        ys = ys
rev-append (x :: xs) ys = rev-append xs (x :: ys)

rev : {A : Type} → List A → List A
rev xs = rev-append xs []
```

The function `reverse` is slow for large lists as it runs in quadratic time, but the function `rev` is much faster as it runs in linear time. Although the algorithm for `reverse` is simple and clear, that for `rev` is slightly more complicated, and so perhaps we would like to make sure that we didn't make a mistake, by proving that `reverse xs` and `rev xs` are equal. We will do that later.

More sophisticated examples of types in Agda

Sometimes we may wish to consider lists over a type `A` of a given length `n : ℕ`. The elements of this type, written `Vector A n`, are called *vectors*, and the type can be defined as follows:

```
data Vector (A : Type) : ℕ → Type where
 []   : Vector A 0
 _::_ : {n : ℕ} → A → Vector A n → Vector A (suc n)
```

This is called a *dependent type* because it is a type that depends on *elements* `n` of another type, namely `ℕ`.

In Agda, we can't define the `head` and `tail` functions on lists, because types don't have `undefined` elements like in Haskell, which would be needed for the head and tail of the empty list. Vectors solve this problem:

```
head : {A : Type} {n : ℕ} → Vector A (suc n) → A
head (x :: xs) = x

tail : {A : Type} {n : ℕ} → Vector A (suc n) → Vector A n
tail (x :: xs) = xs
```

Agda accepts the above definitions because it knows that the input vector has at least one element, and hence does have a head and a tail.

Dependent types are pervasive in Agda.

The empty type and the unit type

A type with no elements can be defined as follows:

```
data 𝟘 : Type where
```

We will also need the type with precisely one element, which we define as follows:

```
data 𝟙 : Type where
 ⋆ : 𝟙
```

Here is an example of a dependent type defined using the above types:

```
_≣_ : ℕ → ℕ → Type
0     ≣ 0     = 𝟙
0     ≣ suc y = 𝟘
suc x ≣ 0     = 𝟘
suc x ≣ suc y = x ≣ y
```

```
infix 0 _≣_
```

The idea of the above definition is that `x ≣ y` is a type which either has precisely one element, if `x` and `y` are the same natural number, or else is empty, if `x` and `y` are different. The following definition says that for any natural number `x` we can find an element of the type `x ≣ x`.

```
ℕ-refl : (x : ℕ) → x ≣ x
ℕ-refl 0       = ⋆
ℕ-refl (suc x) = ℕ-refl x
```

The identity type former `_≡_`

It is possible to generalize the above definition for any type in place of that of natural numbers as follows:

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

infix 0 _≡_
```

Here are some functions we can define with the identity type:

```
trans : {A : Type} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans (refl x) (refl x) = refl x

sym : {A : Type} {x y : A} → x ≡ y → y ≡ x
sym (refl x) = refl x

ap : {A B : Type} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
ap f (refl x) = refl (f x)
```

The identity type is a little bit subtle and there is a lot to say about it. For the moment, let's convince ourselves that we can convert back and forth between the types `x ≡ y` and `x ≣ y`, in the case that `A` is the type of natural numbers, as follows:

```
back : (x y : ℕ) → x ≡ y → x ≣ y
back x x (refl x) = ℕ-refl x

forth : (x y : ℕ) → x ≣ y → x ≡ y
forth 0       0       ⋆ = refl 0
forth (suc x) (suc y) p = I
 where
  IH : x ≡ y
  IH = forth x y p

  I : suc x ≡ suc y
  I = ap suc IH
```

List reversal revisited

Recall that we defined an easy to understand reversal function `reverse` and a faster, but more difficult to understand function `rev`. To convince ourselves that `rev` is correct, we can write an Agda function with the following type:

```
rev-correct : {A : Type} (xs : List A) → rev xs ≡ reverse xs
```

In order to do that, we first define three auxiliary programs with the following types:

```
[]-right-neutral : {X : Type} (xs : List X) → xs ++ [] ≡ xs

++assoc : {A : Type} (xs ys zs : List A) → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

rev-append-spec : {A : Type} (xs ys : List A)
                → rev-append xs ys ≡ reverse xs ++ ys
```

Here are the definitions of the functions, but we don't expect you to be able to follow the details in this introductory handout:

```
[]-right-neutral []        = refl []
[]-right-neutral (x :: xs) = II
```

```
 where
  IH : xs ++ [] ≡ xs
  IH = []-right-neutral xs

  I : x :: (xs ++ []) ≡ x :: xs
  I = ap (x ::_) IH

  II : (x :: xs) ++ [] ≡ x :: xs
  II = I

++assoc []       ys zs = refl (ys ++ zs)
++assoc (x :: xs) ys zs = II
  where
    IH : (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
    IH = ++assoc xs ys zs

    I : x :: ((xs ++ ys) ++ zs) ≡ x :: (xs ++ (ys ++ zs))
    I = ap (x ::_) IH

    II : ((x :: xs) ++ ys) ++ zs ≡ (x :: xs) ++ (ys ++ zs)
    II = I

rev-append-spec []        ys = refl ys
rev-append-spec (x :: xs) ys = II
 where
  IH : rev-append xs (x :: ys) ≡ reverse xs ++ (x :: ys)
  IH = rev-append-spec xs (x :: ys)

  I : reverse xs ++ ((x :: []) ++ ys) ≡ (reverse xs ++ (x :: [])) ++ ys
  I = sym (++assoc (reverse xs) (x :: []) ys)

  II : rev-append (x :: xs) ys ≡ reverse (x :: xs) ++ ys
  II = trans IH I

rev-correct xs = IV
 where
  I : rev-append xs [] ≡ reverse xs ++ []
  I = rev-append-spec xs []

  II : reverse xs ++ [] ≡ reverse xs
  II = []-right-neutral (reverse xs)

  III : rev-append xs [] ≡ reverse xs
  III = trans I II

  IV : rev xs ≡ reverse xs
  IV = III
```

In the next few handouts, we are going to look at simpler examples first, to prepare you to understand the above definitions of the functions.

Propositions as types

The Curry--Howard interpretation of logic, after Haskell Curry and William Howard, interprets logical statements, also known as propositions, as *types*. Per Martin-Löf extended this interpretation of propositions as types with equality, by introducing the identity type discussed above.

An incomplete table of the Curry--Howard--Martin-Löf interpretation of logical propositions is the following:

Proposition	**Type**
A implies B	function type A → B
∀ x : A, B x	dependent function type (x : A) → B x
equality	identity type `_≡_`

This fragment of logic was enough for us to be able to write the correctness of `rev` as the type

```
{A : Type} (xs : List A) → rev xs ≡ reverse xs
```

which we can read as

> for any type `A` and any list `xs`, we have that `rev xs = reverse xs`,

or, using logical symbolism,

> `∀ A : Type, ∀ xs : List A, rev xs = reverse xs`.

For more complex examples of reasoning about programs, we need to complete the following table:

Logic	English	Type
¬ A	not A	?
A ∧ B	A and B	?
A ∨ B	A or B	?
A → B	A implies B	A → B
∀ x : A, B x	for all x:A, B x	(x : A) → B x
∃ x : A, B x	there is x:A such that B x	?
x = y	x equals y	x ≡ y

This will be the subject of future handouts.

Proofs as functional programs

Notice that we didn't write a *proof*, in the usual mathematical sense, of the statement

> for any type `A` and any list `xs`, we have that `rev xs = reverse xs`.

We instead wrote a *program* of type

> {A : Type} (xs : List A) → rev xs ≡ reverse xs

This is precisely the point of "propositions as types": proofs become functional programs. You may not know a lot (or even anything) about proofs, but you certainly know a lot about functional programming. The interpretation of logical statements as types allows you to apply your expertise as a functional programmer to write (rigorous) proofs checked by the computer.

> If your Agda program compiles without errors, your proof, written as a program, is correct!

The computer checks your proof for you. A proof is nothing but a functional program.

Rigorous reasoning about programs

What if you are not interested in *proving* correctness in the logical sense of proof? You will still need to make sure your programs are correct, one way or another, if you don't want to be fired or sued or lose clients or kill somebody because you programmed a traffic light incorrectly. What this module offers you is a rigorous way to reason about programs. Even if you may not, in practice, prove programs correct using Agda if you work in industry, we intend that by completing this module you will learn what is involved in thinking about programs and arguing that they are correct or satisfy certain desirable properties and don't have certain undesirable properties.

Precise types

So far we have emphasized reasoning about programs, and in particular arguing that they are correct by writing functional programs, as exemplified by the program `rev-correct`.

But there is another way logic is useful for programming. Very often, in all programming languages, and in Haskell in particular, we work with types that contain "junk", and we have to be very careful to ignore junk when it doesn't matter, and take it into account when it does matter.

For example, in Haskell we normally work with the type of *all* binary trees, even in situations where we are only interested in *binary search trees*. Using logical statements, we can define a type which has all binary search trees and no junk. In this example, logic is used as *part* of the programming activity, rather than just as a sanity check *after* the programs have been written.

Another example is monads. In Haskell we just write the types of the monad operations. But monads have to satisfy certain laws to really be monads, and this is important. In Agda we can write the monad laws, and, when we define a monad, ensure that it is a monad by writing suitable Agda code. There are many more examples like that.

One way to write better programs is to work with more precise types. This is one important thing that we will explore in this module.

Do people in industry use tools such as Agda?

Yes, definitely. A lot. There are well-paid jobs in industry in software verification.

- A List of companies that use formal verification methods in software engineering
- Formal methods & industry
- Agda Jobs

Are we advocating that all programmers should switch to Agda?

No, of course not. You shouldn't do that except in special situations. What we are saying is that you should learn to reason about programs, and we are using Agda for that purpose, in particular because it allows to reason about programs and record the reasoning in Agda itself, and because it allows to define more precise types by the availability of logic at the programming level, via the proposition-as-types and proofs-as-programs understanding of logic. But the ideas discussed in this module to reason about programs can be applied beyond Agda and beyond functional programming, and they have.

Example.

- A verified C compiler

Is Agda unique in being able to express both programs and logical statements?

No, for example, there are also Coq and Lean among many others.

Agda installation

We offer you a complete Agda environment

- in the UG04 Lab machines, and
- as a Jupyter Notebook accessible via a browser,

so that you don't need to install it in your own machine.

We are using Agda 2.6.2, the latest version at the time of writing. There is a standard library, but we are not going to use it, at least not to begin with.

You may still wish to install Agda in your own machine, but we are not able to provide support, although you are welcome to ask questions on Teams. It is much easier to install on Linux and Mac, and possible on Windows.

Agda resources that you will need for daily use

- Getting started
- Language reference
- Emacs mode
- Key bindings
- Global commands
- Commands in context of a goal
- Other commands
- Unicode input

Emacs resources

Agda has a very nice interactive environment for writing programs which works in the text editor emacs.

- Install emacs
- A guided tour of Emacs
- Emacs manual
- Emacs reference card
- A tutorial introduction to emacs

The Getting Started section of the online book Programming Language Foundations in Agda has a nice installation guide as well as a summary of emacs commands.

Visual Studio Code

There is plugin for Agda support available on the Visual Studio Marketplace. We haven't tried it.

Further reading

- The Agda Wiki
- Agda tutorials

- Dependently Typed Programming in Agda
- Dependent types at work

Advanced reading

- Programming Language Foundations in Agda