The Curry-Howard Correspondence

Lecture 11 of CSE 3100
Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

"Every good idea will be discovered twice: once by a logician and once by a computer scientist."

– Philip Wadler

Lecture plan

- The Curry-Howard correspondence between type systems and logic
- Classical logic vs. constructive logic
- Curry-Howard for predicate logic
- Proof by induction in Agda

The Curry-Howard Correspondence

Goal of today's lecture

Goal. We want to write proofs that our program satisfies certain properties, and have the compiler check that these proofs are correct.

Examples.

- For any x : Nat, $x + x$ is an even number.
- The length of map f xs is equal to the length of xs.
- reverse = foldr ($\lambda x\ xs \rightarrow xs$ ++ x) [] and reverse' = foldl ($\lambda xs\ x \rightarrow x$ :: xs) [] always return the same result.

Formal verification with dependent types

Reminder. Formal verification is the process proving correctness of programs with respect to a certain formal specification.

Our goal is to use Agda as a proof assistant for doing formal verification.

To do this, we first need to answer the question: what exactly is a proof?

What even is a proof? (1/3)

Traditionally, a proof is a sequence of statements where each statement is a direct consequence of previous statements.

Example. A proof that if (1) $A \Rightarrow B$ and (2) $A \wedge C$, then $B \wedge C$:

(3)	A	(follows from 2)
(4)	B	(modus ponens with 1 and 3)
(5)	C	(follows from 2)
(6)	$B \wedge C$	(follows from 4 and 5)

What even is a proof? (2/3)

We can make the dependencies of a proof more explicit by writing it down as a proof tree.

Example. Here is the same proof that if (1) $A \Rightarrow B$ and (2) $A \wedge C$, then $B \wedge C$:

$$\dfrac{\dfrac{A \Rightarrow B^{(1)} \qquad \dfrac{A \wedge C^{(2)}}{A}}{B} \qquad \dfrac{A \wedge C^{(2)}}{C}}{B \wedge C}$$

What even is a proof? (3/3)

To represent these proofs in a programming language, we can annotate each node of the tree with a proof term:

$$\dfrac{p : A \Rightarrow B \qquad \dfrac{q : A \wedge C}{\mathsf{fst}\ q : A}}{p\ (\mathsf{fst}\ q) : B} \qquad \dfrac{q : A \wedge C}{\mathsf{snd}\ q : C}$$

$$\dfrac{\dfrac{p : A \Rightarrow B \qquad \dfrac{q : A \wedge C}{\mathsf{fst}\ q : A}}{p\ (\mathsf{fst}\ q) : B} \qquad \dfrac{q : A \wedge C}{\mathsf{snd}\ q : C}}{(p\ (\mathsf{fst}\ q), \mathsf{snd}\ q) : B \wedge C}$$

What even is a proof? (3/3)

To represent these proofs in a programming language, we can annotate each node of the tree with a proof term:

$$\dfrac{\dfrac{p : A \Rightarrow B \qquad \dfrac{q : A \wedge C}{\textsf{fst}\ q : A}}{p\ (\textsf{fst}\ q) : B} \qquad \dfrac{q : A \wedge C}{\textsf{snd}\ q : C}}{(p\ (\textsf{fst}\ q), \textsf{snd}\ q) : B \wedge C}$$

Hmm, these proof terms start to look a lot like functional programs…

The Curry-Howard correspondence

Haskell B. Curry

We can interpret logical propositions ($A \land B$, $\neg A$, $A \Rightarrow B, \ldots$) as the types of all their possible proofs.

In particular: A false proposition has no proofs, so it corresponds to an empty type.

What is conjunction $A \wedge B$?

What do we know about the proposition $A \wedge B$ (A and B)?

- To prove $A \wedge B$, we need to provide a proof of A and a proof of B.
- Given a proof of $A \wedge B$, we can get proofs of A and B

What is conjunction $A \wedge B$?

What do we know about the proposition $A \wedge B$ (A and B)?

- To prove $A \wedge B$, we need to provide a proof of A and a proof of B.
- Given a proof of $A \wedge B$, we can get proofs of A and B

$\Rightarrow$ The type of proofs of $A \wedge B$ is the type of pairs $A \times B$

What is implication $A \Rightarrow B$?

What do we know about the proposition $A \Rightarrow B$ (A implies B)?

- To prove $A \Rightarrow B$, we can assume we have a proof of A and have to provide a proof of B
- From a proof of $A \Rightarrow B$ and a proof of A, we can get a proof of B

What is implication $A \Rightarrow B$?

What do we know about the proposition $A \Rightarrow B$ (A implies B)?

- To prove $A \Rightarrow B$, we can assume we have a proof of A and have to provide a proof of B
- From a proof of $A \Rightarrow B$ and a proof of A, we can get a proof of B

$\Rightarrow$ The type of proofs of $A \Rightarrow B$ is the function type $A \to B$

Proof by implication (Modus ponens)

Modus ponens says that if P implies Q and P is true, then Q is true.

Question. How can we prove this in Agda?

Proof by implication (Modus ponens)

Modus ponens says that if P implies Q and P is true, then Q is true.

Question. How can we prove this in Agda?

Answer.

```
modusPonens : {P Q : Set} → (P → Q) × P → Q
modusPonens (f , x) = f x
```

What is disjunction $A \vee B$?

What do we know about the proposition $A \vee B$ (A or B)?

- To prove $A \vee B$ we need to provide a proof of A or a proof of B.
- If we have:
    - a proof of $A \vee B$
    - a proof of C assuming a proof of A
    - a proof of C assuming a proof of B

  then we have a proof of C.

What is disjunction $A \vee B$?

What do we know about the proposition $A \vee B$ (A or B)?

- To prove $A \vee B$ we need to provide a proof of A or a proof of B.
- If we have:
  - a proof of $A \vee B$
  - a proof of C assuming a proof of A
  - a proof of C assuming a proof of B

  then we have a proof of C.

$\Rightarrow$ The type of proofs of $A \vee B$ is the sum type Either A B

Proof by cases

Proof by cases says that if $P \vee Q$ is true and we can prove R from P and also prove R from Q, then we can prove R.

Question. How can we prove this in Agda?

Proof by cases

Proof by cases says that if $P \vee Q$ is true and we can prove R from P and also prove R from Q, then we can prove R.

Question. How can we prove this in Agda?

Answer.

```
cases : {P Q R : Set}
        → Either P Q → (P → R) × (Q → R) → R
cases (left x)  (f , g) = f x
cases (right y) (f , g) = g y
```

Quiz question

Question. Which Agda type represents the proposition *"If (P implies Q) then (P or R) implies (Q or R)"*?

1. $(\text{Either}\ P\ Q) \to \text{Either}\ (P \to R)\ (Q \to R)$
2. $(P \to Q) \to \text{Either}\ P\ R \to \text{Either}\ Q\ R$
3. $(P \to Q) \to \text{Either}\ (P \times R)\ (Q \times R)$
4. $(P \times Q) \to \text{Either}\ P\ R \to \text{Either}\ Q\ R$

What is truth?

What do we know about the proposition ‘true’?

- To prove ‘true’, we don’t need to provide anything
- From ‘true’, we can deduce nothing

What is truth?

What do we know about the proposition ‘true’?

- To prove ‘true’, we don’t need to provide anything
- From ‘true’, we can deduce nothing

$\Rightarrow$ The type of proofs of truth is the *unit type* $\top$ with one constructor tt:

```
data ⊤ : Set where
  tt : ⊤
```

What is falsity?

What do we know about the proposition ‘false’?

- There is no way to prove ‘false’
- From a proof t of ‘false’, we get a proof `absurd` t of any proposition A

What is falsity?

What do we know about the proposition ‘false’?

- There is no way to prove ‘false’
- From a proof t of ‘false’, we get a proof `absurd` t of any proposition A

$\Rightarrow$ The type of proofs of falsity is the empty type $\bot$ with no constructors:

data $\bot$: Set where

Principle of explosion

The principle of explosion[1] says that if we assume a false statement, we can prove any proposition P.

Question. How can we prove this in Agda?

[1] Also known as *ex falso quodlibet* = *from falsity follows anything*.

Principle of explosion

The principle of explosion[1] says that if we assume a false statement, we can prove any proposition P.

Question. How can we prove this in Agda?

Answer.

$$\mathsf{absurd} : \{P : \mathsf{Set}\} \to \bot \to P$$
$$\mathsf{absurd}\ ()$$

[1]Also known as *ex falso quodlibet* = *from falsity follows anything.*

Curry-Howard for propositional logic

We can translate from the language of logic to the language of types according to this table:

Propositional logic		Type system
proposition	P	type
proof of a proposition	$p : P$	program of a type
conjunction	$P \times Q$	pair type
disjunction	$\text{Either}\ P\ Q$	either type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	empty type

Derived notions

Negation. We can encode $\neg P$ (“not P”) as the type $P \to \bot$.

Equivalence. We can encode $P \Leftrightarrow Q$ (“P is equivalent to Q”) as $(P \to Q) \times (Q \to P)$.

An exercise in translation

Exercise. Translate the following statements to types in Agda, and prove them by constructing a program of that type:

1. If *P* implies *Q* and *Q* implies *R*, then *P* implies *R*
2. If *P* is false and *Q* is false, then (either *P* or *Q*) is false.
3. If *P* is both true and false, then any proposition *Q* is true.

Three layers of Curry-Howard

1. Propositions are *types*

Three layers of Curry-Howard

1. Propositions are *types*
2. Proofs are *programs*

Three layers of Curry-Howard

1. Propositions are *types*
2. Proofs are *programs*
3. Simplifying a proof is *evaluating* a program

Three layers of Curry-Howard

1. Propositions are *types*
2. Proofs are *programs*
3. Simplifying a proof is *evaluating* a program

Example. An indirect proof of $A \to A$ evaluates to direct proof:

$$
\begin{aligned}
&\lambda\, x \to (\lambda\, y \to \mathsf{fst}\; y)\,(x, x) \\
&\quad\longrightarrow \quad \lambda\, x \to \mathsf{fst}\,(x, x) \\
&\quad\longrightarrow \quad \lambda\, x \to x
\end{aligned}
$$

Classical vs. constructive logic

Non-constructive statements

In classical logic we can prove certain 'non-constructive' statements:

- $P \vee (\neg P)$ (excluded middle)
- $\neg\neg P \Rightarrow P$ (double negation elimination)

However, Agda uses a constructive logic: a proof of $A \vee B$ gives us a decision procedure to tell whether A or B holds.

When P is unknown, it's impossible to decide whether P or $\neg P$ holds, so the excluded middle is unprovable in Agda.

From classical to constructive logic

Consider the proposition P ("P is true") vs. $\neg\neg P$ ("It would be absurd if P were false").

Classical logic can't tell the difference between the two, but constructive logic can.

Theorem. P is provable in classical logic if and only if $\neg\neg P$ is provable in constructive logic. (proof by Gödel and Gentzen)

Predicate logic

Curry-Howard beyond simple types

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Curry-Howard beyond simple types

- Classical logic corresponds to continuations (e.g. Lisp)

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Curry-Howard beyond simple types

- Classical logic corresponds to continuations (e.g. Lisp)
- Linear logic corresponds to linear types (e.g. Rust)

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Curry-Howard beyond simple types

- Classical logic corresponds to continuations (e.g. Lisp)
- Linear logic corresponds to linear types (e.g. Rust)
- Predicate logic corresponds to dependent types (e.g. Agda)

"Every good idea will be discovered twice: once by a logician and once by a computer scientist." – Philip Wadler

Proving things about programs

So far, we have encoded logical propositions as types and proofs as programs of these types, but there is no interaction yet between the 'program part' and the 'proof part' of Agda.

Question. Can we use the 'proof part' to prove things about the 'program part'?

Proving things about programs

So far, we have encoded logical propositions as types and proofs as programs of these types, but there is no interaction yet between the 'program part' and the 'proof part' of Agda.

Question. Can we use the 'proof part' to prove things about the 'program part'?

Answer. Yes, we can define propositions that depend on (the output of) programs by using dependent types!

Defining predicates

Question. How would you define a type that expresses that a given number n is even?

Defining predicates

Question. How would you define a type that expresses that a given number n is even?

```
data IsEven : Nat → Set where
  e-zero : IsEven zero
  e-suc2 : {n : Nat} →
    IsEven n → IsEven (suc (suc n))

6-is-even : IsEven 6
6-is-even = e-suc2 (e-suc2 (e-suc2 e-zero))

7-is-not-even : IsEven 7 → ⊥
7-is-not-even (e-suc2 (e-suc2 (e-suc2 ())))
```

Defining predicates

To define a predicate P on elements of type A, we can define P as a dependent datatype with base type A:

```
data P : A → Set where
  c₁ : ⋯ → P a₁
  c₂ : ⋯ → P a₂
  -  . . .
```

A predicate for being true

We can define a predicate IsTrue that allows us to use boolean functions as predicates.

```
data IsTrue : Bool → Set where
  is-true : IsTrue true
```

- If b = true, then IsTrue b has exactly one element is-true
- If b = false, then IsTrue b has no elements: it is an empty type

Using the IsTrue predicate

```
_=Nat_ : Nat → Nat → Bool
zero    =Nat zero    = true
(suc x) =Nat (suc y) = x =Nat y
_       =Nat _       = false

length-is-3 : IsTrue (length (1 :: 2 :: 3 :: []) =Nat 3)
length-is-3 = is-true
```

Universal quantification

What do we know about the proposition $\forall (x \in A).\, P(x)$ ('for all x in A, $P(x)$ holds')?

- To prove $\forall (x \in A).\, P(x)$, we assume we have an unknown $x \in A$ and prove that $P(x)$ holds.
- If we have a proof of $\forall (x \in A).\, P(x)$ and a concrete $a \in A$, then we know $P(a)$.

Universal quantification

What do we know about the proposition $\forall(x \in A).\ P(x)$ ('for all x in A, $P(x)$ holds')?

- To prove $\forall(x \in A).\ P(x)$, we assume we have an unknown $x \in A$ and prove that $P(x)$ holds.
- If we have a proof of $\forall(x \in A).\ P(x)$ and a concrete $a \in A$, then we know $P(a)$.

$\Rightarrow$ $\forall(x \in A).\ P(x)$ corresponds to the dependent function type $(x : A) \to P\ x$.

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

```
double : Nat → Nat
double zero    = zero
double (suc m) = suc (suc (double m))
```

```
double-even : (n : Nat) → IsEven (double n)
double-even n = {!  !}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

```
double : Nat → Nat
double zero    = zero
double (suc m) = suc (suc (double m))

double-even : (n : Nat) → IsEven (double n)
double-even zero    = {!  !}
double-even (suc m) = {!  !}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

```
double : Nat → Nat
double zero    = zero
double (suc m) = suc (suc (double m))

double-even : (n : Nat) → IsEven (double n)
double-even zero    = e-zero
double-even (suc m) = {!  !}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

```
double : Nat → Nat
double zero    = zero
double (suc m) = suc (suc (double m))

double-even : (n : Nat) → IsEven (double n)
double-even zero    = e-zero
double-even (suc m) = e-suc2 {!  !}
```

Universal quantification

Example. We can state and prove that for any number n : Nat, double n is even:

$$\begin{aligned}
&\mathsf{double} : \mathsf{Nat} \to \mathsf{Nat} \\
&\mathsf{double}\ \mathsf{zero} \qquad\ \ = \mathsf{zero} \\
&\mathsf{double}\ (\mathsf{suc}\ m) = \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{double}\ m))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{double\text{-}even} : (n : \mathsf{Nat}) \to \mathsf{IsEven}\ (\mathsf{double}\ n) \\
&\mathsf{double\text{-}even}\ \mathsf{zero} \qquad\ \ = \mathsf{e\text{-}zero} \\
&\mathsf{double\text{-}even}\ (\mathsf{suc}\ m) = \mathsf{e\text{-}suc2}\ (\mathsf{double\text{-}even}\ m)
\end{aligned}$$

Existential quantification

Existential quantification

What do we know about existential quantification $\exists(x \in A).\ P(x)$ ("there exists $x \in A$ such that $P(x)$")?

- To prove $\exists(x \in A).\ P(x)$, we need to provide some $v \in A$ and a proof of $P(v)$.
- From a proof of $\exists(x \in A).\ P(x)$, we can get some $v \in A$ and a proof of $P(v)$.

$\Rightarrow$ The proposition $\exists(x \in A).\ P(x)$ corresponds to the type of pairs (v, p) *where the type of p depends on the value of v.*

Dependent pairs

The type Σ[2] is defined as follows:

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

Projections from a dependent pair:

```
fstΣ : {A : Set}{B : A → Set} → Σ A B → A
fstΣ (x , y) = x

sndΣ : {A : Set}{B : A → Set} →
  (z : Σ A B) → B (fstΣ z)
sndΣ (x , y) = y
```

[2]Write `\Sigma` to enter.

Proving an existential statement

Example. Prove that there exists a number *n* such that $n + n = 12$:

```
half-a-dozen : Σ Nat (λ n → IsTrue ((n + n) =Nat 12))
half-a-dozen = 6 , is-true
```

Here the second component is-true has type IsTrue ((6 + 6) =Nat 12).

Induction in Agda

In general, a proof by induction on natural numbers in Agda looks like this:

$$\begin{aligned}
&\mathsf{proof} : (n : \mathsf{Nat}) \to \mathsf{P}\ n\\
&\mathsf{proof}\ \mathsf{zero} \qquad\ \ = \cdots\\
&\mathsf{proof}\ (\mathsf{suc}\ n) = \cdots
\end{aligned}$$

- $\mathsf{proof}\ \mathsf{zero}$ is the base case
- $\mathsf{proof}\ (\mathsf{suc}\ n)$ is the inductive case

When proving the inductive case, we can make use of the induction hypothesis $\mathsf{proof}\ n : P\ n$.

An example: *n* is equal to itself

Let's prove that any number is equal to itself:

n-equals-n : (n : Nat) $\rightarrow$ IsTrue (n =Nat n)
n-equals-n n = is-true

This code results in an error:

true != n =Nat n of type Bool

Question. What did we do wrong?

An example: n is equal to itself

Answer. Since _=Nat_ is defined by pattern matching and recursion, the proof must do the same:

$$
\begin{aligned}
&\text{n-equals-n} : (n : \text{Nat}) \to \text{IsTrue}\ (n \mathbin{=\text{Nat}} n)\\
&\text{n-equals-n zero} \quad\quad = \text{is-true}\\
&\text{n-equals-n}\ (\text{suc}\ m) = \text{n-equals-n}\ m
\end{aligned}
$$

Proving things about programs

General rule of thumb: A proof about a function often follows the same structure as that function:

- To prove something about a function by pattern matching, the proof will also use pattern matching (= proof by cases)
- To prove something about a recursive function, the proof will also be recursive (= proof by induction)

On the need for totality

To ensure the proofs we write are correct, we rely on the totality of Agda:

- The coverage checker ensures that a proof by cases covers all cases.
- The termination checker ensures that inductive proofs are well-founded.

Induction on lists in Agda

In general, a proof by induction on lists in Agda looks like this:

$$\begin{aligned}
&\mathsf{proof} : \{A : \mathsf{Set}\}\ (xs : \mathsf{List}\ A) \to \mathsf{P}\ xs \\
&\mathsf{proof}\ [\,] \qquad\quad\ = \cdots \\
&\mathsf{proof}\ (x :: xs) = \cdots
\end{aligned}$$

- $\mathsf{proof}\ [\,]$ is the base case
- $\mathsf{proof}\ (x :: xs)$ is the inductive case

The inductive case can use the induction hypothesis $\mathsf{proof}\ xs : \mathsf{P}\ xs$.

Live programming exercise

Assignment. Write down the Agda type expressing the statement that for any function f and list xs, length (map f xs) is equal to length xs.

Then, prove it by implementing a function of that type.

What's next?

Next lecture: Equational reasoning about functional programs

To do:

- Read the lecture notes:
  - This lecture: section 3 of Agda lecture notes
  - Next lecture: section 4 of Agda lecture notes
- Do Weblab exercises on Curry-Howard