Equational Reasoning about Functional Programs

Lecture 12 of CSE 3100
Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

– Donald Knuth

Lecture plan

- The identity type
- Equational reasoning in Agda
- Applications of equational reasoning:
  - Proving type class laws
  - Verifying optimizations
  - Verifying compiler correctness

Recap: the Curry-Howard correspondence

Haskell B. Curry

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

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

Recap: the Curry-Howard correspondence

We interpret propositions as the types of their proofs:

Propositional logic		**Type system**
proposition	P	type
proof of a proposition	$p : P$	program of a type
conjunction	$P \times Q$	pair type
disjunction	$\mathsf{Either}\ P\ Q$	either type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	empty type
universal quantification	$(x : A) \to P\ x$	dependent function type
existential quantification	$\Sigma\ A\ (\lambda\ x \to P\ x)$	dependent pair type

The identity type

The identity type

The type IsTrue encodes the property of being equal to true : Bool:

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

We can generalize this to the property of two elements of some type A being equal:

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

Using the identity type

If x and y are equal, $x \equiv y$ has one constructor refl:

```
one-plus-one : 1 + 1 ≡ 2
one-plus-one = refl
```

If x and y are not equal, $x \equiv y$ is an empty type:

```
zero-not-one : 0 ≡ 1 → ⊥
zero-not-one ()
```

Application of the identity type: Writing test cases

One use case of the identity type is for writing test cases:

$$\mathsf{test}_1 : \mathsf{length}\ (42 :: [\,]) \equiv 1$$
$$\mathsf{test}_1 = \mathsf{refl}$$

$$\mathsf{test}_2 : \mathsf{length}\ (\mathsf{map}\ (1 +\_)\ (0 :: 1 :: 2 :: [\,])) \equiv 3$$
$$\mathsf{test}_2 = \mathsf{refl}$$

The test cases are run each time the file is loaded!

Proving correctness of functions

We can use the identity type to prove the correctness of functional programs.

Example. Prove that $\mathsf{not}\ (\mathsf{not}\ b) \equiv b$ for all $b : \mathsf{Bool}$:

$$
\begin{aligned}
&\mathsf{not\text{-}not} : (b : \mathsf{Bool}) \to \mathsf{not}\ (\mathsf{not}\ b) \equiv b \\
&\mathsf{not\text{-}not}\ \mathsf{true}\ \ = \mathsf{refl} \\
&\mathsf{not\text{-}not}\ \mathsf{false} = \mathsf{refl}
\end{aligned}
$$

Quiz question

Question. What is the type of the Agda expression $\lambda\ b \to (b \equiv \mathsf{true})$?

1. $\mathsf{Bool} \to \mathsf{Bool}$
2. $\mathsf{Bool} \to \mathsf{Set}$
3. $(b : \mathsf{Bool}) \to \mathsf{IsTrue}\ b$
4. $(b : \mathsf{Bool}) \to b \equiv \mathsf{true}$

Pattern matching on refl

If we have a proof of $x \equiv y$ as input, we can pattern match on the constructor refl to show Agda that x and y are equal:

```
castVec : {A : Set} {m n : Nat} →
  m ≡ n → Vec A m → Vec A n
castVec refl xs = xs
```

When you pattern match on refl, Agda applies unification to the two sides of the equality.

Symmetry of equality

Symmetry states that if x is equal to y, then y is equal to x:

$$\mathsf{sym} : \{A : \mathsf{Set}\}\ \{x\ y : A\} \to x \equiv y \to y \equiv x$$
$$\mathsf{sym}\ \mathsf{refl} = \mathsf{refl}$$

Congruence

Congruence states that if $\mathsf{f} : A \to B$ is a function and x is equal to y, then $\mathsf{f}\ x$ is equal to $\mathsf{f}\ y$:

$$
\begin{aligned}
&\mathsf{cong} : \{A\ B : \mathsf{Set}\}\ \{x\ y : A\} \to \\
&\quad (f : A \to B) \to x \equiv y \to f\,x \equiv f\,y \\
&\mathsf{cong}\ f\ \mathsf{refl} = \mathsf{refl}
\end{aligned}
$$

Equational reasoning

Equational reasoning

In school, we learned how to prove equations by chaining basic equalities:

```
   (a + b) (a + b)
= a (a + b) + b (a + b)
= a^2 + ab + ba + b^2
= a^2 + ab + ab + b^2
= a^2 + 2ab + b^2
```

This style of proving is called equational reasoning.

Equational reasoning about functional programs

Equational reasoning is well suited for proving things about pure functions:

```
  head (replicate 100 "spam")
= head ("spam" : replicate 99 "spam")
= "spam"
```

Because there are no side effects, everything is explicit in the program itself.

Equational reasoning in Agda

Consider the following definitions:

$$
\begin{aligned}
&[\_] : \{A : \mathsf{Set}\} \to A \to \mathsf{List}\ A \\
&[\ x\ ] = x :: []
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reverse} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{List}\ A \\
&\mathsf{reverse}\ [] = [] \\
&\mathsf{reverse}\ (x :: xs) = \mathsf{reverse}\ xs +\!+ [\ x\ ]
\end{aligned}
$$

Goal. Prove that $\mathsf{reverse}\ [\ x\ ] = [\ x\ ]$.

Example ‘on paper’

```
  reverse [ x ]
=     { definition of [_] }
  reverse (x :: [])
=     { applying reverse (second clause) }
  reverse [] ++ [ x ]
=     { applying reverse (first clause) }
  [] ++ [ x ]
=     { applying _++_ }
  [ x ]
```

Example in Agda

```
reverse-singleton : {A : Set} (x : A) → reverse [ x ] ≡ [ x ]
reverse-singleton x =
  begin
    reverse [ x ]
  =⟨⟩ -  definition of [_]
    reverse (x :: [])
  =⟨⟩ -  applying reverse (second clause)
    reverse [] ++ [ x ]
  =⟨⟩ -  applying reverse (first clause)
    [] ++ [ x ]
  =⟨⟩ -  applying _++_
    [ x ]
  end
```

Equational reasoning in Agda

We can write down an equality proof in equational reasoning style in Agda:

- The proof starts with begin and ends with end.
- In between is a sequence of expressions separated by $=\langle\rangle$, where each expression is equal to the previous one.

Unlike the proof on paper, here the typechecker of Agda guarantees that each step of the proof is correct!

Behind the scenes

Each proof by equational reasoning can be desugared to refl (and trans).

Example.

```
reverse-singleton : {A : Set} (x : A) →
  reverse [ x ] ≡ [ x ]
reverse-singleton x = refl
```

However, proofs by equational reasoning are much easier to read and debug.

Proof by case analysis and induction

Equational reasoning + case analysis

We can use equational reasoning in a proof by case analysis (i.e. pattern matching):

```
not-not : (b : Bool) → not (not b) ≡ b
not-not false =
  begin
    not (not false)
  =⟨⟩                      -- applying the inner not
    not true
  =⟨⟩                      -- applying not
    false
  end
not-not true = {!!} -- similar to above
```

Equational reasoning + induction

We can use equational reasoning in a proof by induction:

```
add-n-zero : (n : Nat) → n + zero ≡ n
add-n-zero zero     = {!!}          -- easy exercise
add-n-zero (suc n) =
  begin
    (suc n) + zero
  =⟨⟩                               -- applying +
    suc (n + zero)
  =⟨ cong suc (add-n-zero n) ⟩      -- using IH
    suc n
  end
```

Here we have to provide an explicit proof that suc (n + zero) = suc n (between the =⟨ and ⟩).

Live coding (live proving?)

Exercise. State and prove associativity of addition on natural numbers:
$x + (y + z) = (x + y) + z$

Hint. If you get stuck, try to work instead backwards from the goal you want to reach!

Application 1: Proving type class laws

Reminder: functor laws

Remember the two functor laws from Haskell:

- fmap id = id
- fmap $(f\ .\ g)$ = fmap f . fmap g

In Haskell we could only verify these laws by hand for each instance, but in Agda we can prove that they hold.

First functor law for List (base case)

```
map-id : {A : Set} (xs : List A) → map id xs ≡ xs
map-id [] =
  begin
    map id []
  =⟨⟩ -- applying map
    []
  end
```

First functor law for List (inductive case)

```
map-id (x :: xs) =
  begin
    map id (x :: xs)
  =⟨⟩                                   -- applying map
    id x :: map id xs
  =⟨⟩                                   -- applying id
    x :: map id xs
  =⟨ cong (x ::_) (map-id xs) ⟩         -- using IH
    x :: xs
  end
```

More live proving

Exercise. Prove the second functor law for List.

First, we need to define function composition:[1]

$$
\begin{aligned}
&\_\circ\_ : \{A\ B\ C : \mathsf{Set}\} \rightarrow \\
&\quad (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) \\
&f \circ g = \lambda\ x \rightarrow f\ (g\ x)
\end{aligned}
$$

Now we can prove that
$\mathsf{map}\ (f \circ g)\ x = (\mathsf{map}\ f \circ \mathsf{map}\ g)\ x$.

[1] Unicode input for $\circ$: `\circ`

Application 2: Verifying optimizations

Reminder: working with accumulators

A slow version of reverse in $O(n^2)$:

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

A faster version of reverse in $O(n)$:

```
reverse-acc : {A : Set} → List A → List A → List A
reverse-acc [] ys = ys
reverse-acc (x :: xs) ys = reverse-acc xs (x :: ys)
```

```
reverse' : {A : Set} → List A → List A
reverse' xs = reverse-acc xs []
```

How can we be sure they are equivalent? By proving it!

Equivalence of reverse and reverse'

```
reverse'-reverse : {A : Set} →
  (xs : List A) → reverse' xs ≡ reverse xs
reverse'-reverse xs =
  begin
    reverse' xs
  =⟨⟩                              -- def of reverse'
    reverse-acc xs []
  =⟨ reverse-acc-lemma xs [] ⟩ -- (see next slide)
    reverse xs ++ []
  =⟨ append-[] (reverse xs) ⟩  -- using append-[]
    reverse xs
  end
```

Proving the lemma (base case)

```
reverse-acc-lemma : {A : Set} → (xs ys : List A)
  → reverse-acc xs ys ≡ reverse xs ++ ys
reverse-acc-lemma [] ys =
  begin
    reverse-acc [] ys
  =⟨⟩ -- definition of reverse-acc
    ys
  =⟨⟩ -- unapplying ++
    [] ++ ys
  =⟨⟩ -- unapplying reverse
    reverse [] ++ ys
  end
```

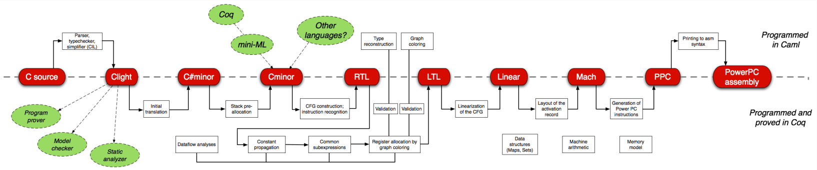
Proving the lemma (inductive case)

```
reverse-acc-lemma (x :: xs) ys =
  begin
    reverse-acc (x :: xs) ys
  =⟨⟩                                 -- def of reverse-acc
    reverse-acc xs (x :: ys)
  =⟨ reverse-acc-lemma xs (x :: ys) ⟩
    reverse xs ++ (x :: ys)           -- ^ using IH
  =⟨⟩                                 -- unapplying ++
    reverse xs ++ ([ x ] ++ ys)
  =⟨ sym (append-assoc (reverse xs) [ x ] ys) ⟩
    (reverse xs ++ [ x ]) ++ ys       -- ^ associativity of ++
  =⟨⟩                                 -- unapplying reverse
    reverse (x :: xs) ++ ys
  end
```

Application 3: Proving compiler correctness

Real-world application: The CompCert C compiler

CompCert is an optimizing compiler for C code, which is formally proven to be correct according to the semantics of the C language, using the dependently typed language Coq.



To learn more: `https://compcert.org/`

A simple expression language

```
data Expr : Set where
  valE  : Nat → Expr
  addE : Expr → Expr → Expr

-- Example expr: (2 + 3) + 4
expr : Expr
expr = addE (addE (valE 2) (valE 3)) (valE 4)

eval : Expr → Nat
eval (valE x)       = x
eval (addE e1 e2) = eval e1 + eval e2
```

Evaluating expressions using a stack

```
data Op : Set where
  PUSH : Nat → Op
  ADD  : Op

Stack  = List Nat
Code   = List Op

-- Example code for (2 + 3) + 4
code : Code
code = PUSH 2 :: PUSH 3 :: ADD
       :: PUSH 4 :: ADD :: []
```

Executing compiled code

Given a list of instructions and an initial stack, we can execute the code:

$$
\begin{aligned}
&\mathrm{exec} : \mathrm{Code} \to \mathrm{Stack} \to \mathrm{Stack}\\
&\mathrm{exec}\ []\ \ s = s\\
&\mathrm{exec}\ (\mathrm{PUSH}\ x :: c)\ s = \mathrm{exec}\ c\ (x :: s)\\
&\mathrm{exec}\ (\mathrm{ADD} :: c)\ (m :: n :: s) = \mathrm{exec}\ c\ (n + m :: s)\\
&\mathrm{exec}\ (\mathrm{ADD} :: c)\ \_ = []
\end{aligned}
$$

Compiling expressions

Goal. Compile an expression to a list of stack instructions.

A first attempt.

```
comp : Expr → Code
comp (valE x)        = [ PUSH x ]
comp (addE e1 e2) =
  comp e1 ++ comp e2 ++ [ ADD ]
```

Problem. This is very inefficient ($O(n^2)$) due to the repeated use of _++_!

Compiling with an accumulator

Problem. This is very inefficient ($O(n^2)$) due to the repeated use of `_++_`!

Instead, we can use an accumulator for the already generated code:

```
comp' : Expr → Code → Code
comp' (valE x)       c = PUSH x :: c
comp' (addE e1 e2) c =
  comp' e1 (comp' e2 (ADD :: c))
```

```
comp : Expr → Code
comp e = comp' e []
```

Proving correctness of comp

We want to prove that executing the compiled code has the same result as evaluating the expression directly:

```
comp-exec-eval : (e : Expr) → exec (comp e) [] ≡ [ eval e ]
comp-exec-eval e =
  begin
    exec (comp e) []
  =⟨ comp'-exec-eval e [] [] ⟩ -- (see next slide)
    exec [] (eval e :: [])
  =⟨⟩                          -- applying exec for []
    eval e :: []
  =⟨⟩                          -- unapplying [_]
    [ eval e ]
  end
```

Proving correctness of comp' (valE case)

```
comp'-exec-eval : (e : Expr) (s : Stack) (c : Code)
  → exec (comp' e c) s ≡ exec c (eval e :: s)
comp'-exec-eval (valE x) s c =
  begin
    exec (comp' (valE x) c) s
  =⟨⟩ -- applying comp'
    exec (PUSH x :: c) s
  =⟨⟩ -- applying exec for PUSH
    exec c (x :: s)
  =⟨⟩ -- unapplying eval for valE
    exec c (eval (valE x) :: s)
  end
```

Proving correctness of comp' (addE case)

```
comp'-exec-eval (addE e1 e2) s c =
  begin
    exec (comp' (addE e1 e2) c) s
  =⟨⟩ -- def of comp'
    exec (comp' e1 (comp' e2 (ADD :: c))) s
  =⟨ comp'-exec-eval e1 s (comp' e2 (ADD :: c)) ⟩ -- IH
    exec (comp' e2 (ADD :: c)) (eval e1 :: s)
  =⟨ comp'-exec-eval e2 (eval e1 :: s) (ADD :: c) ⟩ -- IH
    exec (ADD :: c) (eval e2 :: eval e1 :: s)
  =⟨⟩ -- applying exec for ADD
    exec c (eval e1 + eval e2 :: s)
  =⟨⟩ -- unapplying eval for addE
    exec c (eval (addE e1 e2) :: s)
  end
```

Equational reasoning: summary

Equational reasoning is a simple but powerful technique to reason about pure functional programs.

We can write Agda proofs in equational reasoning style by using the combinators begin, end, _≡⟨⟩_, and _≡⟨_⟩_.

Equational reasoning combines well with case analysis (= pattern matching) and induction (= recursion).

What's next?

Final lecture: Course recap + preparation for exam

To do:

- Read the lecture notes:
    - This lecture: section 4 of Agda lecture notes
- Do exercises on equational reasoning in Weblab
- Send me topics or questions for the final lecture!