Programming Languages

Equational reasoning
Structural induction

Introduction

Structural induction

Extensionality

Type isomorphisms

Case studies

Motivation

We want to prove that certain expressions are equivalent.
What for?

To justify that an algorithm is correct

For example, if we manage to prove that:

$$\forall \texttt{xs} :: \texttt{[Int]}.\ \texttt{quickSort xs = insertionSort xs}$$

this gives us relative confidence in one algorithm with respect to the other.

To enable optimizations

Is it always correct to make the following optimizations?

```
f x + f x            ⇝  2 * f x
map f (map g xs)     ⇝  map (f . g) xs
```

In a functional language, yes.
In an imperative language **no**, since `f` and `g` can have side effects.

Working hypotheses

To reason about equivalence of expressions, we will assume:

1. That we work with **finite** data structures.

   Technically: with **inductive** data types.

2. That we work with **total functions**.
   - Equations must cover all cases.
   - Recursion must always terminate.

3. That the program **does not depend on the order** of equations.

```
empty [] = True          ⇝    empty []      = True
empty _  = False              empty (_ : _) = False
```

Relaxing these hypotheses is possible but more complex.

Equalities by definition

Replacement principle

Let `e1 = e2` be an equation included in the program.
The following operations preserve equality of expressions:

1. Replace any instance of `e1` by `e2`.
2. Replace any instance of `e2` by `e1`.

If an equality can be proved using only the replacement principle, we say the equality holds **by definition**.

Example: replacement principle

We name the equations in the program:

```
      successor :: Int -> Int
{SUC} successor n = n + 1
```

```
    successor (factorial 10) + 1
  = (factorial 10 + 1) + 1          by SUC
  = successor (factorial 10 + 1)    by SUC
```

Equalities by definition

Example: replacement principle

```
{L0}  length []       = 0
{L1}  length (_ : xs) = 1 + length xs
{S0}  sum []       = 0
{S1}  sum (x : xs) = x + sum xs
```

Let's see that `length ["a", "b"] = sum [1, 1]`:

```
    length ["a", "b"]
=   1 + length ["b"]         by L1
=   1 + (1 + length [])      by L1
=   1 + (1 + 0)              by L0
=   1 + (1 + sum [])         by S0
=   1 + sum [1]              by S1
=   sum [1, 1]               by S1
```

Introduction

Structural induction

Extensionality

Type isomorphisms

Case studies

Induction on booleans

The replacement principle is not enough to prove all the equivalences we are interested in.

Example

```
{NT} not True  = False
{NF} not False = True
```

Can we prove `∀x :: Bool. not (not x) = x`?

The problem is that the expression

```
not (not x)
```

is "stuck": no equation can be applied.

Induction on booleans

Induction principle on booleans

If $\mathcal{P}(\texttt{True})$ and $\mathcal{P}(\texttt{False})$ then $\forall \texttt{x} :: \texttt{Bool}.\ \mathcal{P}(\texttt{x})$.

Example

```
{NT} not True  = False
{NF} not False = True
```

To prove `∀x :: Bool. not (not x) = x`
it suffices to prove:

1. `not (not True) = True`.

```
not (not True) = not False = True
               ↑           ↑
               NT          NF
```

2. `not (not False) = False`.

```
not (not False) = not True = False
                ↑          ↑
                NF         NT
```

Induction on pairs

Each data type has its own induction principle.

Example

```
{FST}  fst (x, _)  = x
{SND}  snd (_, y)  = y
{SWAP} swap (x, y) = (y, x)
```

Can we prove `∀p :: (a, b). fst p = snd (swap p)`?

The expressions `(fst p)` and `(snd (swap p))` are "stuck".

Induction on pairs

Induction principle on pairs

If ∀x :: a. ∀y :: b. $\mathcal{P}$((x, y))
then ∀p :: (a, b). $\mathcal{P}$(p).

Example

```
{FST}  fst (x, _)  = x
{SND}  snd (_, y)  = y
{SWAP} swap (x, y) = (y, x)
```

To prove ∀p :: (a, b). fst p = snd (swap p)
it suffices to prove:

- ∀x :: a. ∀y :: b. fst (x, y) = snd (swap (x, y))

```
fst (x, y)  =  x  =  snd (y, x)  =  snd (swap (x, y))
            ↑     ↑              ↑
           FST   SND            SWAP
```

Induction on naturals

```
data Nat = Zero | Succ Nat
```

Induction principle on naturals

If $\mathcal{P}(\texttt{Zero})$ and $\forall \texttt{n} :: \texttt{Nat}.\ (\underbrace{\mathcal{P}(\texttt{n})}_{\text{inductive hypothesis}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Succ n})}_{\text{inductive thesis}})$,

then $\forall \texttt{n} :: \texttt{Nat}.\ \mathcal{P}(\texttt{n})$.

Induction on naturals

Example

```
{S0} sum Zero     m = m
{S1} sum (Succ n) m = Succ (sum n m)
```

To prove $\forall$`n :: Nat. sum n Zero = n`
it suffices to prove:

1. `sum Zero Zero = Zero`.
   Immediate by `S0`.
2. $\underbrace{\texttt{sum n Zero = n}}_{\text{I.H.}} \Rightarrow \underbrace{\texttt{sum (Succ n) Zero = Succ n}}_{\text{I.T.}}$.

$$\texttt{sum (Succ n) Zero} \underset{\texttt{S1}}{\overset{\uparrow}{=}} \texttt{Succ (sum n Zero)} \underset{\text{I.H.}}{\overset{\uparrow}{=}} \texttt{Succ n}$$

Structural induction

In the **general case**, we have an inductive data type:

$$
\begin{array}{lcl}
\texttt{data T} & = & \texttt{CBase}_1 \; \langle \textit{parameters} \rangle \\
 & & \texttt{...} \\
 & | & \texttt{CBase}_n \; \langle \textit{parameters} \rangle \\
 & | & \texttt{CRecursive}_1 \; \langle \textit{parameters} \rangle \\
 & & \texttt{...} \\
 & | & \texttt{CRecursive}_m \; \langle \textit{parameters} \rangle
\end{array}
$$

Structural induction principle

Let $\mathcal{P}$ be a property about expressions of type `T` such that:

- $\mathcal{P}$ holds for all base constructors of `T`,
- $\mathcal{P}$ holds for all recursive constructors of `T`, assuming as inductive hypothesis that it holds for the parameters of type `T`,

then $\forall \texttt{x} :: \texttt{T}.\ \mathcal{P}(\texttt{x})$.

Structural induction

Example: induction principle on lists

```
data [a] = [] | a : [a]
```

Let $\mathcal{P}$ be a property on expressions of type `[a]` such that:

- $\mathcal{P}(\texttt{[]})$
- $\forall \texttt{x} :: \texttt{a}.\ \forall \texttt{xs} :: \texttt{[a]}.\ (\underbrace{\mathcal{P}(\texttt{xs})}_{\text{I.H.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{x : xs})}_{\text{I.T.}})$

Then $\forall \texttt{xs} :: \texttt{[a]}.\ \mathcal{P}(\texttt{xs})$.

Example: induction principle on binary trees

```
data BT a = Nil | Bin (BT a) a (BT a)
```

Let $\mathcal{P}$ be a property on expressions of type `BT a` such that:

- $\mathcal{P}(\texttt{Nil})$
- $\forall \texttt{l} :: \texttt{BT a}.\ \forall \texttt{r} :: \texttt{a}.\ \forall \texttt{ri} :: \texttt{BT a}.$

  $$(\underbrace{(\mathcal{P}(\texttt{l}) \wedge \mathcal{P}(\texttt{ri}))}_{\text{I.H.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Bin l r ri})}_{\text{I.T.}})$$

Then $\forall \texttt{x} :: \texttt{BT a}.\ \mathcal{P}(\texttt{x})$.

Structural induction

Example: induction principle on polynomials

```
data Poly a = X
            | Const a
            | Sum (Poly a) (Poly a)
            | Prod (Poly a) (Poly a)
```

Let $\mathcal{P}$ be a property on expressions of type `Poly a` such that:

- $\mathcal{P}(\texttt{X})$
- $\forall \texttt{k} :: \text{a}.\ \mathcal{P}(\texttt{Const k})$
- $\forall \texttt{p} :: \texttt{Poly a}.\ \forall \texttt{q} :: \texttt{Poly a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{I.H.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Sum p q})}_{\text{I.T.}})$$
- $\forall \texttt{p} :: \texttt{Poly a}.\ \forall \texttt{q} :: \texttt{Poly a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{I.H.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Prod p q})}_{\text{I.T.}})$$

Then $\forall \texttt{x} :: \texttt{Poly a}.\ \mathcal{P}(\texttt{x})$.

Example: induction on lists

```
{M0} map f []       = []
{M1} map f (x : xs) = f x : map f xs
{A0} []       ++ ys = ys
{A1} (x : xs) ++ ys = x : (xs ++ ys)
```

Property. If `f :: a -> b`, `xs :: [a]`, `ys :: [a]`, then:

`map f (xs ++ ys) = map f xs ++ map f ys`

By induction on the structure of `xs`, it suffices to check:

1. Base case, $\mathcal{P}(\texttt{[]})$.
2. Inductive case, $\forall \texttt{x} :: \texttt{a}.\ \forall \texttt{xs} :: \texttt{[a]}.\ (\mathcal{P}(\texttt{xs}) \Rightarrow \mathcal{P}(\texttt{x : xs}))$.

with $\mathcal{P}(\texttt{xs}) :\equiv (\texttt{map f (xs ++ ys) = map f xs ++ map f ys})$.

Example: induction on lists

Base case:

```
   map f ([] ++ ys)
=  map f ys                 by A0
=  [] ++ map f ys           by A0
=  map f [] ++ map f ys     by M0
```

Inductive case:

```
   map f ((x : xs) ++ ys)
=  map f (x : (xs ++ ys))           by A1
=  f x : map f (xs ++ ys)           by M1
=  f x : (map f xs ++ map f ys)     by I.H.
=  (f x : map f xs) ++ map f ys     by A1
=  map f (x : xs) ++ map f ys       by M1
```

Example: relationship between foldr and foldl

Property. If `f :: a -> b -> b`, `z :: b`, `xs :: [a]`, then:

$$\underbrace{\texttt{foldr f z xs = foldl (flip f) z (reverse xs)}}_{\mathcal{P}(\texttt{xs})}$$

By induction on the structure of `xs`. The base case $\mathcal{P}(\texttt{[]})$ is easy.
Inductive case, $\forall \texttt{x :: a}.\ \forall \texttt{xs :: [a]}.\ (\mathcal{P}(\texttt{xs}) \Rightarrow \mathcal{P}(\texttt{x : xs}))$:

```
    foldr f z (x : xs)
 =  f x (foldr f z xs)                          (Def. foldr)
 =  f x (foldl (flip f) z (reverse xs))         (I.H.)
 =  flip f (foldl (flip f) z (reverse xs)) x    (Def. flip)
 =  foldl (flip f) z (reverse xs ++ [x])        (???)
 =  foldl (flip f) z (reverse (x : xs))         (Def. reverse)
```

To justify the missing step **(???)**, we can prove:

Lemma. If `g :: b -> a -> b`, `z :: b`, `x :: a`, `xs :: [a]`, then:

```
foldl g z (xs ++ [x]) = g (foldl g z xs) x
```

Generation lemmas

Using the structural induction principle, we can prove:

Generation lemma for pairs

If `p` :: `(a, b)`, then ∃`x` :: `a`. ∃`y` :: `b`. `p = (x, y)`.

```
data Either a b = Left a | Right b
```

Generation lemma for sums

If `e` :: `Either a b`, then:

- either ∃`x` :: `a`. `e = Left x`
- or ∃`y` :: `b`. `e = Right y`

Introduction

Structural induction

Extensionality

Type isomorphisms

Case studies

Intensional vs. extensional points of view

Does the following equivalence of expressions hold?

```
quickSort = insertionSort
```

It depends on the point of view:

Intensional point of view. (with an 's')
Two values are equal if they are built in the same way.

Extensional point of view.
Two values are equal if they are indistinguishable when observed.

Example

`quickSort` and `insertionSort`

- are **not** intensionally equal;
- **are** extensionally equal: they compute the same function.

Functional extensionality principle

Let `f, g :: a -> b`.

Immediate property

If `f = g` then $(\forall$`x` :: `a`. `f x = g x`$)$.

Functional extensionality principle

If $(\forall$`x` :: `a`. `f x = g x`$)$ then `f = g`.

Functional extensionality principle

Example: functional extensionality

```
{I} id x = x                 {C} (g . f) x = g (f x)
{S} swap (x, y) = (y, x)
```

Let's see that `swap . swap = id` :: `(a, b) -> (a, b)`.
By functional extensionality, it suffices to check:

$$\forall \texttt{p} :: \texttt{(a, b). (swap . swap) p = id p}$$

By induction on pairs, it suffices to check:

$$\forall \texttt{x} :: \texttt{a.}\ \forall \texttt{y} :: \texttt{b. (swap . swap) (x, y) = id (x, y)}$$

Indeed:

```
    (swap . swap) (x, y)
  = swap (swap (x, y))      (by C)
  = swap (y, x)             (by S)
  = (x, y)                  (by S)
  = id (x, y)               (by I)
```

Summary: equational reasoning

We reason equationally using three principles:

1. **Replacement principle**
   If the program declares that `e1 = e2`, any instance of `e1` is equal to the corresponding instance of `e2`, and vice versa.
2. **Structural induction principle**
   To prove $\mathcal{P}$ for all instances of a type `T`, it suffices to prove $\mathcal{P}$ for each of the constructors
   (assuming the I.H. for recursive constructors).
3. **Functional extensionality principle**
   To prove that two functions are equal, it suffices to prove that they are equal pointwise.

Correctness of equational reasoning

Suppose we manage to prove that `e1 = e2`.
What does that assure us about `e1` and `e2`?

Caution: they do not necessarily result in the same "data"

For example, it can be shown extensionally that:

```
quickSort = insertionSort
```

but `quickSort` and `insertionSort` are different "data".

They are different codes that represent the same mathematical function.

Correctness with respect to observations

If we prove `e1 = e2 :: A`, then:

$$\texttt{obs e1} \rightsquigarrow \texttt{True} \quad \text{if and only if} \quad \texttt{obs e2} \rightsquigarrow \texttt{True}$$

for every possible "observation" `obs :: A -> Bool`.

Proving inequalities

How do we prove that an equality `e1 = e2 :: A` **does not** hold?

By the contrapositive of the previous statement, it suffices to find an observation `obs :: A -> Bool` that distinguishes them.

Example

Prove that the equality **does not** hold:

```
id = swap :: (Int, Int) -> (Int, Int)
```

```
obs :: ((Int, Int) -> (Int, Int)) -> Bool
obs f = fst (f (1, 2)) == 1
```

```
obs id    ⇝  True
obs swap  ⇝  False
```

Introduction

Structural induction

Extensionality

Type isomorphisms

Case studies

Same information, different form

What is the relationship between the following values?

```
("hello", (1, True)) :: (String, (Int, Bool))
((True, "hello"), 1) :: ((Bool, String), Int)
```

They represent the same information, but written in different ways.

We can transform values of one type into values of the other:

```
f :: (String, (Int, Bool)) -> ((Bool, String), Int)
f (s, (i, b)) = ((b, s), i)

g :: ((Bool, String), Int) -> (String, (Int, Bool))
g ((b, s), i) = (s, (i, b))
```

It can be shown that:

```
g . f = id     f . g = id
```

Type isomorphisms

Definition

We say that two data types `A` and `B` are **isomorphic** if:

1. There is a total function `f :: A -> B`.
2. There is a total function `g :: B -> A`.
3. It can be shown that `g . f = id :: A -> A`.
4. It can be shown that `f . g = id :: B -> B`.

We write `A` $\simeq$ `B` to indicate that `A` and `B` are isomorphic.

Example of isomorphism: currying

Example

Let's see that `((a, b) -> c)` $\simeq$ `(a -> b -> c)`.

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

Example of isomorphism: currying

Let's see that

`uncurry . curry = id  :: ((a, b) -> c) -> (a, b) -> c`

By functional extensionality, it suffices to see that if
`f :: (a, b) -> c`:

`(uncurry . curry) f = id f  :: (a, b) -> c`

By functional extensionality, it suffices to see that if `p :: (a, b)`:

`(uncurry . curry) f p = id f p  :: c`

By induction on pairs, it suffices to see that if `x :: a`, `y :: b`:

`(uncurry . curry) f (x, y) = id f (x, y)  :: c`

Indeed:

```
    (uncurry . curry) f (x, y)
  = uncurry (curry f) (x, y)     (Def. (.))
  = curry f x y                  (Def. uncurry)
  = f (x, y)                     (Def. curry)
  = id f (x, y)                  (Def. id)
```

(And `curry . uncurry = id` also holds).

More type isomorphisms

```
(a, b)            ≃       (b, a)
(a, (b, c))       ≃       ((a, b), c)
a -> b -> c       ≃       b -> a -> c
a -> (b, c)       ≃       (a -> b, a -> c)
Either a b -> c   ≃       (a -> c, b -> c)
```

Introduction

Structural induction

Extensionality

Type isomorphisms

Case studies

Example — Need for auxiliary lemmas

We assume the usual definitions for `(.)` and `(++)` and the following for `reverse`:

```
{R0} reverse []       = []
{R1} reverse (x : xs) = reverse xs ++ [x]
```

Consider also the following definition:

```
     zeros :: [a] -> [Int]
{Z0} zeros []       = []
{Z1} zeros (_ : xs) = 0 : zeros xs
```

Let's prove that `zeros . reverse = reverse . zeros`.

What happens?

We need an **auxiliary lemma**:

`∀xs ys :: [a]. zeros (xs ++ ys) = zeros xs ++ zeros ys`

Example — Need to generalize the inductive predicate

Consider the following definition, using iterative recursion:

```
     sum :: Int -> [Int] -> Int
{S0} sum k []         = k
{S1} sum k (x : xs) = sum (x + k) xs
```

Let's prove that for `k :: Int` and `xs :: [Int]` the following holds:

```
sum k (xs ++ ys) = sum (sum k xs) ys
```

What happens?

We need to **generalize** the inductive predicate from $\mathcal{P}$ to $\mathcal{Q}$:

$\mathcal{P}(\texttt{xs}) \equiv$ `sum k (xs ++ ys) = sum (sum k xs) ys`

$\mathcal{Q}(\texttt{xs}) \equiv$ `∀k' :: Int. sum k' (xs ++ ys) = sum (sum k' xs) ys`

Example — Need to generalize the inductive predicate

We define functions to accumulate a list, using iterative and structural recursion:

```
{L0} accumL k []       = []
{L1} accumL k (x : xs) = (x + k) : accumL (x + k) xs

{R0} accumR []       = []
{R1} accumR (x : xs) = x : map (+ x) (accumR xs)
```

Let's prove that `accumL 0 = accumR`.

What happens?

We need to **generalize** the inductive predicate from $\mathcal{P}$ to $\mathcal{Q}$:

$$\mathcal{P}(\texttt{xs}) \equiv \boxed{\texttt{accumL 0 xs = accumR xs}}$$

$$\mathcal{Q}(\texttt{xs}) \equiv \boxed{\forall\texttt{k} :: \texttt{Int. accumL k xs = map (+ k) (accumR xs)}}$$

(The complete proof requires some more auxiliary lemmas).

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Recommended reading

Chapter 6 of Bird's book.

Richard Bird. *Thinking functionally with Haskell*
Cambridge University Press, 2015.