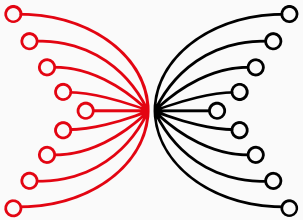
Laws and Proofs

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-01-19



INPUT | OUTPUT

Goals

- Revisiting equational reasoning.
- Stating and proving laws.
- (Structural) Induction.
- Parametricity.

Equational reasoning

Haskell evaluation, informally

- The evaluation (reduction) process in Haskell is based on applying equations that define functions from left to right.
- If no more reductions apply, we obtain a value that is the result of the expression.
- Applying an equation from left to right usually gets us a step closer to the value (if evaluation terminates).
- Applying an equation from right to left also does not change the value of an expression.

Equational reasoning

- We can transform expressions into other expressions by applying equations from left to right or right to left without changing their values.
- A sequence of equational rewrites consitutes a proof that one expression is equivalent to another.
- We can also apply other transformations and reasoning principles of which we have established that they do not change the value of the expression. (In particular, lemmas and theorems that we have already proved.)

Working with variables

- If we assume a value of a particular type and give it a name and make no further assumptions about the value in our transformations, then the statement holds for any well-typed choice of that value.

Example

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

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

Theorem

```
forall f a b .
curry (uncurry f) a b = f a b
```

We leave the types implicit if they can easily be inferred.

Proof

```
  curry (uncurry f) a b
=   {apply curry }
  uncurry f (a, b)
=   {apply uncurry }
  f a b
```

A second example

Theorem

```
forall f a b .
uncurry (curry f) (a, b) = f (a, b)
```

A second example

Theorem

```
forall f a b .
uncurry (curry f) (a, b) = f (a, b)
```

Both theorems together seem to prove that `curry` and `uncurry` are mutual inverses and form an isomorphism between functions of type

```
a -> b -> c
```

and functions of type

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

Proof

```
  uncurry (curry f) (a, b)
=   {apply uncurry }
  curry f a b
=   {apply curry }
  f (a, b)
```

The principle of extensionality

```
forall f a b .
curry (uncurry f) a b = f a b
```

- This theorem states that the functions `f` and `curry (uncurry f)` are extensionally equal, i. e., equal on all possible inputs.

The principle of extensionality

```
forall f a b .
curry (uncurry f) a b = f a b
```

- This theorem states that the functions `f` and `curry (uncurry f)` are extensionally equal, i. e., equal on all possible inputs.
- This is the form of equality that usually interests us. Stronger forms of equality (such as intensional equality) fix too much and do not allow us to do any interesting transformations.

What are all possible inputs?

```
forall f a b .
uncurry (curry f) (a, b) = f (a, b)
```

Does this theorem state that `f` and `uncurry (curry f)` are extensionally equal?

What are all possible inputs?

```
forall f a b .
uncurry (curry f) (a, b) = f (a, b)
```

Does this theorem state that `f` and `uncurry (curry f)` are extensionally equal?

What if we apply it to `undefined`?

Let’s investigate

```haskell
fun :: (Int, Int) -> Int
fun _ = 0
```

```
GHCi> fun undefined
0
GHCi> uncurry (curry f) undefined
*** Exception: Prelude.undefined
```

Let's investigate

```
fun :: (Int, Int) -> Int
fun _ = 0
```

```
GHCi> fun undefined
0
GHCi> uncurry (curry f) undefined
*** Exception: Prelude.undefined
```

So there is a value of type `(a, b)`, namely `undefined`, on which both functions do not coincide. If we are precise, they are not extensionally equal.

Fast and loose reasoning

- If we want to derive properties that truly hold for all inputs, we have to take special care of partial values, i. e., values involving `undefined`.
- All of `undefined`, `error` calls and nonterminating terms are typically thrown together and given the name $\bot$ when talking about theory. (Can you see why?)
- We often don't care about partial values though, and are ignoring them. This degree of imprecision is in principle justified by the paper "Fast and Loose Reasining is Morally Correct" by Danielsson, Hughes, Jansson and Gibbons.[1]

[1] It essentially says that the imprecisions introduced by ignoring undefined values do not accumulate without control. Statements being made in this way will remain valid for total values.

Induction

Functions over lists

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Theorem

```
forall xs .
xs ++ [] = xs
```

How do we prove this?

Constructing lists

```haskell
data [a] = [] | a : [a]  -- special syntax
```

Constructing lists

```
data [a] = [] | a : [a]  -- special syntax
```

There are two (or three) ways to construct a list:

- The empty list `[]` is a list.
- If we have a list `xs`, then `x : xs` is a list.
- (The undefined value $\bot$ is a list.)

Constructing lists

```
data [a] = [] | a : [a]  -- special syntax
```

There are two (or three) ways to construct a list:

- The empty list `[]` is a list.
- If we have a list `xs`, then `x : xs` is a list.
- (The undefined value $\bot$ is a list.)

Again, we will often drop the third case, but if we want to be precise, we have to include it.

Induction principle for lists

From the datatype, we can read off an induction principle.

For `xs :: [a]`, let `P xs` be a property.

Induction principle for lists

From the datatype, we can read off an induction principle.

For `xs :: [a]`, let `P xs` be a property.

E.g. `xs ++ [] = xs` is a property.

Induction principle for lists

From the datatype, we can read off an induction principle.

For `xs :: [a]`, let `P xs` be a property.

E.g. `xs ++ [] = xs` is a property.

Then `forall xs . P xs` holds if:

- `P []` holds.
- `forall y ys . P ys ⇒ P (y : ys)` holds.
- (`P ⊥` holds.)

Proving for the empty list

Case `xs = []`:

```
   xs ++ []
=    { assumption }
   [] ++ []
=    { apply (++) }
   []
=    { assumption }
   xs
```

Proving for non-empty lists

Case `xs = y : ys`, assuming the induction hypothesis `ys ++ [] = ys`:

```
   xs ++ []
=    { assumption }
   (y : ys) ++ []
=    { apply (++) }
   y : (ys ++ [])
=    { induction hypothesis }
   y : ys
=    { assumption }
   xs
```

Proving for undefined lists

Case `xs` $= \bot$:

```
   xs ++ []
=    { assumption }
   ⊥ ++ []
=    { apply (++) which forces the first argument }
   ⊥
=    { assumption }
   xs
```

Proving for undefined lists

Case `xs = ⊥`:

```
   xs ++ []
=    { assumption }
   ⊥ ++ []
=    { apply (++) which forces the first argument }
   ⊥
=    { assumption }
   xs
```

So this property holds even in the context of `⊥`.

If we add this case, then we cover also partially defined lists such as e.g. `1 : 2 : ⊥`.

Associativity of append

Theorem

```
forall xs ys zs .
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

We prove this by induction on `xs`.

Empty list

Case `xs = []`:

```
   [] ++ (ys ++ zs)
=    { apply (++) }
   ys ++ zs
=    { reverse-apply (++) }
   ([] ++ ys) ++ zs
```

Non-empty list

Case `xs = w : ws`, assuming property for `ws`:

```
   (w : ws) ++ (ys ++ zs)
=    { apply (++) }
   w : (ws ++ (ys ++ zs))
=    { induction hypothesis }
   w : ((ws ++ ys) ++ zs)
=    { reverse-apply (++) }
   (w : (ws ++ ys)) ++ zs
=    { reverse-apply (++) }
   ((w : ws) ++ ys) ++ zs
```

Undefined list

Case `xs` $= \bot$:

```
   ⊥ ++ (ys ++ zs)
=    { apply (++) }
   ⊥
=    { reverse-apply (++) }
   ⊥ ++ zs
=    { reverse-apply (++) }
   (⊥ ++ ys) ++ zs
```

Two ways to reverse a list

A more realistic example

```haskell
reverse1 :: [a] -> [a]
reverse1 []       = []
reverse1 (x : xs) = reverse1 xs ++ [x]
```

```haskell
reverse2 :: [a] -> [a]
reverse2 = reverse2' []

reverse2' :: [a] -> [a] -> [a]
reverse2' acc []       = acc
reverse2' acc (x : xs) = reverse2' (x : acc) xs
```

Can we prove that `reverse1` and `reverse2` are equivalent (extensionally equal)?

Stating the theorem

Theorem

```
forall xs .
reverse1 xs = reverse2 xs
```

Stating the theorem

Theorem

```
forall xs .
reverse1 xs = reverse2 xs
```

- It turns out that this theorem is not trivial to prove directly.
- But we can easily prove it using the two properties of `(++)` we have established already, and a lemma.
- Coming up with the right lemma requires a bit of creativity.

What we need

Lemma

```
forall y ys .
reverse2' xs ys = reverse1 ys ++ xs
```

(We actually used this as the specification for `reverse2'` back in the exercises of the first week.)

Let's prove the lemma first

Case `ys = []`:

```
   reverse2' xs []
=    {apply reverse2'}
   xs
=    {reverse-apply (++)}
   [] ++ xs
=    {reverse-apply reverse1}
   reverse1 [] ++ xs
```

And the interesting case

Case `ys = z : zs`, assuming the property for `zs`:

```
  reverse2' xs (z : zs)
=   { apply reverse2' }
  reverse2' (z : xs) zs
=   { induction hypothesis }
  reverse1 zs ++ (z : xs)
=   { use [z] ++ xs = z : xs }
  reverse1 zs ++ ([z] ++ xs)
=   { use associativity of (++) }
  (reverse1 zs ++ [z]) ++ xs
=   { reverse-apply reverse1 }
  reverse1 (z : zs) ++ xs
```

Let's consider undefined lists

Case `ys = ⊥`:

```
   reverse2' xs ⊥
=    {apply reverse2'}
   ⊥
=    {reverse-apply (++)}
   ⊥ ++ xs
=    {reverse-apply reverse1}
   reverse1 ⊥ ++ xs
```

Now the original property

We can prove this now without (further) induction:

```
   reverse1 xs
=    { use the append-empty-list property }
   reverse1 xs ++ []
=    { use the lemma }
   reverse2' [] xs
=    { reverse-apply reverse2' }
   reverse2 xs
```

Functor laws

Trees are functors

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Trees are functors

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Trees are functors

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```haskell
instance Functor Tree where
  fmap f (Leaf a)   = Leaf (f a)
  fmap f (Node l r) = Node (fmap f l) (fmap f r)
```

The functor laws

Instances of `Functor` are supposed to adhere to the following two laws:

```
forall x .
fmap id x = x
```

```
forall f g x .
fmap (f . g) x = (fmap f . fmap g) x
```

The functor laws

Instances of `Functor` are supposed to adhere to the following two laws:

```
forall x .
fmap id x = x
```

```
forall f g x .
fmap (f . g) x = (fmap f . fmap g) x
```

Let's try to prove this for the `Tree` instance.

Constructing trees

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Constructing trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

There are two (or three) ways to construct a tree:

- Given an element `a`, `Leaf a` is a tree.
- If we have trees `l` and `r`, then `Node l r` is a tree.
- (The undefined value $\bot$ is a tree.)

Induction principle for trees

For `t :: Tree a`, let `P t` be a property.

Induction principle for trees

For `t :: Tree a`, let `P t` be a property.

Then `forall t . P t` holds if:

- `forall a . P (Leaf a)` holds.
- `forall l r . (P l ∧ P r) ⇒ P (Node l r)` holds.
- (`P ⊥` holds.)

Proving the first functor law

Theorem

```
forall (t :: Tree a) .
fmap id t = t
```

Leaves

Case `t = Leaf a`:

```
   fmap id (Leaf a)
=    {apply fmap }
   Leaf (id a)
=    {apply id }
   Leaf a
```

Nodes

Case `t = Node l r`, assuming property for `l` and `r`:

```
   fmap id (Node l r)
=    { apply fmap }
   Node (fmap id l) (fmap id r)
=    { induction hypothesis, twice }
   Node l r
```

Undefined

Case `t = ⊥`:

```
   fmap id ⊥
=    {apply fmap}
   ⊥
```

Proving the second functor law

Theorem

```
forall f g (t :: Tree a) .
fmap (f . g) t = (fmap f . fmap g) t
```

Leaves

Case `t = Leaf a`:

```
  fmap (f . g) (Leaf a)
=   { apply fmap }
  Leaf ((f . g) a)
=   { apply (.) }
  Leaf (f (g a))
=   { reverse-apply fmap }
  fmap f (Leaf (g a))
=   { reverse-apply fmap }
  fmap f (fmap g (Leaf a))
=   { reverse-apply (.) }
  (fmap f . fmap g) (Leaf a)
```

Nodes

Case `t = Node l r`, assuming the property for `l` and `r`:

```
   fmap (f . g) (Node l r)
=    { apply fmap }
   Node (fmap (f . g) l) (fmap (f . g) r)
=    { induction hypothesis, twice }
   Node ((fmap f . fmap g) l) ((fmap f . fmap g) r)
=    { apply (.), twice }
   Node (fmap f (fmap g l)) (fmap f (fmap g r))
=  ...
```

Continuing

```
  fmap (f . g) (Node l r)
=   { steps on previous slide }
  Node (fmap f (fmap g l)) (fmap f (fmap g r))
```

Continuing

```
  fmap (f . g) (Node l r)
=   { steps on previous slide }
  Node (fmap f (fmap g l)) (fmap f (fmap g r))
=   { reverse-apply fmap }
  fmap f (Node (fmap g l) (fmap g r))
=   { reverse-apply fmap }
  fmap f (fmap g (Node l r))
=   { reverse-apply (.) }
  (fmap f . fmap g) (Node l r)
```

Parametric polymorphism

One function, several types

Some Haskell expressions and functions can have more than one type.

Example:

```
fst (x, y) = x
```

Possible type signatures (all would work):

```
fst :: (a, a) -> a
fst :: (Int, a) -> Int
fst :: (Int, Int) -> Int
fst :: (a, b) -> a
fst :: (Int, Char) -> Int
```

Is one of these clearly the “best” choice?

Most general type

Haskell's type system is designed such that (ignoring some language extensions) each term has a *most general type*:

- the most general type allows the most flexible use;
- all other types the term has can be obtained by instantiating the most general type, i.e., by substituting type variables with type expressions.

Instantiating types

The type signature

```
fst :: (a, b) -> a
```

declares the most general type for `fst`. Types like

```
fst :: (a, a) -> a
fst :: (Int, Char) -> Int
fst :: (a -> Int -> b, c) -> a -> Int -> b
```

are instantiations of the most general type.

Instantiating types

The type signature

```
fst :: (a, b) -> a
```

declares the most general type for `fst`. Types like

```
fst :: (a, a) -> a
fst :: (Int, Char) -> Int
fst :: (a -> Int -> b, c) -> a -> Int -> b
```

are instantiations of the most general type.

Type inference will always infer the most general type!

No run-time type information

Haskell terms carry no type information at run-time.

Remember

You can only ever use a term in the ways its type dictates.

No run-time type information

Haskell terms carry no type information at run-time.

Remember

You can only ever use a term in the ways its type dictates.

Example:

```
fst :: (a, b) -> a
fst (x, y) = x

restrictedFst :: (Int, Int) -> Int
restrictedFst = fst  -- ok
```

```
newFst :: (a, b) -> a
newFst = restrictedFst  -- type error!
```

Parametric polymorphism

- A type with type variables (but no class constraints) is called (parametrically) polymorphic.
- Type variables can be instantiated to any type expression, but several occurrences of the same variable have to be the same type.
- If a function argument has polymorphic type, then you know nothing about it. No pattern matching is possible. You can only pass it on.
- If a function result has polymorphic type, then (except for `undefined` and `error`) you can only try to build one from the function arguments.

Let us look at examples.

Example

How many functions can you think of[2] that have this type:

```
(Int, Int) -> (Int, Int)
```

[2] Example originally by Doaitse Swierstra.

Example

How many functions can you think of[2] that have this type:

```
(Int, Int) -> (Int, Int)
```

And of this one?

```
(a, a) -> (a, a)
```

[2] Example originally by Doaitse Swierstra.

Example

How many functions can you think of[2] that have this type:

```
(Int, Int) -> (Int, Int)
```

And of this one?

```
(a, a) -> (a, a)
```

And of this one?

```
(a, b) -> (b, a)
```

[2] Example originally by Doaitse Swierstra.

Parametricity

- Parametric polymorphism restricts how a function can be implemented.
- Prevents you from making implementation errors.
- If you see a function with parametrically polymorphic type, you *know* that it cannot look at the polymorphic values.
- By looking at polymorphic types alone, one can obtain non-trivial properties of the functions. (This is sometimes called “parametricity”.)

Parametricity on map

```
map :: (a -> b) -> [a] -> [b]
```

Parametricity on map

```
map :: (a -> b) -> [a] -> [b]
```

Parametricity tells us:

- Resulting list contains elements of type `b`.
- The only way to obtain type `b` is by applying the function to elements of the given list.
- We do *not* know how long the resulting list is, or in which order the elements occur.

A common pitfall: who gets to choose

```
parse :: String -> a
parse "False" = False
parse "0"     = 0
 ...
```

What is wrong here?

A common pitfall: who gets to choose

```
parse :: String -> a
parse "False" = False
parse "0"     = 0
 ...
```

What is wrong here?

For polymorphic types, it is always the caller who gets to choose at which type the function should be used.

A function with polymorphic result type (but no polymorphic arguments) is impossible to write without either looping or causing an exception: we'd have to produce a value that belongs to every type imaginable!

What if we need to return values of different types?

Option 1: use `Either`:

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

parse :: String -> Either Bool Int
parse "False" = Left False
parse "0"     = Right 0
```

[3] Common when embedding dynamically typed languages (SQL, JSON, ...).

What if we need to return values of different types?

Option 1: use `Either`:

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

parse :: String -> Either Bool Int
parse "False" = Left False
parse "0"     = Right 0
```

Option 2: define your own datatype[3].

```
data Value = VBool Bool | VInt Int

parse :: String -> Value
parse "False" = VBool False
parse "0"     = VInt 0
```

[3] Common when embedding dynamically typed languages (SQL, JSON, ...).