Financial Contracts in Haskell

- Haskell is good for Domain-Specific Languages (DSL)
- A good example are "financial combinators"[1]
- "How to write a financial contract", S.L. Peyton Jones and J-M. Eber, ICFP 2000.
  Let's take a quick tour ...
- Spin-out company: Lexifi.com

[1] Google it!

Proof better than Testing

- We have seen the use of testing for properties of interest.
- What if we don't trust these tests ?
  We want to be really, really sure ...
- Functional language proponents claim that such languages provide for easy reasoning ...
  - ...show me the 'proof' !

Properties of List Functions

- Some interesting list properties:
  ```
  reverse (reverse xs) == xs
  length (xs++ys) == length xs + length ys
  length xs == length (reverse xs)
  ```
- How do we prove these ?
  - We need definitions of , `++`, `reverse` and `length`
    ```
    [] ++ ys = ys
    (x:xs) ++ ys = x:(xs++ys)
    reverse [] = []
    reverse (x:xs) = reverse xs ++ [x]
    length [] = 0
    length (_:xs) = 1 + length xs
    ```
  - We need some proof laws, principles, techniques
    Referential Transparency, Induction, Case-Splitting

Principles

Referential Transparency We can always replace an expression in any context by one that is known to be equal to it.

Induction For any recursive `data` type, we prove the property true for the non-recursive variants, and then for every recursive (composite) variant, we assume the property is true for the recursive components, and from this show it still holds for the composite.

Case-Splitting Conditionals (and Patterns) are handled by case analysis —condition true/false, all possible matches.

The Convenience of Functional Reasoning

- Note how the program, properties and proofs can all be stated and manipulated in the same language: Haskell !

Proof Example (Property)

- Example: lets prove
  `length (xs++ys) = length xs + length ys`
- List are defined inductively as either `[]` or `x:xs`, where `xs` is a pre-existing list.
- We do an inductive proof, on `xs`
  `P(xs)` $\widehat{=}$ `length (xs++ys) = length xs + length ys`
  Why not do induction with `ys` instead?

Proof Example (Base Case)

```
P([])
 = "expand P"
  length ([]++ys) = length [] + length ys
 = "Defs of ++ and length"
  length ys = 0 + length ys
 = "arithmetic"
  length ys = length ys
 = "reflexivity of ="
  True
```

Easy !
Note how each line of the proof has a justification (in double quotes).

Proof Example (Inductive Step)

Assume `P(xs)`,
i.e. `length (xs++ys) = length xs + length ys`
Show `P(x:xs)`:

```
P(x:xs)
 = "expand P"
  length ((x:xs)++ys) = length (x:xs) + length ys
 = "Defs of ++ and length"
  length (x:(xs++ys)) = (1 + length xs) + length ys
 = "Defs of length, + is assoc"
  1 + length (xs++ys) = 1 + (length xs + length ys)
 = "arithmetic"
  length (xs++ys) = length xs + length ys
 = "by ind. hypothesis"
  True
```

Not so hard!

Set as unique ordered List

- Consider using a ordered list with no duplicate elements to represent a set.
- Insertion:

```
ins :: Ord t => t -> [t] -> [t]
ins x [] = [x]
ins x ys@(y:zs)
 | x < y      =  x : ys        -- smallest at start
 | x > y      =  y : ins x zs  -- recurse in
 | otherwise  =  ys            -- already present
```

- Membership test:

```
mbr :: Ord t => t -> [t] -> Bool
mbr _ []    =  False
mbr x (y:ys)
 | x < y      =  False
 | x > y      =  mbr x ys
 | otherwise  =  True
```

Insertion implies Membership

We want to prove that, after we do `ins x ys`, the test `mbr x ys` always returns `True`

```
mbr x (ins x ys)   =  True
```

We do an induction on `ys`:

```
P(ys) ≙ mbr x (ins x ys)
```

So we have to show:

```
P([])
P(ys) ==> P(y:ys)
```

Proof, Base Case

```
P([])
 = "defn of P"
  mbr x (ins x [])
 = "defn. of ins, 1st pattern"
  mbr x [x]
 = "defn. of mbr, 2nd pattern, otherwise case"
  True
```

Proof, Inductive Step

```
P(ys) ==> P(y:ys)
 = "defn of P"
  mbr x (ins x ys)  ==>  mbr x (ins x (y:ys))
```

We shall assume the lhs (induction hypothesis) and attempt to show the rhs is true.
We shall also anticipate a obvious case split, and treat the three conditions `x<y`, `x>y` and `x==y` separately.

Proof, Inductive Step, `x<y`

We assume: `mbr x (ins x ys)` and `x<y`.

```
mbr x (ins x (y:ys))
 = "defn. ins, 2nd pattern, x<y"
  mbr x (x:y:ys)
 = "defn. mbr, 2nd pattern otherwise case"
  True
```

Proof, Inductive Step, `x>y`

We assume: `mbr x (ins x ys)` and `x>y`.

```
mbr x (ins x (y:ys))
 = "defn. ins, 2nd pattern given x>y"
  mbr x (y: ins x ys)
 = "defn. mbr, 2nd pattern, x > y case"
  mbr x (ins x ys)
 = "inductive hypothesis"
  True
```

Proof, Inductive Step, `x==y`

We assume: `mbr x (ins x ys)` and `x==y`.

```
mbr x (ins x (y:ys))
 = "defn. ins, 2nd pattern, x==y"
  mbr x (y:ys)
 = "defn. mbr, 2nd pattern, otherwise case"
  True
```