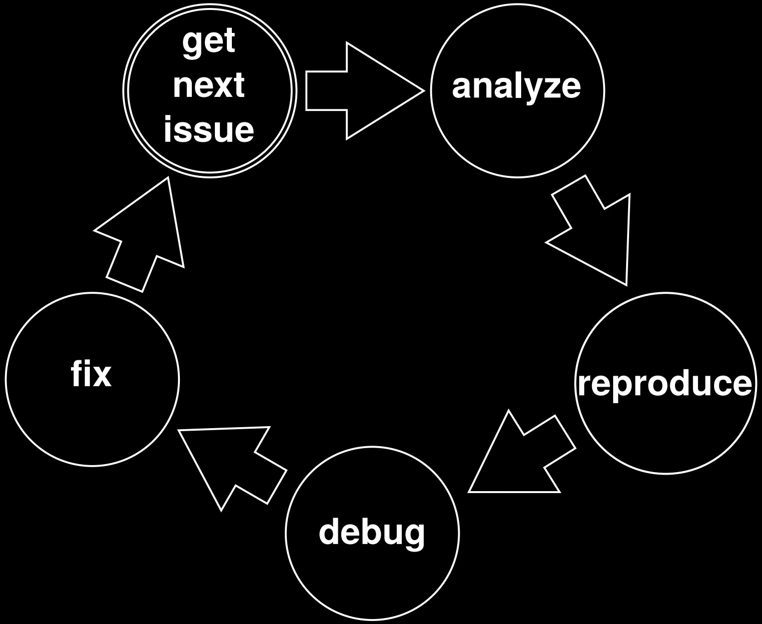
TWEAG
A Modus Create Company

Untangle your spaghetti with Liquid Haskell

Facundo Domínguez

December 8th, 2022


get next issue
analyze
reproduce
debug
fix

Formal verification?

Formal verification?

- Difficult
- Expensive
- Tedious

Formal verification?

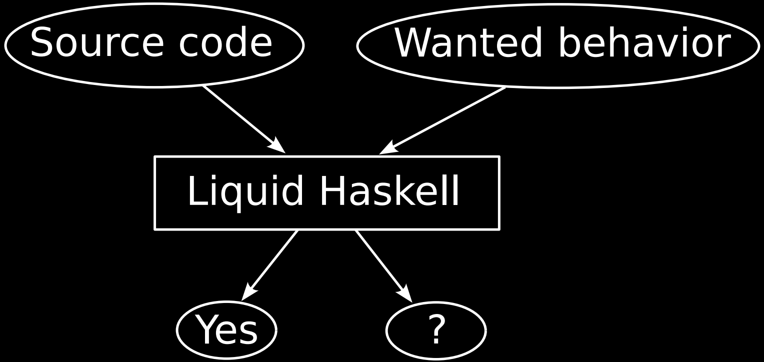
- Difficult
- Expensive
- Tedious
- Easy, Cheap, and Fun

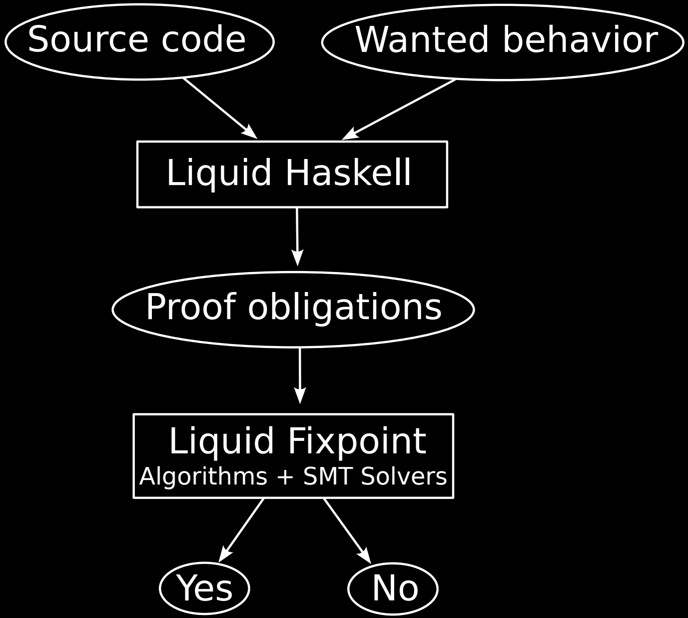
```
Data.List.permutations "abc"

==

["abc", "bac", "cba", "bca", "cab", "acb"]
```

```
permutations ("abc" ++ undefined)

==

[ "abc" ++ undefined
, "bac" ++ undefined
, "cba" ++ undefined
, "bca" ++ undefined
, "cab" ++ undefined
, "acb" ++ undefined
]
++ undefined
```

```haskell
permutations :: [a] -> [[a]]
permutations xs = xs : perms xs []
  where
    perms :: forall a. [a] -> [a] -> [[a]]
    perms []     _  = []
    perms (t:ts) is =
        foldr interleave (perms ts (t:is)) (permutations is)
      where
        interleave :: [a] -> [[a]] -> [[a]]
        interleave ys r = let (_,zs) = interleave' id ys r in zs
        interleave' :: ([a] -> b) -> [a] -> [b] -> ([a], [b])
        interleave' _ []     r = (ts, r)
        interleave' f (y:ys) r =
            let (us, zs) = interleave' (f . (y:)) ys r
             in (y:us, f (t:y:us) : zs)
```

```haskell
permutations :: [a] -> [[a]]
permutations xs = xs : perms xs []
  where
    perms :: forall a. [a] -> [a] -> [[a]]
    perms []     _  = []
    perms (t:ts) is =
        foldr interleave (perms ts (t:is)) (permutations is)
      where
        interleave :: [a] -> [[a]] -> [[a]]
        interleave ys r = let (_,zs) = interleave' id ys r in zs
        interleave' :: ([a] -> b) -> [a] -> [b] -> ([a], [b])
        interleave' _ []     r = (ts, r)
        interleave' f (y:ys) r =
            let (us, zs) = interleave' (f . (y:)) ys r
             in (y:us, f (t:y:us) : zs)
```


Source code
Wanted behavior
Liquid Haskell
Yes
?


Source code
Wanted behavior
Liquid Haskell
Proof obligations
Liquid Fixpoint
Algorithms + SMT Solvers
Yes
No

```haskell
permutations :: [a] -> [[a]]
permutations xs = xs : perms xs []
  where
    perms :: forall a. [a] -> [a] -> [[a]]
    perms []     _  = []
    perms (t:ts) is =
        foldr interleave (perms ts (t:is)) (permutations is)
      where
        interleave :: [a] -> [[a]] -> [[a]]
        interleave ys r = let (_,zs) = interleave' id ys r in zs
        interleave' :: ([a] -> b) -> [a] -> [b] -> ([a], [b])
        interleave' _ []     r = (ts, r)
        interleave' f (y:ys) r =
            let (us, zs) = interleave' (f . (y:)) ys r
             in (y:us, f (t:y:us) : zs)
```

```haskell
permutations :: [a] -> [[a]]
permutations xs = xs : perms xs []

perms []       _  = []
perms (t:ts) is =
    foldr (interleave t ts) (perms ts (t:is)) (permutations is)

interleave t ts ys r =
    let (_,zs) = interleave' id t ts ys r
     in zs

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```haskell
permutations :: [a] -> [[a]]
permutations xs = xs : perms xs []

perms []     _  = []
perms (t:ts) is =
    foldr (interleave t ts) (perms ts (t:is)) (permutations is)

interleave t ts ys r =
    let (_,zs) = interleave' id t ts ys r
     in zs

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```
interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```
interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)

fst (interleave' f t ts ys r) =
    ys ++ ts
```

```haskell
{-@



@-}

interleave' :: ([a] -> b) -> a -> [a] -> [a] -> [b] -> ([a], [b])
```

```
{-@
interleave'
  ::   ([a] -> b) ->   a ->    [a] ->    [a] ->   [b] ->
         ([a], [b])



@-}

interleave' :: ([a] -> b) -> a -> [a] -> [a] -> [b] -> ([a], [b])
```

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
        ([a], [b])


@-}

interleave' :: ([a] -> b) -> a -> [a] -> [a] -> [b] -> ([a], [b])
```

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b])

     }
@-}

interleave' :: ([a] -> b) -> a -> [a] -> [a] -> [b] -> ([a], [b])
```

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' :: ([a] -> b) -> a -> [a] -> [a] -> [b] -> ([a], [b])
```

```haskell
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```haskell
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r) :: { v:_ | fst v = ts }
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```haskell
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r) :: { v:_ | fst v = ts }
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```haskell
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
          :: { v:_ | fst v = y : us }
```

```haskell
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
          :: { v:_ | fst v = ys ++ ts && fst v = us }
     in (y:us, f (t:y:us) : zs)
          :: { v:_ | fst v = y : us }
```

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
          :: { v:_ | fst v = ys ++ ts && fst v = us }
     in (y:us, f (t:y:us) : zs)
          :: { v:_ | fst v = y : us }
```

```haskell
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts
     }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

QED

```
interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)

snd (interleave' f t ts ys r) = ?
```

```
insertAt :: Int -> a -> [a] -> [a]
insertAt 0 'z' "ab" = "zab"
insertAt 1 'z' "ab" = "azb"
insertAt 2 'z' "ab" = "abz"
```

```
interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)

snd (interleave' f t ts ys r) =
    [ f (insertAt n t ys ++ ts)
      | n <- [0..length ys - 1]
    ]
    ++ r
```

```haskell
interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)

snd (interleave' f t ts ys r) =
    [ f (insertAt n t ys ++ ts)
      | n <- [0..length ys - 1]
    ]
    ++ r
```

UNOPTIMIZED

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
        fst v = ys ++ ts





     }
@-}
```

```
{-@
interleave'
  :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
         fst v = ys ++ ts
           &&
         snd v = [ f (insertAt n t ys ++ ts)
                   | n <- [0..lenght ys - 1]
                 ]
                 ++ r
     }
@-}
```

```
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
      { v:([a], [b]) |
        fst v = ys ++ ts &&
        snd v = [ f (insertAt n t ys ++ ts) | n <- [0..lenght ys - 1] ]
                ++ r }
```

```
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
      { v:([a], [b]) |
        fst v = ys ++ ts &&
        snd v = [ f (insertAt n t ys ++ ts) | n <- [0..lenght ys - 1] ]
                ++ r }

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts &&
       snd v = [ f (insertAt n t ys ++ ts) | n <- [0..lenght ys - 1] ]
               ++ r }

interleave' f t ts []     r = (ts, r) :: { v:_ | snd v = r }
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```haskell
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts &&
       snd v = [ f (insertAt n t ys ++ ts) | n <- [0..lenght ys - 1] ]
               ++ r }

interleave' f t ts []     r = (ts, r) :: { v:_ | snd v = r }
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts &&
       snd v = [ f (insertAt n t ys ++ ts) | n <- [0..lenght ys - 1] ]
               ++ r }

interleave' f t ts []     r = (ts, r) :: { v:_ | snd v = r }
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
          :: { v:_ | snd v = f (t:y:us) : zs }
```

```
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts &&
       snd v = [ f (insertAt n t ys ++ ts) | n <- [0..lenght ys - 1] ]
               ++ r }

interleave' f t ts []     r = (ts, r) :: { v:_ | snd v = r }
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
          :: { v:_ | snd v = [ (f . (y :)) (insertAt n t ys ++ ts)
                               | n <- [0..length ys - 1]
                              ] ++ r && snd v = zs }
     in (y:us, f (t:y:us) : zs)
          :: { v:_ | snd v = f (t:y:us) : zs }
```

Inferred

```
:: { v:_ | snd v = f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
                                  | n <- [0..length ys - 1]
                                ] ++ r }
```

Inferred

```
:: { v:_ | snd v = f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
                                  | n <- [0..length ys - 1]
                                ] ++ r }
```

Wanted

```
:: { v:_ | snd v = [ f (insertAt n t (y:ys) ++ ts)
                   | n <- [0..length (y:ys) - 1]
                 ] ++ r }
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
               | n <- [0..length ys - 1]
             ] ++ r

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [0..length (y:ys) - 1]
] ++ r
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
               | n <- [0..length ys - 1]
             ] ++ r

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [0..length (y:ys) - 1]
] ++ r
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
             | n <- [0..length ys - 1]
             ]

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [0..length (y:ys) - 1]
]
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
               | n <- [0..length ys - 1]
             ]

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [0..length (y:ys) - 1]
]
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
               | n <- [0..length ys - 1]
             ]

    =

f (insertAt 0 t (y:ys) ++ ts)
  : [ f (insertAt n t (y:ys) ++ ts)
      | n <- [1..length (y:ys) - 1]
    ]
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
               | n <- [0..length ys - 1]
             ]

    =

f ((t:y:ys) ++ ts)
  : [ f (insertAt n t (y:ys) ++ ts)
      | n <- [1..length (y:ys) - 1]
    ]
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
               | n <- [0..length ys - 1]
             ]

    =

f (t:y: (ys ++ ts))
  : [ f (insertAt n t (y:ys) ++ ts)
      | n <- [1..length (y:ys) - 1]
    ]
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
                 | n <- [0..length ys - 1]
               ]

    =

f (t:y:us)
  : [ f (insertAt n t (y:ys) ++ ts)
      | n <- [1..length (y:ys) - 1]
    ]
```

```
f (t:y:us) : [ f (y : insertAt n t ys ++ ts)
                 | n <- [0..length ys - 1]
               ]

    =

f (t:y:us)
  : [ f (insertAt n t (y:ys) ++ ts)
      | n <- [1..length (y:ys) - 1]
    ]
```

```
[ f (y : insertAt n t ys ++ ts)
  | n <- [0..length ys - 1]
]

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [1..length (y:ys) - 1]
]
```

```
[ f (y : insertAt n t ys ++ ts)
  | n <- [0..length ys - 1]
]

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [1..length (y:ys) - 1]
]
```

```
[ f (y : insertAt n t ys ++ ts)
  | n <- [0..length ys - 1]
]

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [1..length ys + 1 - 1]
]
```

```haskell
[ f (y : insertAt n t ys ++ ts)
  | n <- [0..length ys - 1]
]

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [1..length ys + 1 - 1]
]
```

```
[ f (y : insertAt n t ys ++ ts)
  | n <- [i..j]
]

    =

[ f (insertAt n t (y:ys) ++ ts)
  | n <- [i+1..j+1]
]
```

```haskell
{-@
lemmaListInsertAt
  :: t:a -> f:([a] -> [a]) -> y:a -> ys:[a] -> ts:[a]
  -> { i:Int | 0 <= i } -> j:Int
  -> { _v:() |
       [ f (y : insertAt n t ys ts) | n <- [i..j] ]
       = [ f (insertAt n t (y:ys) ts) | n <- [i+1 .. j+1] ]
     }
@-}
lemmaListInsertAt
    :: a -> ([a] -> [a]) -> a -> [a] -> [a] -> Int -> Int -> ()
lemmaListInsertAt t f y ys ts i j =
```

```haskell
{-@
assume lemmaListInsertAt
  :: t:a -> f:([a] -> [a]) -> y:a -> ys:[a] -> ts:[a]
  -> { i:Int | 0 <= i } -> j:Int
  -> { _v:() |
       [ f (y : insertAt n t ys ts) | n <- [i..j] ]
       = [ f (insertAt n t (y:ys) ts) | n <- [i+1 .. j+1] ]
     }
@-}
lemmaListInsertAt
    :: a -> ([a] -> [a]) -> a -> [a] -> [a] -> Int -> Int -> ()
lemmaListInsertAt t f y ys ts i j = ()
```

```
{-@
lemmaListInsertAt
  :: t:a -> f:([a] -> [a]) -> y:a -> ys:[a] -> ts:[a]
  -> { i:Int | 0 <= i } -> j:Int
  -> { _v:() |
       [ f (y : insertAt n t ys ts) | n <- [i..j] ]
       = [ f (insertAt n t (y:ys) ts) | n <- [i+1 .. j+1] ]
    }
@-}
lemmaListInsertAt
    :: a -> ([a] -> [a]) -> a -> [a] -> [a] -> Int -> Int -> ()
lemmaListInsertAt t f y ys ts i j =
    if i <= j then lemmaListInsertAt t f y ys ts (i+1) j
      else ()
```

```
{-@
lemmaListInsertAt
  :: t:a -> f:([a] -> [a]) -> y:a -> ys:[a] -> ts:[a]
  -> { i:Int | 0 <= i } -> j:Int
  -> { _v:() |
       [ f (y : insertAt n t ys ts) | n <- [i..j] ]
       = [ f (insertAt n t (y:ys) ts) | n <- [i+1 .. j+1] ]
     } / [j-i+1]
@-}
lemmaListInsertAt
    :: a -> ([a] -> [a]) -> a -> [a] -> [a] -> Int -> Int -> ()
lemmaListInsertAt t f y ys ts i j =
    if i <= j then lemmaListInsertAt t f y ys ts (i+1) j
      else ()
```

```
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts &&
       snd v = [ f (insertAt n t ys ++ ts) | n <- [0..length ys - 1] ]
               ++ r }

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in ( y:us
        , const (f (t:y:us) : zs)
                (lemmaListInsertAt t f y ys ts 0 (length ys - 1))
        )
```

```
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
      { v:([a], [b]) |
        fst v = ys ++ ts &&
        snd v = [ f (insertAt n t ys ++ ts) | n <- [0..length ys - 1] ]
                ++ r }

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in ( y:us
        , const (f (t:y:us) : zs)
                (lemmaListInsertAt t f y ys ts 0 (length ys - 1))
        )

const a _ = a
```

```haskell
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
      { v:([a], [b]) |
        fst v = ys ++ ts &&
        snd v = [ f (insertAt n t ys ++ ts) | n <- [0..length ys - 1] ]
                ++ r }

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in ( y:us
        , const (f (t:y:us) : zs)
                (lemmaListInsertAt t f y ys ts 0 (length ys - 1))
        )
```

QED

```haskell
permutations :: [a] -> [[a]]
permutations xs = xs : perms xs []

perms []          _  = []
perms (t:ts) is =
    foldr (interleave t ts) (perms ts (t:is)) (permutations is)

interleave t ts ys r =
    let (_,zs) = interleave' id t ts ys r
     in zs

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in (y:us, f (t:y:us) : zs)
```

```
interleave t ts ys r =
    let (_,zs) = interleave' id t ts ys r
     in zs
```

```
interleave t ts ys r =
  [ insertAt n t ys ++ ts | n <- [0..length ys - 1] ] ++ r
```

```
perms []     _  = []
perms (t:ts) is =
    foldr (interleave t ts) (perms ts (t:is)) (permutations is)

perms ts is =
  xs : concat
    [ interleave (ts!!n) (drop (n+1) ts) ys []
      | n <- [0..length ts - 1]
      , ys <- permutations (reverse (take n ts) ++ is)
    ]
```

4 lemmas

```
permutations xs = xs : perms xs []
```

```
permutations xs =
  xs : concat
    [ interleave (ts!!n) (drop (n+1) xs) ys []
      | n <- [0..length xs - 1]
      , ys <- permutations (reverse (take n xs))
    ]
```

4 lemmas

```
permutations xs = xs : perms xs []
```

```
permutations xs =
  xs : concat
    [ interleave (ts!!n) (drop (n+1) xs) ys []
      | n <- [0..length xs - 1]
      , ys <- permutations (reverse (take n xs))
    ]
```

UNOPTIMIZED

4 lemmas

Documenting and explaining

Easy?? Cheap?? Fun??

```haskell
{-@
interleave' :: f:([a] -> b) -> t:a -> ts:[a] -> ys:[a] -> r:[b] ->
     { v:([a], [b]) |
       fst v = ys ++ ts &&
       snd v = [ f (insertAt n t ys ++ ts) | n <- [0..length ys - 1] ]
               ++ r }
@-}

interleave' f t ts []     r = (ts, r)
interleave' f t ts (y:ys) r =
    let (us, zs) = interleave' (f . (y:)) t ts ys r
     in ( y:us
        , const (f (t:y:us) : zs)
                (lemmaListInsertAt t f y ys ts 0 (length ys - 1))
        )
```

```
permutations ("abc" ++ undefined)

==

[ "abc" ++ undefined
, "bac" ++ undefined
, "cba" ++ undefined
, "bca" ++ undefined
, "cab" ++ undefined
, "acb" ++ undefined
]
++ undefined
```

```
map (take n) (take (factorial n) $ permutations ([1..n] ++ undefined))
  =
permutations [1..n]
```

```
map (take n) (take (factorial n) $ permutations ([1..n] ++ sfx))
  =
permutations [1..n]
```

```
lemmaPermutationsDecomposition
  ::  n:Int | n >= 0
  -> sfx:[Int]
  { _v:() |
    map (take n) (take (factorial n) (permutations ([1..n] ++ sfx)))
      =
    permutations [1..n]
  }
```

proof requires 16 lemmas

```
lemmaPermutationsDecomposition
  ::  n:Int | n >= 0
  -> sfx:[Int]
  { _v:() |
    map (take n) (take (factorial n) (permutations ([1..n] ++ sfx)))
      =
    permutations [1..n]
  }
```

proof requires 16 lemmas

Hunting for opportunities

Hunting for opportunities

- Consider unclear or difficult code
- Consider common runtime checks

Consider common runtime checks

Consider common runtime checks

```
(Data.Array.!) :: Ix i => Array i e -> i -> e
```

Consider common runtime checks

```
(Data.Array.!) :: Ix i => Array i e -> i -> e
```

```
unsafeAt
  :: Ix i
  => a:Array i e
  -> { x:i | inRange (bounds a) x }
  -> e
```

Consider type indices

Consider type indices

```
int i :: Expr TInt
char c :: Expr TChar
toUpper :: Expr (TChar :-> TChar)
```

Consider type indices

```
int i :: Expr TInt
char c :: Expr TChar
toUpper :: Expr (TChar :-> TChar)
```

```
int i :: { e:Expr | typeOf e = TInt }
char c :: { e:Expr | typeOf e = TChar }
toUpper :: { e:Expr | typeOf e = TChar :-> TChar }
```

Resist full verification

- Benefit from checks that work out-of-the-box
- Assume freely

Engage with Liquid Haskellers

- Share problems, Learn, and Help

RESOURCES

- Specs, proofs, and documentation of `Data.List.permutations`
  `https://github.com/ucsd-progsys/liquidhaskell/blob/develop/tests/ple/pos/Permutations.hs`
- Reaching to Liquid Haskellers
  `https://github.com/ucsd-progsys/liquidhaskell#ask-for-help`
- A comparison with "Dependent" Haskell
  `https://www.tweag.io/blog/2022-01-19-why-liquid-haskell/`
- Liquid Haskell: Theorem Proving for All
  Niki Vazou
  Haskell Exchange 2018
  `https://skillsmatter.com/skillscasts/11068-keynote-looking-forward-to-niki-vazou-s-keynote-at-haskellx-2018`