Theorem Proving for All

Haskell'18

by Niki Vazou, Joachim Breitner, Rose Kunkel,
David Van Horn, and Graham Hutton

Theorem Proving for All

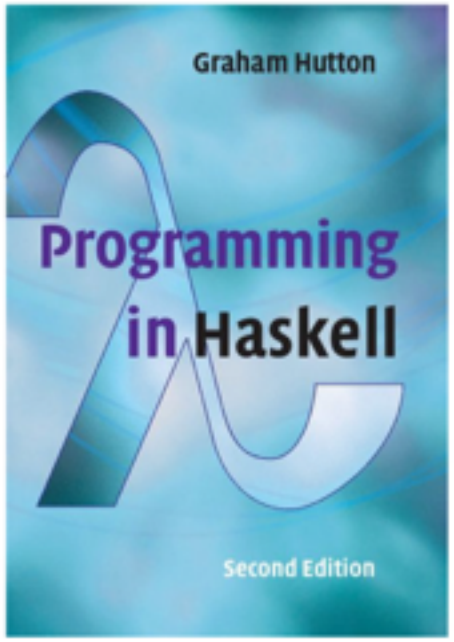
Haskell Programmers

by Niki Vazou, Joachim Breitner, Rose Kunkel,
David Van Horn, and Graham Hutton

16

Reasoning about programs

In this chapter we introduce the idea of reasoning about Haskell programs. We start by reviewing the notion of equational reasoning, then consider how it can be applied in Haskell, introduce the important technique of induction, show how induction can be used to eliminate uses of the append operator, and conclude by proving the correctness of a simple compiler.



But, reasoning is in pen-and-paper :(

Can we do it in Haskell?

Theorem: For any list x, `reverse [x] = [x]`

Proof.

```
      reverse [x]
-   applying reverse on [x]
=     reverse [] ++ [x]
-   applying reverse on []
=     [] ++ [x]
-   applying ++ on [] and [x]
=     [x]
      QED
```

Proof is not machine checked.

Theorem: For any list x, `reverse [x] = [x]`

Proof.

```
      reverse [x]

–   obviously!

=     [x]
      QED
```

Proof is not machine checked.

Theorem: For any list x, `reverse [x] = [x]`

Proof.

```
        reverse [x]
  -   applying reverse on [x]
  =     reverse [] ++ [x]
  -   applying reverse on []
  =     [] ++ [x]
  -   applying ++ on [] and [x]
  =     [x]
        QED
```

Proof is not machine checked.
Proof- and Haskell-reverse may be different!

Theorem: For any list x, `reverse [x] = [x]`

Proof.

```
      reverse [x]
-   applying reverse on [x]
=     reverse [] ++ [x]
-   applying reverse on []
=     [] ++ [x]
-   applying ++ on [] and [x]
=     [x]
      QED
```

Proof as a Haskell function

Theorem: For any list x, `reverse [x] = [x]`

```
singletonP x
    =   reverse [x]
    -- applying reverse on [x]
    =   reverse [] ++ [x]
    -- applying reverse on []
    =   [] ++ [x]
    -- applying ++ on [] and [x]
    =   [x]
        QED
```

Proof as a Haskell function

Theorem: For any list x, `reverse [x] = [x]`

```
singletonP x
    =   reverse [x]
    -  applying reverse on [x]
    =   reverse [] ++ [x]
    -  applying reverse on []
    =   [] ++ [x]
    -  applying ++ on [] and [x]
    =   [x]
        QED
```

Theorem: For any list x, `reverse [x] = [x]`

```
singletonP x
    =   reverse [x]
    –  applying reverse on [x]
    =   reverse [] ++ [x]
    –  applying reverse on []
    =   [] ++ [x]
    –  applying ++ on [] and [x]
    =   [x]
        QED
```

How to encode equality?

Equational Operator in (Liquid) Haskell

checks both arguments are equal

```
(==.) :: x:a              x == y }
      -> {v:              && v == y }
x ==. y = y
```

This introduces 3 equals and refinement types, refine it!

- explain ole more
- format pen&pencil proof with different

returns 2nd argument,

to continue the proof!

Theorem: For any list x, `reverse [x] = [x]`

```
singletonP x
    =   reverse [x]
    –  applying reverse on [x]
    ==. reverse [] ++ [x]
    –  applying reverse on []
    =   [] ++ [x]
    –  applying ++ on [] and [x]
    =   [x]
        QED
```

Theorem: For any list x, `reverse [x] = [x]`

```
singletonP x
    =   reverse [x]
    -   applying reverse on [x]
    ==. reverse [] ++ [x]
    -   applying reverse on []
    ==. [] ++ [x]
    -   applying ++ on [] and [x]
    ==. [x]
        QED
```

Theorem: For any list x, `reverse [x] = [x]`

```
singletonP x
    =   reverse [x]
    -   applying reverse on [x]
    ==. reverse [] ++ [x]
    -   applying reverse on []
    ==. [] ++ [x]
    -   applying ++ on [] and [x]
    ==. [x]
    QED
```

How to encode QED?

Define QED as data constuctor…

```
data QED = QED
```

… that casts anything into a proof
(i.e., a unit value).

```
(***) :: a -> QED -> ()
_ *** QED = ()
```

Theorem: For any list x, reverse [x] = [x]

```
singletonP x
    =   reverse [x]
    -   applying reverse on [x]
    ==. reverse [] ++ [x]
    -   applying reverse on []
    ==. [] ++ [x]
    -   applying ++ on [] and [x]
    ==. [x]
    *** QED
```

Theorem: For any list x, `reverse [x] = [x]`

```
singletonP x
    =   reverse [x]
    -  applying reverse on [x]
    ==. reverse [] ++ [x]
    -  applying reverse on []
    ==. [] ++ [x]
    -  applying ++ on [] and [x]
    ==. [x]
    *** QED
```

Proofs are functions

Theorems are types
Proofs are functions

— Curry & Howard

Theorems are types

Theorem:

For any list x, `reverse [x] = [x]`

Type:

```
x:a → { v:() | reverse [x] = [x] }
```

Theorems are types

Theorem:

For any list x, `reverse [x] = [x]`

Type:

```
x:a → { reverse [x] = [x] }
```

Theorem Proving in Haskell

```
singletonP :: x:a → { reverse [x] = [x] }
singletonP x
    =   reverse [x]
    -  applying reverse on [x]
    ==. reverse [] ++ [x]
    -  applying reverse on []
    ==. [] ++ [x]
    -  applying ++ on [] and [x]
    ==. [x]
    *** QED
```

Theorems are Types

```
singletonP :: x:a → { reverse [x] = [x] }
```

Theorem Application is Function Call

```
singletonP 1 :: { reverse [1] = [1] }
```

Theorem Application is Function Call

```
singletonP1 :: { reverse [1] = [1] }
singletonP1
  =   reverse [1]
    ? singletonP 1
  ==. [1]
  *** QED

                    (?) :: a -> () -> a
                    x ? _ = x
```

Theorem Proving for All

Reasoning about Haskell Programs in Haskell!

Equational operators (`==.`, `?`, `QED`, `***`) let us encode proofs as Haskell functions checked by Liquid Haskell.

Theorem Proving for All

Reasoning about Haskell Programs in Haskell!

How to encode inductive proofs?

Theorem: For any list x, reverse [x] = [x].

Proof.

Base Case:

```
    reverse (reverse [])
    – applying inner reverse
=   reverse []
    – applying reverse
=   []
    QED
```

Inductive Case:

```
    reverse (reverse (x:xs))
  – applying inner reverse
=   reverse (reverse xs ++ [x])
  – distributivity on (reverse xs) [x]
=   reverse [x] ++ reverse (reverse xs)
  – involution on xs
=   reverse [x] ++ xs
  – singleton on x
=   [x] ++ xs
  – applying ++
=   x:([] ++ xs)
  – applying ++
=   (x:xs)
    QED
```

Step 1: Define a recursive function!

Theorem: For any list x, reverse [x] = [x].

Proof.

```
involutionP []
=   reverse (reverse [])
    – applying inner reverse
=   reverse []
    – applying reverse
=   []
    QED
```

```
involutionP (x:xs)
=   reverse (reverse (x:xs))
    – applying inner reverse
=   reverse (reverse xs ++ [x])
    – distributivity on (reverse xs) [x]
=   reverse [x] ++ reverse (reverse xs)
    – involution on xs
=   reverse [x] ++ xs
    – singleton on x
=   [x] ++ xs
    – applying ++
=   x:([] ++ xs)
    – applying ++
=   (x:xs)
    QED
```

Step 1: Define recursive function

Step 2: Use equational reasoning!

Theorem: For any list x, reverse [x] = [x].

Proof.

```
involutionP []
==. reverse (reverse [])
    – applying inner reverse
==. reverse []
    – applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
    – applying inner reverse
==. reverse (reverse xs ++ [x])
    – distributivity on (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
    – involution on xs
==. reverse [x] ++ xs
    – singleton on x
==. [x] ++ xs
    – applying ++
==. x:([] ++ xs)
    – applying ++
==. (x:xs)
*** QED
```

Step 2: Use equational operators!

Step 3: Lemmata are function calls!

Theorem: For any list x, reverse [x] = [x].

Proof.

```
involutionP []
==. reverse (reverse [])
    - applying inner reverse
==. reverse []
    - applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
    - applying inner reverse
==. reverse (reverse xs ++ [x])
    ? distributivityP (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
    ? involutionP xs
==. reverse [x] ++ xs
    ? singletonP x
==. [x] ++ xs
    - applying ++
==. x:([] ++ xs)
    - applying ++
==. (x:xs)
*** QED
```

Step 3: Lemmata are function calls!

Theorem: For any list x, reverse [x] = [x].

Proof.

```
involutionP []
==. reverse (reverse [])
    - applying inner reverse
==. reverse []
    - applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
   - applying inner reverse
==. reverse (reverse xs ++ [x])
   ? distributivityP (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
   ? involutionP xs
==. reverse [x] ++ xs
   ? singletonP x
==. [x] ++ xs
   - applying ++
==. x:([] ++ xs)
   - applying ++
==. (x:xs)
*** QED
```

Note: Inductive hypothesis is recursive call!

Theorem: For any list x, reverse [x] = [x].

Proof.

```
involutionP []
==. reverse (reverse [])
    - applying inner reverse
==. reverse []
    - applying reverse
==. []
*** QED
```

```
involutionP (x:xs)
==. reverse (reverse (x:xs))
    - applying inner reverse
==. reverse (reverse xs ++ [x])
    ? distributivityP (reverse xs) [x]
==. reverse [x] ++ reverse (reverse xs)
    ? involutionP xs
==. reverse [x] ++ xs
    ? singletonP x
==. [x] ++ xs
    - applying ++
==. x:([] ++ xs)
    - applying ++
==. (x:xs)
*** QED
```

Question: Is the proof well founded?

Theorem Proving for All

Formally Reason about Haskell Programs in Haskell!

Application: Function Optimization

Theorem Proving for All

Formally Reason about Haskell Programs in Haskell!

Application ptimization

Not in the paper: future of liquid

In the paper

Case Study: Tree Optimizations

Case Study: Compiler Correctness

Comparison with other Theorem Provers

Theorem Proving for All

Formally Reason about Haskell Programs in Haskell!

Application: Function Optimization

Online demo:

http://goto.ucsd.edu/~nvazou/theorem-proving-for-all/

Thanks!