Exploring Languages with Interpreters and Functional Programming Chapter 25

H. Conrad Cunningham

11 April 2022

Contents

25 Proving Haskell Laws **2**
25.1 Chapter Introduction 2
25.2 Referential Transparency Revisited 2
25.3 Stating and Proving Laws 2
25.3.1 Example: `++` associativity and identity element 2
25.3.2 Structural induction proof method 3
25.3.3 Proving associativity of `++` 4
25.3.4 Reviewing proof method 6
25.3.5 Proving identity element for `++` 7
25.4 Example: Relating `length` and `++` 8
25.5 Example: Relating `take` and `drop` 9
25.6 Example: Equivalence of Functions 10
25.7 What Next? 13
25.8 Exercises 13
25.9 Acknowledgements 16
25.10Terms and Concepts 16
25.11References 17

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser

that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

25 Proving Haskell Laws

25.1 Chapter Introduction

The goal of this chapter is to show how to state and prove Haskell "laws".

This chapter depends upon the reader understanding Haskell's polymorphic, higher-order list programming concepts (e.g., from Chapters 4-5, 8-9, and 13-17), but it is otherwise independent of other preceding chapters.

The chapter provides useful tools that can be used in stating and formally proving function and module contracts (Chapters 6, 7, and 22) and type class laws (Chapter 23). It supports reasoning about program generalization (Chapter 19) and type inference (Chapter 24).

The following two chapters on program synthesis (Chapters 26 and 27) build on the concepts and techniques introduced by this chapter.

25.2 Referential Transparency Revisited

Referential transparency is probably the most important property of purely functional programming languages like Haskell.

Chapter 2 defines referential transparency to mean that, within some well-defined context, a variable (or other symbol) always represents the same value. This allows one expression to be replaced by an equivalent expression or, more informally, "equals to be replaced by equals".

Chapter 8 shows how referential transparency underpins the evaluation (i.e., substitution or reduction) model for Haskell and similar functional languages.

In this chapter, we see that referential transparency allows us to state and prove various "laws" or identities that hold for functions and to use these "laws" to transform programs into equivalent ones. Referential transparency underlies how we reason about Haskell programs.

25.3 Stating and Proving Laws

As a purely functional programming language, Haskell supports mathematical reasoning mostly within the programming language itself. We can state properties of functions and prove them using a primarily equational, or calculational, style of proof. The proof style is similar to that of high school trigonometric identities.

25.3.1 Example: ++ associativity and identity element

We have already seen a number of these laws. Again consider the append operator (++) for *finite lists* from Chapter 14.

```
infixr 5 ++
```

```
(++) :: [a] -> [a] -> [a]
[] ++ xs     = xs             -- append.1
(x:xs) ++ ys = x:(xs ++ ys)   -- append.2
```

The append operator ++: has two useful properties that we have already seen.

Associativity: For any finite lists `xs`, `ys`, and `zs`,
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.
Identity: For any finite list `xs`,
`[] ++ xs = xs = xs ++ []`.

Note: The above means that the append operator ++ and the set of finite lists form the algebraic structure called a *monoid*.

How do we prove these properties?

25.3.2 Structural induction proof method

The answer is, of course, *induction*. But we need a type of induction that allows us to prove theorems over the set of all finite lists. In fact, we have already been using this form of induction in the informal arguments that the list-processing functions terminate.

Induction over the natural numbers is a special case of a more general form of induction called *structural induction*. This type of induction is over the syntactic structure of recursively (inductively) defined objects. Such objects can be partially ordered by a complexity ordering from the most simple (minimal) to the more complex.

If we think about the usual axiomization of the natural numbers (i.e., Peano's postulates), then we see that `0` is the only simple (minimal) object and that the successor function `((+) 1)` is the only constructor.

In the case of finite lists, the only simple object is the nil list `[]` and the only constructor is the cons operator `(:)`.

To prove a proposition `P(x)` holds for any finite object `x`, one must prove the following cases.

Base cases: That `P(e)` holds for each simple (minimal) object `e`.

Inductive cases: That, for all object constructors `C`, if `P(x)` holds for some arbitrary object(s) `x`, then `P(C(x))` also holds.

That is, we can *assume* `P(x)` holds, then *prove* that `P(C(x))` holds. This shows that the constructors preserve proposition 'P.

To prove a proposition `P(xs)` holds for any finite list `xs`, the above reduces to the following cases.

Base case `xs = []`: That `P([])` holds.

Inductive case `xs = (a:as)`. That, if `P(as)` holds, then `P(a:as)` also holds.

One, often useful, strategy for discovering proofs of laws is the following:

- Determine whether induction is needed to prove the law. Some laws can be proved directly from the definitions and other previously proved laws.
- Carefully choose the induction variable (or variables).
- Identify the base and inductive cases.
- For each case, use *simplification* independently on each side of the equation. Often, it is best to start with the side that is the most complex.

  Simplification means to substitute the right-hand side of a *definition* or the induction hypothesis for some expression matching the left-hand side.
- Continue simplifying each expression as long as possible.

  Often we can show that the two sides of an equation are the same or that simple manipulations (perhaps using previously proved laws) will show that they are the same.
- If necessary, identify subcases and prove each subcase independently.

A formal proof of a case should, in general, be shown as a calculation that transforms one side of the equation into the other by substitution of equals for equals.

This formal proof can be constructed from the calculation suggested in the above

25.3.3 Proving associativity of ++

Now that we have the mathematical machinery we need, let's prove that ++ is associative for all finite lists. The following proofs assume that all arguments of the functions are defined.

Prove: For any finite lists `xs`, `ys`, and `zs`,
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.

Proof:

There does not seem to be a non-inductive proof, thus we proceed by structural induction over the finite lists. But on which variable(s)?

By examining the definition of ++, we see that it has two legs differentiated by the value of the left operand. The right operand is not decomposed. To use this definition in the proof, we need to consider the left operands of the ++ in the associative law.

Thus we choose to do the induction on `xs`, the leftmost operand, and consider two cases—a base case and an inductive case.

Base case `xs = []`:

First, we simplify the left-hand side.

```
  [] ++ (ys ++ zs)
```

= { `append.1` (left to right), omit outer parentheses }

```
  ys ++ zs
```

We do not know anything about `ys` and `zs`, so we cannot simplify further.

Next, we simplify the right-hand side.

```
  ([] ++ ys) ++ zs
```

= { `append.1` (left to right), omit parentheses around `ys` }

```
  ys ++ zs
```

Thus we have simplified the two sides to the same expression.

Of course, a formal proof can be written more elegantly as:

```
  [] ++ (ys ++ zs)
```

= { `append.1` (left to right) }

```
  ys ++ zs
```

= { `append.1` (right to left, applied to left operand) }

```
  ([] ++ ys) ++ zs
```

Thus the base case is established.

Note the equational style of reasoning. We proved that one expression was equal to another by beginning with one of the expressions and repeatedly substituting "equals for equals" until we got the other expression.

Each transformational step was justified by a definition, a known property, or (as we see later) the induction hypothesis. We normally do not state justifications like "omit parentheses" or "insert parentheses". We show these justifications for these steps in braces in the equational arguments. This style follows the common practice in the program derivaton community [7,7,13].

In the inductive case, we find it helpful to state both the inductive assumption and the proof goal explicitly, as we do below.

Inductive case `xs = (a:as)`:

Assume `as ++ (ys ++ zs) = (as ++ ys) ++ zs`;
prove `(a:as) ++ (ys ++ zs) = ((a:as) ++ ys) ++ zs`.

First, we simplify the left-hand side.

```
  (a:as) ++ (ys ++ zs)
```

= { `append.2` (left to right) }

```
    a:(as ++ (ys ++ zs))
```

= { induction hypothesis }

```
    a:((as ++ ys) ++ zs)
```

We do not know anything further about `as`, `ys`, and `zs`, so we cannot simplify further.

Next, we simplify the right-hand side.

```
    ((a:as) ++ ys) ++ zs
```

= { `append.2` (left to right, on inner ++) }

```
    (a:(as ++ ys)) ++ zs
```

= { `append.2` (left to right, on outer ++) }

```
    a:((as ++ ys) ++ zs)
```

Thus we have simplified the two sides to the same expression.

Again, a formal proof can be written more elegantly as follows.

```
    (a:as) ++ (ys ++ zs)
```

= { `append.2` (left to right) }

```
    a:(as ++ (ys ++ zs))
```

= { induction hypothesis }

```
    a:((as ++ ys) ++ zs)
```

= { `append.2` (right to left, on outer ++) }

```
    (a:(as ++ ys)) ++ zs
```

= { `append.2` (right to left, on inner ++) }

```
    ((a:as) ++ ys) ++ zs
```

Thus the inductive case is established.

Therefore, we have proven the ++ associativity property. **Q.E.D.**

The above proof and the ones that follow assume that the arguments of the functions are all defined (i.e., not equal to $\perp$).

25.3.4 Reviewing proof method

You should practice writing proofs in the "more elegant" form given above. This end-to-end calculational style is more useful for synthesis of programs.

Reviewing what we have done, we can identify the following guidelines:

- Determine whether induction is really needed.

- Choose the induction variable carefully.
- Be careful with parentheses.

  Substitutions, comparisons, and pattern matches must done with the fully parenthesized forms of definitions, laws, and expressions in mind, that is, with parentheses around all binary operations, simple objects, and the entire expression. We often omit "unneeded" parentheses to make the expression more readable.
- Start with the more complex side of the equation.

  That gives us more information with which to work.

25.3.5 Proving identity element for ++

Now let's prove the identity property.

Prove: For any finite list `xs`,
`[] ++ xs = xs = xs ++ []`.

Proof:

The equation `[] ++ xs = xs` follows directly from `append.1`. Thus we consider the equation `xs ++ [] = xs`, which we prove by structural induction on `xs`.

Base case `xs = []`:

```
  [] ++ []
```

= { `append.1` (left to right) }

```
  []
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `as ++ [] = as`; *prove* `(a:as) ++ [] = (a:as)`.

```
  (a:as) ++ []
```

= { `append.2` (left to right) }

```
  a:(as ++ [])
```

= { induction hypothesis }

```
  a:as
```

This establishes the inductive case.

Therefore, we have proved that `[]` is the *identity element* for `++`. **Q.E.D.**

25.4 Example: Relating `length` and `++`

Suppose that the list `length` function is defined as follows (from Chapter 13}).

```
length :: [a] -> Int
length []     =  0              -- length.1
length (_:xs) =  1 + length xs  -- length.2
```

Prove: For all finite lists `xs` and `ys`:
`length (xs++ys) = length xs + length ys`.

Proof:

Because of the way ++ is defined, we choose `xs` as the induction variable.

Base case `xs = []`:

```
    length [] + length ys
```

= { `length.1` (left to right) }

```
    0 + length ys
```

= { 0 is identity for addition }

```
    length ys
```

= { `append.1` (right to left) }

```
    length ([] ++ ys)
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `length (as ++ ys) = length as + length ys`;
prove `length ((a:as) ++ ys) = length (a:as) + length ys`.

```
    length ((a:as) ++ ys)
```

= { `append.2` (left to right) }

```
    length (a:(as ++ ys))
```

= { `length.2` (left to right) }

```
    1 + length (as ++ ys)
```

= { induction hypothesis }

```
    1 + (length as + length ys)
```

= { associativity of addition }

```
    (1 + length as) + length ys
```

= { `length.2` (right to left, value of `a` arbitrary) }

```
    length (a:as) + length ys
```

This establishes the inductive case.

Therefore, `length (xs ++ ys) = length xs + length ys`. **Q.E.D.**

Note: The proof above uses the associativity and identity properties of integer addition.

25.5 Example: Relating `take` and `drop`

Remember the definitions for the list functions `take` and `drop` from Chapter 13}.

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []                -- take.1
take _ []         = []                -- take.2
take n (x:xs)     = x : take (n-1) xs -- take.3

drop :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs               -- drop.1
drop _ []          = []               -- drop.2
drop n (_:xs)      = drop (n-1) xs    -- drop.3
```

Prove: For any natural numbers `n` and finite lists `xs`,
`take n xs ++ drop n xs = xs`.

Proof:

Note that both `take` and `drop` use both arguments to distinguish the cases. Thus we must do an induction over all natural numbers `n` and all finite lists `xs`.

We would expect four cases to consider, the combinations from `n` being zero and nonzero and `xs` being nil and non-nil. But an examination of the definitions for the functions reveal that the cases for `n = 0` collapse into a single case.

Base case `n = 0`:

```
    take 0 xs ++ drop 0 xs
```

= { `take.1`, `drop.1` (both left to right) }

```
    [] ++ xs
```

= { ++ identity `xs` }

```
    xs
```

This establishes the case.

Base case `n = m+1`, `xs = []`:

```
    take (m+1) [] ++ drop (m+1) []
```

= { `take.2`, `drop.2` (both left to right) }

```
    [] ++ []
```

= { ++ identity }

```
[]
```

This establishes the case.

Inductive case `n = m+1, xs = (a:as)`:

Assume `take m as ++ drop m as = as`;
prove `take (m+1) (a:as) ++ drop (m+1) (a:as) = (a:as)`.

```
take (m+1) (a:as) ++ drop (m+1) (a:as)
```

= { `take.3`, `drop.3` (both left to right) }

```
(a:(take m as)) ++ drop m as
```

= { `append.2` (left to right) }

```
a:(take m as ++ drop m as)
```

= { induction hypothesis }

```
(a:as)
```

This establishes the case.

Therefore, the property is proved. **Q.E.D.**

25.6 Example: Equivalence of Functions

What do we mean when we say two functions are equivalent?

Usually, we mean that the "same inputs" yield the "same outputs". For example, single argument functions `f` and `g` are equivalent if `f x = g x` for all `x`.

In Chapter 14. we defined two versions of a function to reverse the elements of a list. Function `rev` uses backward recursion and function `reverse` (called `reverse'` in Chapter 14) uses a forward recursive auxiliary function `rev'`.

```
rev :: [a] -> [a]
rev []      =  []                          -- rev.1
rev (x:xs)  =  rev xs ++ [x]               -- rev.2

reverse :: [a] -> [a]
reverse xs  =  rev' xs []                  -- reverse.1
    where rev' [] ys     = ys              -- reverse.2
          rev' (x:xs) ys = rev' xs (x:ys)  -- reverse.3
```

To show `rev` and `reverse` are equivalent, we must prove that, for all finite lists `xs`:

```
rev xs = reverse xs
```

If we unfold (i.e., simplify) `reverse` one step, we see that we need to prove:

```
rev xs = rev' xs []
```

Thus let's try to prove this by structural induction on `xs`.

Base case `xs = []`:

```
  rev []
```

= { `rev.1` (left to right) }

```
  []
```

= { `reverse.2` (right to left) }

```
  rev' [] []
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `rev as = rev' as []`; *prove* `rev (a:as) = rev' (a:as) []`.

First, we simplify the left side.

```
  rev (a:as)
```

= { `rev.2` (left to right) }

```
  rev as ++ [a]
```

Then, we simplify the right side.

```
  rev' (a:as) []
```

= { `reverse.3` (left to right) }

```
  rev' as [a]
```

Thus we need to show that `rev as ++ [a] = rev' as [a]`. But we do not know how to proceed from this point.

Maybe another induction. But that would probably just bring us back to a point like this again. We are stuck!

Let's look back at `rev xs = rev' xs []`. This is difficult to prove directly. Note the asymmetry, one argument for `rev` versus two for `rev'`.

Thus let's look for a new, more symmetrical, problem that might be easier to solve. Often it is easier to find a solution to a problem that is symmetrical than one which is not.

Note the place we got stuck above (proving `rev as ++ [a] = rev' as [a]`) and also note the equation `reverse.3`. Taking advantage of the identity element for `++`, we can restate our property in a more symmetrical way as follows:

```
rev xs ++ [] = rev' xs []
```

Note that the constant `[]` appears on both sides of the above equation. We can now apply the following generalization heuristic [8,13]. (That is, we try to solve a "harder" problem.)

Heuristic: *Replace constant by variable*

> That is, generalize by replacing a constant (or any subexpression) by a new variable.

Thus we try to prove the more general proposition:

```
rev xs ++ ys = rev' xs ys
```

The case `ys = []` gives us what we really want to hold. Intuitively, this new proposition seems to hold. Now let's prove it formally. Again we try structural induction on `xs`.

Base case `xs = []`:

```
    rev [] ++ ys
```

= { `rev.1` (left to right) }

```
    [] ++ ys
```

= { `append.1` (left to right) }

```
    ys
```

= { `reverse.2` (right to left) }

```
    rev' [] ys
```

This establishes the base case.

Inductive case `xs = (a:as)`:

Assume `rev as ++ ys = rev' as ys` for any finite list `ys`; *prove* `rev (a:as) ++ ys = rev' (a:as) ys`.

```
    rev (a:as) ++ ys
```

= { `rev.2` (left to right) }

```
    (rev as ++ [a]) ++ ys
```

= { ++ associativity, Note 1 }

```
    rev as ++ ([a] ++ ys)
```

= { singleton law, Note 2 }

```
    rev as ++ (a:ys)
```

= { induction hypothesis }

```
    rev' as (a:ys)
```

= { `reverse.3` (right to left) }

```
rev' (a:as) ys
```

This establishes the inductive case.

Notes:

1. We could apply the induction hypothesis here, but it does not seem profitable. Keeping the expressions in terms of `rev` and `++` as long as possible seems better; we know more about those expressions.
2. The *singleton law* is `[x] ++ xs = x:xs` for any element `x` and finite list `xs` of the same type. Proof of this is left as an exercise for the reader.

Therefore, we have proved `rev xs ++ ys = rev' xs ys` and, hence:

```
rev xs = reverse xs
```

The key to the performance improvement here is the solution of a "harder" problem: function `rev'` does both the reversing and appending of a list while `rev` separates the two actions.

25.7 What Next?

This chapter illustrated how to state and prove Haskell "laws" about already defined functions.

Chapters 26} and 27} on *program synthesis* illustrate how to use similar reasoning methods to synthesize (i.e., derive or calculate) function definitions from their specifications.

25.8 Exercises

This set of exercises uses functions defined in this and previous chapters including the following:

- Functions `map`, `filter`, `foldr`, `foldl`, and `concatMap` are defined in Chapter 15.
- Functional composition, identity combinator `id`, and function `all` are defined in Chapter 16}.
- Functions `takeWhile` and `dropWhile` are defined in Chapter 17.

Prove the following properties using the proof methods illustrated in this chapter.

1. Prove for all `x` of some type and finite lists `xs` of the same type (i.e., the *singleton law*):

   ```
   [x] ++ xs  =  (x:xs)
   ```

2. Consider the definition for `length` given in the text of this chapter and the following definition for `len`:

```
len :: Int -> [a] -> Int
len n [ ]    = n              -- len.1
len n (_:xs) = len (n+1) xs   -- len.2
```

Prove for any finite list `xs`: `len 0 xs = length xs`.

3. Prove for all finite lists `xs` and `ys` of the same type:

   ```
   reverse (xs ++ ys)  =  reverse ys ++ reverse xs
   ```

   Hint: The function `reverse` (called`reverse'` in Chapter 14.) uses forward recursion. Backward recursive definitions are generally easier to use in inductive proofs. In Chapter 14., we also defined a backward recursive function `rev` and proved that `rev xs = reverse xs` for all finite lists `xs`. Thus, you may find it easier to substitute `rev` for `reverse` and instead prove:

   ```
   rev (xs ++ ys)  =  rev ys ++ rev xs
   ```

4. Prove for all finite lists `xs` of some type:

   ```
   reverse (reverse xs)  =  xs
   ```

5. Prove for all natural numbers `m` and `n` and all finite lists `xs`:

   ```
   drop n (drop m xs)  =  drop (m+n) xs
   ```

6. Consider the rational number package from Chapter 7.. Prove for any `Rat` value `r` that satisfied the interface invariant for the abstract module `RationalRep`:

   ```
   addRat r zeroRat  =  r  =  addRat zeroRat r
   ```

7. Consider the two definitions for the Fibonacci function in Chapter 9. Prove for any natural number `n`:

   ```
   fib n  =  fib' n
   ```

   Hint: First prove, for `n` ≥ 2:

   ```
   fib'' n p q = fib'' (n-2) p q + fib'' (n-1) p q
   ```

8. Prove that the `id` function is the identity element for functional composition. That is, for any function `f :: a ->  b`, prove:

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

9. Prove that functional composition is associative. That is, for any function `f :: a -> a`, `g :: a -> a`, and `h :: a -> a`, prove:

   ```
   (f . g) . h  =  f . (g . h)
   ```

10. Prove for all finite lists `xs` and `ys` of the same type and function `f` on that type:

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

11. Prove for all finite lists `xs` and `ys` of the same type and predicate `p` on that type:

```
filter p (xs ++ ys)  =  filter p xs ++ filter p ys
```

12. Prove for all finite lists `xs` and `ys` of the same type and all predicates `p` on that type:

```
all p (xs ++ ys)  =  (all p xs) && (all p ys)
```

    The definition for `&&` is as follows:

```
(&&) :: Bool -> Bool -> Bool
False && x = False  -- second argument not evaluated
True  && x = x      -- second argument returned
```

13. Prove for all finite lists `xs` of some type and predicates `p` and `q` on that type:

```
filter p (filter q xs)  =  filter q (filter p xs)
```

14. Prove for all finite lists `xs` and `ys` of the same type and for all functions `f` and values `a` of compatible types:

```
foldr f a (xs ++ ys)  =  foldr f (foldr f a ys) xs
```

15. Prove for all finite lists `xs` of some type and all functions `f` and `g` of conforming types:

```
map (f . g) xs  =  (map f . map g) xs
```

16. Prove for all finite lists of finite lists `xss` of some base type and function `f` on that type:

```
map f (concat xss)  =  concat (map (map f) xss)
```

17. Prove for all finite lists `xs` of some type and functions `f` on that type:

```
map f xs  =  foldr ((:) .f) [] xs
```

18. Prove for all lists `xs` and predicates `p` on the same type:

```
takeWhile p xs ++ dropWhile p xs  =  xs
```

19. Prove that, if `***` is an associative binary operation of type `t -> t` with identity element `z` (i.e., a monoid), then:

```
foldr (***) z xs  =  foldl (***) z xs
```

20. Consider the Haskell type for the natural numbers given in an exercise in Chapter 21.

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

    For the functions defined in that exercise, prove the following:

    a. Prove that `intToNat` and `natToInt` are inverses of each other.

b. Prove that `Zero` is the (right and left) identity element for `addNat`.

c. Prove for any `Nat`s `x` and `y`:

```
addNat (Succ x) y  =  addNat x (Succ y)
```

d. Prove associativity of addition on `Nat`'s. That is, for any `Nat`s `x`, `y`, and `z`:

```
addNat x (addNat y z)  =  addNat (addNat x y) z
```

e. Prove commutativity of addition on `Nat`'s. That is, for any `Nat`s `x` and `y`:

```
addNat x y  =  addNat y x
```

25.9 Acknowledgements

In Summer 2018, I adapted and revised this chapter from Chapter 11 of my *Notes on Functional Programming with Haskell* [9].

These previous notes drew on the presentations in the first edition of the classic Bird and Wadler textbook [3] and other functional programming sources [1,2,15,17,18]. They were also influenced by my research, study, and teaching related to program specification, verification, derivation, and semantics [[4]; [5]; [6]; [7]; [8]; [10]; [11]; [12]; [13]; [14]; [16]; vanGesteren1990].

I incorporated this work as new Chapter 25, Proving Haskell Laws, in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I retired from the full-time faculty in May 2019. As one of my

post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

25.10 Terms and Concepts

Referential transparency, equational reasoning, laws, definition, simplification, calculation, associativity, identity, monoid, singleton law, equivalence of functions.

25.11 References

[1] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] Richard Bird. 2015. *Thinking functionally with Haskell* (First ed.). Cambridge University Press, Cambridge, UK.

[3] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[4] K. Mani Chandy and Jayadev Misra. 1988. *Parallel program design: A foundation*. Addison-Wesley, Boston, Massachusetts, USA.

[5] Edward Cohen. 1990. *Programming in the 1990's: An introduction to the calculation of programs*. Springer, New York, New York, USA.

[6] H. Conrad Cunningham. 1989. The shared dataspace approach to concurrent computation: The Swarm programming model, notation, and logic. PhD thesis. Washington University, Department of Computer Science, St. Louis, Missouri, USA.

[7] H. Conrad Cunningham. 2006. *A programmer's introduction to predicate logic*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci450/notes/haskell_notes.pdf

[8] H. Conrad Cunningham. 2006. *Notes on program semantics and derivation*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/reports/umcis-1994-02.pdf

[9] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/Notes_FP_Haskell/Notes_on_Functional_Programming_with_Haskell.pdf

[10] Edsger W. Dijkstra. 1976. Updating a sequential file. In *A discipline of programming*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 117--122.

[11] Edsger W. Dijkstra and Wim H. J. Feijen. 1988. *A method of programming*. Addison-Wesley, TBD.

[12] Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate calculus and program semantics*. Springer, New York, New York, USA.

[13] David Gries. 1981. *Science of programming*. Springer, New York, New York, USA.

[14] David Gries and Fred B. Schneider. 2013. *A logical approach to discrete math*. Springer, New York, New York, USA.

[15] Robert R. Hoogerwoord. 1989. The design of functional programs: A calculational approach. PhD thesis. Eindhoven Technical University, Eindhoven, The Netherlands.

[16] Anne Kaldewaij. 1990. *Programming: The derivation of algorithms.* Prentice Hall, New York, New York, USA.

[17] Simon Thompson. 1996. *Haskell: The craft of programming* (First ed.). Addison-Wesley, Boston, Massachusetts, USA.

[18] E. Peter Wentworth. 1990. *Introduction to functional programming using RUFL.* Rhodes University, Department of Computer Science, Grahamstown, South Africa.