Exploring Languages with Interpreters and Functional Programming
Chapter 80

H. Conrad Cunningham

20 February 2019

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of Feburary 2019 is a recent version of Firefox from Mozilla.

80 Review of Relevant Mathematics

80.1 Chapter Introduction

Students studying from this textbook should already have sufficient familiarity with the relevant mathematical concepts from the usual prerequisite courses. However, they may need to relate the mathematics with the programming constructs in functional programming.

The goal of this chapter is to review the mathematical concepts of functions and a few other mathematical concepts used in these notes. The concept of function in functional programming corresponds closely to the mathematical concept of function.

TODO: Add discussion of logic needed for specification and statement of laws?

80.2 Natural Numbers and Ordering

Several of the examples in these notes use natural numbers.

For this study, we consider the set of *natural numbers* N to consist of 0 and the positive integers.

Inductively, $n \in N$ if and only if one of the following holds

- $n = 0$
- There exists $m \in N$ such that $m = S(n)$

where S is the *successor* function, which returns the next element.

Furthermore,

- No element is the successor of more one other natural number.
- 0 is not the successor of any natural number. That is, it is the least (base) element.

The natural numbers thus form a *totally ordered* set in conjunction with the binary relation $\leq$ (less or equal). That is, the relation $\leq$ satisfies the following properties on set N:

- $n \leq n$ for all $n \in N$ (*reflexivity*)
- $m \leq n$ and $n \leq m$ implies $m = n$ (*antisymmetry*)
- $m \leq n$ and $n \leq p$ implies $m \leq p$ (*transitivity*)
- Either $m \leq n$ or $n \leq m$ for all $m, n \in N$ (*trichotomy*)

It is also a *partial ordering* because it satisfies the first three properties above.

For all $m, n \in N$, we can define the other ordering relations in terms of $=$, $\neq$, and $\leq$ as follows:

- $m < n$ (less) to mean $m \leq n$ and $m \neq n$. We say that m is smaller (or simpler) than n.
- $m > n$ (greater) to mean $n \leq m$ and $n \neq m$. We say that m is larger (or more complex) than n.
- $m \geq n$ (greater or equal) to mean the same as $n \leq m$

80.3 Functions

As we have studied in mathematics courses, a *function* is a mapping from a set A into a set B such that each element of A is mapped into a unique element of B.

- The set A (on which f is defined) is called the *domain* of f.
- The set of all elements of B into which f maps elements of A is called the *range* (or *codomain*) of f, and is denoted by $f(A)$.

If f is a function from A into B, then we write:

$$f : A \to B$$

We also write the equation

$$f(a) = b$$

to mean that the *value* (or *result*) from *applying* function f to an element $a \in A$ is an element $b \in B$.

If a function

$$f : A \to B$$

and $A \subseteq A'$, then we say that f is a *partial function* from A' to B and a *total function* from A to B. That is, there are some elements of A' on which f may be undefined.

80.4 Recursive Functions

Informally, a *recursive function* is a function defined using recursion.

In computing science, *recursion* is a method in which an "object" is defined in terms of smaller (or simpler) "objects" of the same type. A recursion is usually defined in terms of a recurrence relation.

A *recurrence relation* defines an "object" x_n as some combination of zero or more other "objects" x_i for $i < n$. Here $i < n$ means that i is smaller (or simpler) than n. If there is no smaller object, then n is a base object.

For example, consider a recursive function to compute the sum s of the first n natural numbers.

We can define a recurrence relation for s with the following equations:

$s(n) = 0$, *if* $n = 0$
$s(n) = n + s(n-1)$, *if* $n \geq 1$

For example, consider $s(3)$,

$s(3) = 3+s(2) = 3+(2+s(1)) = 3+(2+(1+s(0))) = 3+(2+(1+0)) =$
6

80.5 Mathematical Induction Natural Numbers

We can give two mathematical definitions of factorial, *fact* and *fact'*, that are equivalent for all natural number arguments.

We can define *fact* using the product operator as follows:

$fact(n) = \prod_{i=1}^{i=n} i$

We can also define the factorial function *fact'* with a *recursive* definition (or *recurrence relation*) as follows:

fact'$(n) = 1$, *if* $n = 0$
fact'$(n) = n \times$ *fact'*$(n-1)$, *if* $n \geq 1$

It is, of course, easy to see that the recurrence relation definition is equivalent to the previous definition. But how can we prove it?

To prove that the above definitions of the factorial function are equivalent, we can use *mathematical induction* over the natural numbers.

Mathematical induction: To prove a logical proposition $P(n)$ holds for any natural number n, we must show two things:

- For the *base case* $n = 0$, show that $P(0)$ holds.
- For the *inductive case* $n = m + 1$, show that, if $P(m)$ holds for some natural number m, then $P(m + 1)$ also holds.

The $P(m)$ assumption is called the *induction hypothesis*.

Now let's prove that the two definitions *fact* and *fact'* are equivalent.

Prove For all natural numbers n, *fact*(n) = *fact'*(n).

Base case $n = 0$**.**

---

	fact(0)
=	{ definition of *fact* (left to right) }
	$(\Pi i : 1 \leq i \leq 0 : i)$
=	{ empty range for Π, 1 is the identity element of $\times$ }
	1
=	{ definition of *fact'* (first leg, right to left) }

fact'(0)

Inductive case $n = m + 1$**.**
Given induction hypothesis $fact(m) = fact'(m)$, prove $fact(m+1) = fact'(m+1)$.

	$fact'(m+1)$
=	{ definition of *fact* (left to right) }
	$(\Pi i : 1 \leq i \leq m+1 : i)$
=	{ $m+1 > 0$, so $m+1$ term exists, split it out }
	$(m+1) \times (\Pi i : 1 \leq i \leq m : i)$
=	{ definition of *fact* (right to left) }
	$(m+1) \times fact(m)$
=	{ induction hypothesis }
	$(m+1) \times fact'(m)$
=	{ $m+1 > 0$, definition of *fact'* (second leg, right to left) }
	$fact'(m+1)$

Therefore, we have proved $fact(n) = fact'(n)$ for all natural numbers n. QED

In the inductive step above, we explicitly state the induction hypothesis and assertion we wish to prove in terms of a different variable name (m instead of n) than the original statement. This helps to avoid the confusion in use of the induction hypothesis that sometimes arises.

We use an equational style of reasoning. To prove that an equation holds, we begin with one side and prove that it is equal to the other side. We repeatedly "substitute equals for equal" until we get the other expression.

Each transformational step is justified by a definition, a known property of arithmetic, or the induction hypothesis.

The structure of this inductive argument closely matches the structure of the recursive definition of *fact'*.

What does this have to do with functional programming? Many of the functions we will define in these notes have a recursive structure similar to *fact'*. The proofs and program derivations that we do will resemble the inductive argument above.

Recursion, induction, and iteration are all manifestations of the same phenomenon.

80.6 Operations

A function

$$\oplus : (A \times A) \to A$$

is called a *binary operation on A*. We usually write binary operations in *infix* form:

$$a \oplus a'$$

We often call a two-argument function of the form

$$\oplus : (A \times B) \to C$$

a binary operation as well. We can write this two argument function in the equivalent *curried* form:

$$\oplus : A \to (B \to C)$$

The curried form shows a multiple-paramenter function in a form where the function takes the arguments one at a time, returning the resulting function with one fewer arguments.

Let $\oplus$ be a binary operation on some set A and x, y, and z be elements of A. We can define the following kinds of properties.

- Operation $\oplus$ is *closed* on A if and only if $x \oplus y \in A$ for any $x, y \in A$. That is, the operation is a total function on its domain.
- Operation $\oplus$ is *associative* if and only if $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for $x, y, z \in A$.
- Operation $\oplus$ is *commutative* (also called *symmetric*) if and only if $x \oplus y = y \oplus x$ for $x, y \in A$.
- An element e of set A is
  - a *left identity* of $\oplus$ if and only if $e \oplus x = x$ for any $x \in A$
  - a *right identity* of $\oplus$ if and only if $x \oplus e = x$ for any $x \in A$
  - an *identity* of $\oplus$ if and only if it is both a left and a right identity.

  An identity of an operation is called a *unit* of the operation.
- An element z of set A is
  - a *left zero* of $\oplus$ if and only if $z \oplus x = z$ for any $x \in A$
  - a *right zero* of $\oplus$ if and only if $x \oplus z = z$ for any $x \in A$
  - a *zero* of $\oplus$ if and only if it is both a right and a left zero
- If e is the identity of $\oplus$ and $x \oplus y = e$ for some x and y, then
  - x is a *left inverse* of y
  - y is a *right inverse* of x.

  Elements x and y are inverses of each other if $x \oplus y = e = y \oplus x$.
- An element x of set A is *idempotent* if $x \oplus x = x$.

If all elements of A are idempotent with respect to $\oplus$, then $\oplus$ is called *idempotent*.

For example, the addition operation + on natural numbers is closed, associative, and commutative and has the identity element 0. It has neither a left or right zero element and the only element with a left or right inverse is 0. If we consider the set of all integers, then all elements also have inverses.

Also, the multiplication operation * on natural numbers (or on all integers) is closed, associative, and commutative and has identity element 1 and zero element 0. Only value 1 has a left or right inverse.

However, the subtraction operation on natural numbers is not closed, associative, or commutative and has neither a left nor right zero. The value 0 is subtraction's right identity, but subtraction has no left identity. Each element is its own right and left inverse. If we consider all integers, then the operation is also closed.

Also, the "logical and" and "logical or" operations are idempotent with respect to the set of Booleans.

80.7 Algebraic Structures

An *algebraic structure* consists of a set of values, a set of one or more operations on those values, and properties (or "laws") of the operation on the set. We can characterize algebraic structures by the operations and their properties on the set of values.

If we focus on a binary operation $\oplus$ on a set A, then we can define various algebraic structures based on their properties.

- If $\oplus$ is closed on A, then then $\oplus$ and A form a *magma*.
- A magma in which $\oplus$ is an *associative* operation forms a *semigroup*.
- A semigroup in which $\oplus$ has an *identity* element forms a *monoid*.
- A monoid in which every element of A has an inverse forms a *group*.
- A monoid in which $\oplus$ is *commutative* forms a *commutative monoid* (or *Abelian monoid*).
- A group in which $\oplus$ is *commutative* forms an *Abelian group*.

For example, addition on natural numbers forms a commutative monoid and on integers forms an Abelian group.

Note: Above we describe a few common *group-like* algebraic structures, that is, algebras with one operation and one set. If we consider two operations on one set (e.g. $\oplus$ on $\otimes$), then we have various *ring-like* algebraic structures. By adding other operations, we have various other kinds of algebraic structures. If we

consider more than one set, then we moved from a *single-sorted* (or *first-order*) algebra to a *many-sorted* algebra.

80.8 Exercises

TODO: Add

80.9 Acknowledgements

I adapted and revised much of this work in Summer and Fall 2016 from Chapter 2 of my *Notes on Functional Programming with Haskell* [Cunningham 2014a].

In Summer and Fall 2017, I continued to develop this material as a part of Chapter 1, Fundamentals, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I reorganized and expanded the previous Fundamentals chapter into four chapters for the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. These are Chapter 1, Evolution of Programming Languages; Chapter 2, Programming Paradigms; Chapter 3, Object-based Paradigms; and Chapter 91, Review of Relevant Mathematics (this background chapter).

In Spring 2019, I expanded the discussion of algebraic structures a bit.

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.

80.10 References

TODO: Add any needed references

[**Cunningham 2014a**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

80.11 Terms and Concepts

TODO: Add