Notes on Functional Programming with Haskell (Chapters 1 and 2)

H. Conrad Cunningham
cunningham@cs.olemiss.edu

Multiparadigm Software Architecture Group
Department of Computer and Information Science
University of Mississippi
201 Weir Hall
University, Mississippi 38677 USA

Spring Semester 2015

H. Conrad Cunningham, D.Sc.
Professor and Chair
Department of Computer and Information Science
University of Mississippi
201 Weir Hall
University, Mississippi 38677
USA

cunningham@cs.olemiss.edu

Contents

1 INTRODUCTION

1.1 Course Overview

This is a course on functional programming.

As a course on *programming*, it emphasizes the analysis and solution of problems, the development of correct and efficient algorithms and data structures that embody the solutions, and the expression of the algorithms and data structures in a form suitable for processing by a computer. The focus is more on the human thought processes than on the computer execution processes.

As a course on *functional* programming, it approaches programming as the construction of definitions for (mathematical) functions and data structures. Functional programs consist of *expressions* that use these definitions. The execution of a functional program entails the evaluation of the expressions making up the program. Thus the course's focus is on problem solving techniques, algorithms, data structures, and programming notations appropriate for the functional approach.

This is not a course on functional programming *languages*. In particular, the course does not undertake an in-depth study of the techniques for implementing functional languages on computers. The focus is on the concepts for programming, not on the internal details of the technological artifact that executes the programs.

Of course, we want to be able to execute our functional programs on a computer and, moreover, to execute them efficiently. Thus we must become familiar with some concrete programming language and use an implementation of that language to execute our programs. To be able to analyze program efficiency, we must also become familiar with the basic techniques that are used to evaluate expressions. To be specific, this class will use a functional programming environment called GHC (Glasgow Haskell Compiler). GHC is distributed in a "batteries included" bundle called the the Haskell Platform . (That is, it bundles GHC with commonly used libraries and tools.) The language accepted by GHC is the "lazy" functional programming language Haskell 2010. A program processed by GHC evaluates expressions according to an execution model called *graph reduction*.

Being "practical" is not an overriding concern of this course. Although functional languages are increasing in importance, their use has not yet spread much beyond the academic and industrial research laboratories. While a student may take a course on C++ programming and then go out into industry and find a job in which the C++ knowledge and skills can be directly applied, this is not likely to occur with a course on functional programming.

However, the fact that functional languages are not broadly used does not mean that this course is impractical. A few industrial applications are being developed using various functional languages. Many of the techniques of functional programming

can also be applied in more traditional programming and scripting languages. More importantly, any time programmers learn new approaches to problem solving and programming, they become better programmers. A course on functional programming provides a novel, interesting, and, probably at times, frustrating opportunity to learn more about the nature of the programming task. Enjoy the semester!

1.2 Excerpts from Backus' 1977 Turing Award Address

This subsection contains excerpts from computing pioneer John Backus' 1977 ACM Turing Award Lecture published as article "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs [1]" (*Communications of the ACM*, Vol. 21, No. 8, pages 613–41, August 1978). Although functional languages like Lisp go back to the late 1950's, Backus's address did much to stimulate research community's interest in functional programming languages and functional programming.

Programming languages appear to be in trouble. Each successive language incorporates, with little cleaning up, all the features of its predecessors plus a few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. ... Each new language claims new and fashionable features, such as strong typing or structured control statements, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.

Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But there is a desperate need for a powerful methodology to help us think about programs, and no conventional language even begins to meet that need. In fact, conventional languages create unnecessary confusion in the way we think about programs. ...

In order to understand the problems of conventional programming languages, we must first examine their intellectual parent, the von Neumann computer. What is a von Neumann computer? When von Neumann and others conceived of it ... [in the 1940's], it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this ... concept.

In its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the

contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent form the store, then its address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must either be generated by a fixed rule (e.g., "add 1 to the program counter") or by an instruction that was sent through the tube, in which case its address must have been sent, and so on.

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. ...

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our ... old belief that there is only one kind of computer is the basis our our belief that there is only one kind of programming language, the conventional—von Neumann—language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem. [Note: Backus was one of the designers of Fortran and of Algol-60.]

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-at-time terms in much the same way the computer's bottleneck does.

Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements in the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control.

Our fixation on von Neumann languages has continued the primacy of the von Neumann computer, and our dependency on *it* has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures. ...

Note: In his Turing Award Address, Backus went on to describe FP, his proposal for a functional programming language. He argued that languages like FP would allow programmers to break out of the von Neumann bottleneck and find new ways of thinking about programming. Although languages like Lisp had been in existence since the late 1950's, the widespread attention given to Backus' address and paper stimulated new interest in functional programming to develop by researchers around the world.

Aside: Above Backus states that "the world of statements is a disorderly one, with few mathematical properties". Even in 1977 this was a bit overstated since Dijkstra's work on the weakest precondition calculus and other work on axiomatic semantics had already appeared. However, because of the referential transparency (discussed later) property of purely functional languages, reasoning can often be done in an equational manner within the context of the language itself. In contrast, the wp-calculus and other axiomatic semantic approaches must project the problem from the world of programming language statements into the world of predicate calculus, which is much more orderly.

1.3 Programming Language Paradigms

Reference: The next two subsections are based, in part, on Hudak's article "Conception, Evolution, and Application of Functional Programming Languages [4]" (*ACM Computing Surveys,* Vol. 21, No. 3, pages 359–411, September 1989).

Programming languages are often classified according to one of two different paradigms: imperative and declarative.

Imperative languages

A program in an imperative language has an *implicit state* (i.e., values of variables, program counters, etc.) that is modified (i.e., side-effected) by *constructs* (i.e., commands) in the source language.

As a result, such languages generally have an explicit notion of *sequencing* (of the commands) to permit precise and deterministic control of the state changes.

Imperative programs thus express *how* something is to be computed.

These are the "conventional" or "von Neumann languages" discussed by Backus. They are well suited to traditional computer architectures.

Most of the languages in existence today are in this category: Fortran, Algol, Cobol, Pascal, Ada, C, C++, Java, etc.

Declarative languages

A program in a declarative language has *no implicit* state. Any needed state information must be handled explicitly.

A program is made up of *expressions* (or terms) rather than commands.

Repetitive execution is accomplished by *recursion* rather than by sequencing.

Declarative programs express *what* is to be computed (rather than how it is to be computed).

Declarative programs are often divided into two types:

Functional (or applicative) languages

The underlying model of computation is the mathematical concept of a *function.*

In a computation a function is applied to zero or more arguments to compute a single result, i.e., the result is deterministic (or predictable).

Purely functional:	FP, Haskell, Miranda, Hope, Orwell
Hybrid languages:	Lisp, Scheme, SML
	(Scheme & SML have powerful declarative subsets)
Dataflow languages:	Id, Sisal

Relational (or logic) languages

The underlying model of computation is the mathematical concept of a *relation* (or a *predicate*).

A computation is the (nondeterministic) association of a group of values—with backtracking to resolve additional values.

Examples: Prolog (pure), Parlog, KL1

Note: Most Prolog implementations have imperative features such as the cut and the ability to assert and retract clauses.

1.4 Reasons for Studying Functional Programming

1. **Functional programs are easier to manipulate mathematically than imperative programs.**

   The primary reason for this is the property of *referential transparency*, probably the most important property of modern functional programming languages.

   Referential transparency means that, within some well-defined context, a variable (or other symbol) *always* represents the *same value*. Since a variable always has the same value, we can replace the variable in an expression by its value or vice versa. Similarly, if two subexpressions have equal values, we can replace one subexpression by the other. That is, "equals can be replaced by equals".

   Functional programming languages thus use the same concept of a variable that mathematics uses.

   On the other hand, in most imperative languages a variable represents an address or "container" in which values may be stored; a program may change the value stored in a variable by executing an assignment statement.

   Because of referential transparency, we can construct, reason about, and manipulate functional programs in much the same way we can any other mathematical expressions [2, 3]. Many of the familiar "laws" from high school algebra still hold; new "laws" can be defined and proved for less familiar primitives and even user-defined operators. This enables a relatively natural equational style of reasoning.

   For example, we may want to prove that a program meets its specification or that two programs are equivalent (in the sense that both yield the same "outputs" given the same "inputs").

   We can also construct and prove algebraic "laws" for functional programming. For example, we might prove that some operation (i.e., two-argument function) is commutative or associative or perhaps that one operation distributes over another.

   Such algebraic laws enable one program to be transformed into another equivalent program either by hand or by machine. For example, we might use the

laws to transform one program into an equivalent program that can be executed more efficiently.

2. **Functional programming languages have powerful abstraction mechanisms.**

   Speaking operationally, a function is an abstraction of a pattern of behavior.

   For example, if we recognize that a C or Pascal program needs to repeat the *same* operations for each member of a set of similar data structures, then we usually encapsulate the operations in a function or procedure. The function or procedure is an abstraction of the application of the operation to data structures of the given type.

   Now suppose instead that we recognize that our program needs to perform *similar*, but different, operations for each member of a set of similar data structures. Can we create an abstraction of the application of the similar operations to data structures of the given type?

   For instance, suppose we want to compute either the sum or the product of the elements of an array of integers. Addition and multiplication are similar operations; they are both associative binary arithmetic operations with identity elements.

   Clearly, C or Pascal programs implementing sums and products can go through the same pattern of operations on the array: initialize a variable to the identity element and then loop through the array adding or multiplying each element by the result to that point. Instead of having separate functions for each operation, why not just have one function and supply the operation as an argument?

   A function that can take functions as arguments or return functions as results is called a *higher-order function*. Most imperative languages do not fully support higher-order functions.

   However, in most functional programming languages functions are treated as *first class* values. That is, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions.

   Typically, functions in imperative languages are not treated as first-class values.

   The higher-order functions in functional programming languages enable very regular and powerful abstractions and operations to be constructed. By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

   A programmer needs to write fewer "lines of code" in a concise programming notation than in a verbose one. Thus the programmer should be able to complete the task in less time. Since, in general, a short program is easier to comprehend than a long one, a programmer is less likely to make an error in a short program than in a long one. Consequently, functional programming can lead to both increased programmer productivity and increased program reliability.

Caveat: Excessive concern for conciseness can lead to cryptic, difficult to understand programs and, hence, low productivity and reliability. Conciseness should not be an end in itself. The understandability and correctness of a program are more important goals.

Higher-order functions also increase the *modularity* of programs by enabling simple program fragments to be "glued together" readily into more complex programs [5].

3. **Functional programming enables new algorithmic approaches.**

   This is especially true for languages (like Haskell) that use what is called *lazy evaluation*.

   In a lazy evaluation scheme, the evaluation of an expression is deferred until the value of the expression is actually needed elsewhere in the computation. That is, the expression is evaluated on demand. This contrasts with what is called *eager evaluation* in which an expression is evaluated as soon as its inputs are available.

   For example, if eager evaluation is used, an argument (which may be an arbitrary expression) of a function call is evaluated before the body of the function. If lazy evaluation is used, the argument is not evaluated until the value is actually needed during the evaluation of the function body. If an argument's value is never needed, then the argument is expression is never evaluated.

   Why should we care? Well, this facility allows programmers to construct and use data structures that are conceptually unbounded or infinite in size. As long as a program never actually needs to inspect the entire structure, then a terminating computation is still possible.

   For example, we might define the list of natural numbers as a list beginning with 0, followed by the list formed by adding one to each element of the list of natural numbers.

   Lazy evaluation thus allows programmers to separate the data from the control. They can define a data structure without having to worry about how it is processed and they can define functions that manipulate the data structure without having to worry about its size or how it is created. This ability to separate the data from the control of processing enables programs to be highly modular [5].

   For example, we can define the list of even naturals by applying a function that filters out odd integers to the infinite list of naturals defined above. This definition has no operational control within it and can thus be combined with other functions in a modular way.

4. **Functional programming enables new approaches to program development.**

   As we discussed above, it is generally easier to reason about functional programs than imperative programs. It is possible to prove algebraic "laws" of functional programs that give the relationships among various operators in the language. We can use these laws to transform one program to another equivalent one.

   These mathematical properties also open up new ways to write programs.

   Suppose we want a program to break up a string of text characters into lines. Section 4.3 of the Bird and Wadler textbook [2] and Section **??** of these notes shows a novel way to construct this program.

   First, Bird and Wadler construct a program to do the *opposite* of what we want—to combine lines into a string of text. This function is very easy to write.

   Next, taking advantage of the fact that this function is the inverse of the desired function, they use the "laws" to manipulate this simple program to find its inverse. The result is the program we want!

5. **Functional programming languages encourage (massively) parallel execution.**

   To exploit a parallel processor, it must be possible to decompose a program into components that can be executed in parallel, assign these components to processors, coordinate their execution by communicating data as needed among the processors, and reassemble the results of the computation.

   Compared to traditional imperative programming languages, it is quite easy to execute components of a functional program in parallel [6]. Because of the referential transparency property and the lack of sequencing, there are no time dependencies in the evaluation of expressions; the final value is the same regardless of which expression is evaluated first. The nesting of expressions within other expressions defines the data communication that must occur during execution.

   Thus executing a functional program in parallel does not require the availability of a highly sophisticated compiler for the language.

   However, a more sophisticated compiler can take advantage of the algebraic laws of the language to transform a program to an equivalent program that can more efficiently be executed in parallel.

   In addition, frequently used operations in the functional programming library can be be optimized for highly efficient parallel execution.

   Of course, compilers can also be used to decompose traditional imperative languages for parallel execution. But it is not easy to find all the potential parallelism. A "smart" compiler must be used to identify unnecessary sequencing and find a safe way to remove it.

In addition to the traditional imperative programming languages, imperative languages have also been developed especially for execution on a parallel computer. These languages shift some of the work of decomposition, coordination, and communication to the programmer.

A potential advantage of functional languages over parallel imperative languages is that the functional programmer does not, in general, need to be concerned with the specification and control of the parallelism.

In fact, functional languages probably have the problem of too much potential parallelism. It is easy to figure out what can be executed in parallel, but it is sometimes difficult to determine what components should actually be executed in parallel and how to allocate them to the available processors. Functional languages may be better suited to the massively parallel processors of the future than most present day parallel machines.

6. **Functional programming is important in some application areas of computer science.**

   The artificial intelligence (AI) research community has used languages such as Lisp and Scheme since the 1960's. Some AI applications have been commercialized during the past two decades.

   Also a number of the specification, modeling, and rapid-prototyping languages that are appearing in the software engineering community have features that are similar to functional languages.

7. **Functional programming is related to computing science theory.**

   The study of functional programming and functional programming languages provides a good opportunity to learn concepts related to programming language semantics, type systems, complexity theory, and other issues of importance in the theory of computing science.

8. **Functional programming is an interesting and mind-expanding activity for students of computer science!?**

   Functional programming requires the student to develop a different perspective on programming.

1.5 Objections Raised Against Functional Programming

1. **Functional programming languages are inefficient toys!**

   This was definitely true in the early days of functional programming. Functional languages tended to execute slowly, require large amounts of memory, and have limited capabilities.

   However, research on implementation techniques has resulted in more efficient and powerful implementations today.

   Although functional language implementations will probably continue to increase in efficiency, they likely will never become as efficient as the implementations of imperative "von Neumann" languages are on traditional "von Neumann" architectures.

   However, new computer architectures may allow functional programs to execute competitively with the imperative languages on today's architectures. For example, computers based on the dataflow and graph reduction models of computation are more suited to execute functional languages than imperative languages.

   Also the ready availability of parallel computers may make functional languages more competitive because they more readily support parallelism than traditional imperative languages.

   Moreover, processor time and memory usage just aren't as important concerns as they once were. Both fast processors and large memories have become relatively inexpensive and readily available. Now it is common to dedicate one or more processors and several megabytes of memory to individual users of workstations and personal computers.

   As a result, the community can now afford to dedicate considerable computer resources to improving programmer productivity and program reliability; these are issues that functional programming may address better than imperative languages.

2. **Functional programming languages are not (and cannot be) used in the real world!**

   It is still true that functional programming languages are not used very widely in industry. But, as we have argued above, the functional style is becoming more important—especially as commercial AI applications have begun to appear.

   If new architectures like the dataflow machines emerge into the marketplace, functional programming languages will become more important.

   Although the functional programming community has solved many of the difficulties in implementation and use of functional languages, more research is needed on several issues of importance to the real world: on facilities for input/output, nondeterministic, realtime, parallel, and database programming.

More research is also needed in the development of algorithms for the functional paradigm. The functional programming community has developed functional versions of many algorithms that are as efficient, in terms of big-O complexity, as the imperative versions. But there are a few algorithms for which efficient functional versions have not yet been found.

3. **Functional programming is awkward and unnatural!**

   Maybe. It might be the case that functional programming somehow runs counter to the way that normal human minds work—that only mental deviants can ever become effective functional programmers. Of course, some people might say that about programming and programmers in general.

   However, it seems more likely that the awkwardness arises from the lack of education and experience. If we spend many years studying and doing programming in the imperative style, then any significantly different approach will seem unnatural.

   Let's give the functional approach a fair chance.

2 FUNCTIONS AND THEIR DEFINITIONS

2.1 Mathematical Concepts and Terminology

In mathematics, 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 mapped to elements of A by f 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 $b \in B$.

A function $f : A \to B$ is *one-to-one* (or *injective*) if and only if distinct elements of A are mapped to distinct elements of B. That is, $f(a) = f(a')$ if and only if $a = a'$.

A function $f : A \to B$ is *onto* (or *surjective*) if and only if, for every element $b \in B$, there is some element $a \in A$ such that $f(a) = b$.

A function $f : A \to B$ is a *one-to-one correspondence* (or *bijection*) if and only if f is one-to-one and onto.

Given functions $f : A \to B$ and $g : B \to C$, the *composition* of f and g, written $g \circ f$, is a function from A into C such that

$$(g \circ f)(a) = g(f(a)).$$

A function $f^{-1} : B \to A$ is an *inverse* of $f : A \to B$ if and only if, for every $a \in A$, $f^{-1}(f(a)) = a$.

An inverse exists for any one-to-one function.

If function $f : A \to B$ is a one-to-one correspondence, then there exists an inverse function $f^{-1} : B \to A$ such that, for every $a \in A$, $f^{-1}(f(a)) = a$ and that, for every $b \in B$, $f(f^{-1}(b)) = b$. Hence, functions that are one-to-one correspondences are also said to be *invertible*.

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.

A function $\oplus : (A \times A) \to A$ is called a *binary operation on A*. We usually write binary operations in infix form: $a \oplus a'$. (In computing science, we often call a function $\oplus : (A \times B) \to C$ a binary operation as well.)

Let $\oplus$ be a binary operation on some set A and x, y, and z be elements of A.

- Operation $\oplus$ is *associative* if and only if $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for any x, y, and z.
- Operation $\oplus$ is *commutative* (also called *symmetric*) if and only if $x \oplus y = y \oplus x$ for any x and y.
- An element e of set A is a *left identity* of $\oplus$ if and only if $e \oplus x = x$ for any x, a *right identity* if and only if $x \oplus e = x$, and an *identity* if and only if it is both a left and a right identity. An identity of an operation is sometimes 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, a *right zero* if and only if $x \oplus z = z$, and a *zero* 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 and y is a *right inverse* of x. Elements x and y are inverses of each other if $x \oplus y = e = y \oplus x$.
- If $\oplus$ is an *associative* operation, then $\oplus$ and A are said to form a *semigroup*.
- A semigroup that also has an *identity* element is called a *monoid*.
- If every element of a monoid has an inverse then the monoid is called a *group*.
- If a monoid or group is also commutative, then it is said to be *Abelian*.

2.2 Function Definitions

Note: Mathematicians usually refer to the positive integers as the *natural numbers*. Computing scientists usually include 0 in the set of natural numbers.

Consider the factorial function *fact*. This function can be defined in several ways. For any natural number, we might define *fact* with the equation

$$fact(n) = 1 \times 2 \times 3 \times \cdots \times n$$

or, more formally, using the product operator as

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

or, in the notation that the instructor prefers, as

$$fact(n) = (\Pi\, i : 1 \leq i \leq n : i).$$

We note that $fact(0) = 1$, the *identity* element of the multiplication operation.

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

$$fact'(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \times fact'(n-1), & \text{if } n \geq 1 \end{cases}$$

It is, of course, easy to see that the recurrence relation definition is equivalent to the previous definitions. 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.

2.3 Mathematical Induction over Natural Numbers

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

Base case $n = 0$**.** That $P(0)$ holds.

Inductive case $n = m+1$**.** 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, that is, for all natural numbers n,

$$fact(n) = fact'(n).$$

Base case $n = 0$.

$$\begin{array}{ll} & fact(0) \\ = & \quad \{ \text{ definition of } fact \text{ (left to right) } \} \\ & (\Pi\, i : 1 \leq i \leq 0 : i) \\ = & \quad \{ \text{ empty range for } \Pi,\ 1 \text{ is the identity element of } \times \ \} \\ & 1 \\ = & \quad \{ \text{ definition of } fact' \text{ (first leg, right to left) } \} \\ & fact'(0) \end{array}$$

Inductive case $n = m{+}1$.

Given $fact(m) = fact'(m)$, prove $fact(m{+}1) = fact'(m{+}1)$.

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

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

Note the equational style of reasoning we used. 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 of arithmetic, or the induction hypothesis.

Note that the structure of the 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 this course 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.

References

[1] J. Backus. Can programming languages be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall International, New York, 1988.

[3] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[4] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN NOTICES*, 27(5), May 1992.

[5] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[6] P. Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* MIT Press, Cambridge, Massachusetts, 1989.