LAZY FUNCTIONAL PROGRAMMING IN HASKELL

TUTORIAL PRESENTATION

H. Conrad Cunningham, Yi Liu, and Hui Xiong
Department of Computer and Information Science
University of Mississippi
University, MS 38677 USA
{*cunningham,liuyi,hxiong*}*@cs.olemiss.edu*

In ACM's 1977 Turing Award Lecture [1], Backus argues that conventional imperative programming languages are inherently disorderly:

> 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.

Why is the expression world orderly? The primary reason is 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. This property is called *referential transparency*.

An expression represents a (mathematical) function. The variables of the expression are the parameters of the function and the result from evaluating the expression is the value of the function. The purpose of a *functional programming* language, as we study in this tutorial, is to extend the advantages of expressions to the entire program, ridding ourselves of what Backus called the "disorderly" world of assignment statements.

A functional program consists entirely of functions. The main program is a function that takes the user's input as its arguments and delivers the program's output as its result. For nontrivial programs, this function is defined in terms of other functions, which themselves are defined in terms of yet other functions, and so forth until every function can be defined in terms of the language's primitive operations.

Another advantage of functional programming languages is that they support very powerful and regular abstraction mechanisms. These mechanisms result, in part, from the way that functions are handled. Unlike most conventional languages, functional

languages treat functions 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.

Functions that take functions as arguments or return functions as results, often called *higher-order functions*, provide quite flexible mechanisms for encapsulating patterns of computation. For example, we can define a higher-order function that encapsulates the computational pattern "apply operation OP to each element of a list and return the resulting list" by making the operation OP a parameter of the function. By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

Functions can also be *partially applied*. That is, we can "call" a multi-parameter function supplying arguments for just a subset of its parameters. The partial application of the function returns another function—a function that takes the remaining parameters and returns the expected result of the original function. Along with higher-order functions, this feature enables programs to construct new operators from existing ones by "freezing in" some of the arguments, to pass these operators to other parts of the program, and to apply them as needed.

The actual functional programming environment discussed in this tutorial is the Hugs 98 interpreter [7]. Hugs 98 accepts a language that is syntactically and semantically similar to the "lazily-evaluated" functional programming language Haskell 98 [4] [2, 3, 5, 8]. Hugs is a freely available interpreter that runs on a number of computing platforms.

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. In particular, an argument of a function (which may be an arbitrary expression) is not evaluated until the first time the corresponding parameter is referenced during the evaluation of the function. If the parameter is never referenced, then the corresponding argument is never evaluated. As a consequence of lazy evaluation, a data structure can be defined without having to worry about how it is processed and it can be processed without having to worry about how it is created. A data structure may even be conceptually "infinite" in length (as long as the program never actually tries to access all of it). Lazy evaluation allows programmers to separate the data from the control of processing, thus enabling programs to be highly modular [6].

This tutorial introduces those in attendance to the concepts of programming using the lazily evaluated, purely functional programming language Haskell and gives several examples that illustrate its power and elegance.

PRESENTERS

H. Conrad Cunningham is Chair and Associate Professor of Computer and Information Science at the University of Mississippi. His professional interests include concurrent and distributed computing, programming methodology, and software architecture. He has a BS degree in mathematics from Arkansas State University and MS and DSc degrees in computer science from Washington University in St. Louis. Cunningham created a beginning graduate course on Functional Programming in 1991 and has taught the course ten times. He has written an extensive set of lecture notes on functional programming with Hugs [3].

Yi Liu is a PhD student in Computer and Information Science at the University of Mississippi with interests in software engineering and artificial intelligence. She has a Master's degree in computer science from Nanjing University in China. She was a student in Cunningham's functional programming class in the spring 2003 semester.

Hui Xiong is an MS student in Computer and Information Science at the University of Mississippi with interests in software engineering. She has a Bachelor's degree in English from Wuhan University in China. She was a student in Cunningham's functional programming class in the spring 2003 semester.

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*, Vol. 21, No. 8, pp. 613-641, August 1978.

[2] R. Bird. *Introduction to Functional Programming Using Haskell,* Second edition, Prentice Hall Europe, 1998.

[3] H. C. Cunningham. *Notes on Functional Programming with Gofer,* Technical Report UMCIS-1995-01, University of Mississippi, Department of Computer and Information Science, Revised January 1997. http://www.cs.olemiss.edu/~hcc/reports/gofer_notes.pdf

[4] Haskell Organization. *Haskell: A Purely Functional Language,* http://www.haskell.org

[5] P. Hudak. *The Haskell School of Expression*, Cambridge University Press, 2000.

[6] J. Hughes. "Why Functional Programming Matters," *The Computer Journal*, Vol. 32, No. 2, pp. 98-107, 1989.

[7] Hugs Project. *Hugs Online*, http://www.haskell.org/hugs.

[8] S. Thompson. *Haskell: The Craft of Functional Programming*, Addison Wesley, 1999.