Haskell

*A Brief Introduction To A Non-strict Purely Functional Programming Language**

CSE 60531 Programming Languages Fall 2006

Peter Bui <pbui@cse.nd.edu>

Introduction

Named after Haskell Brooks Curry[1][2], Haskell is a non-strict pure typeful functional programming language. The non-strict part of this description denotes that Haskell performs lazy evaluation. This means that expressions are not evaluated until their value is needed, and then they are only computed enough to satisfy the particular need[9]. This is opposed to the strict evaluation technique used by more common languages such as C/C++ and Java. The "pure" aspect of the definition means that Haskell does not allow side effects, which is anything that affects the "state of the world" (for example input/output). To separate any impure computations from the rest of the program, Haskell employs a system of *monads* which will be discussed later. The term "typeful" manifests the fact that in Haskell, types are pervasive since the language employs a static type system that ensures that Haskell programs are *type safe*, meaning the programmer cannot mismatch types[3][5]. The last part of the description of the language, "functional", stems from the fact that evaluating a Haskell program is equivalent to evaluating a function in a mathematical sense. Again, this is unlike imperative languages where the evaluation is based upon a sequence of statements, one after another[3]. As can seen, Haskell is a unique language that breaks away from normal conventional programming languages.

The remainder of this paper summarizes the history of the Haskell programming language, provides an overview of its syntax, explains its execution model, examines some unique language features, and critically analyzes the success and impact of the programming language.

Brief History

Although Haskell was first introduced in 1987, it was not until the introduction of Haskell 98 that the language stabilized into a consistent and usable state. The original motivation and historical foundation of the programming language is best summarized by the introductory text in the Haskell 98 Report that forms the basis of modern Haskell[8]:

> In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, to discuss an unfortunate situation

*As will be examined in the paper, Haskell can be considered the de facto non-strict purely functional programming language.

[1] Haskell's work on combinatory logic[12] along with Alonzo Church's work on lambda calculus make up large portions of the mathematical theory behind functional programming. In fact, the authors credit Curry as the godfather of the programming language.

[2] The naming of language after a computer luminary is not unprecedented. For instance, Ada is named after Ada Lovelace, who is considered the first computer programmer.

[3] This is in contrast to languages such as Perl, Tcl, and Scheme which are dynamically typed. Languages such as C/C++ are weakly typed because the programmer can dynamically cast types.

> in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages.

According to the Haskell 98 committee, the primary goals of the language were that:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

As be seen, the language grew out of the functional programming community and its desire to standardize a powerful non-strict purely functional programming language that was suitable for both academic (research and teaching), and industrial (large systems) uses. Moreover, by making the the language freely available and by incorporating ideas deemed useful by the community at large, the committee encouraged the spread and adoption of the language. Since the introduction of the Haskell 98 standard, two main implementations have emerged: Hugs[6] and The Glasgow Haskell Compiler (GHC)[4]. The former is an interpreter that provides an almost complete implementation of the Haskell 98 standard, while the latter is "state-of-the art, open source" Haskell compiler that supports the entire 98 standard along with a variety of extensions.

General Syntax

Because Haskell is a large and complex language, this section only briefly discusses the syntax of Haskell functions as these are the core of most functional languages. It does this by providing a simple function definition and explaining the syntax involved in the program.

Figure 1: Factorial(n)

```
0 -- Factorial(n) Example
1 factorial :: Integer -> Integer
2 factorial  0 = 1
3 factorial  n | n > 0 = n * factorial (n-1)
4
5 -- Main
6 main :: IO ()
7 main = print (factorial 4)
```

Before getting into a syntax code example, it is necessary to examine the high-level construction of the language. Unlike conventional languages such as C/C++ where programmers code the proper sequence for actions the computer should execute, Haskell is a declarative language. This means that users type in

functions or equations to specify the behavior of the system. In general, then, Haskell programs can be seen as a collection of definitions and expressions.

The example in figure 1 is the Haskell implementation of the classic $factorial$ function[15]. The meaning of the code is as follows:

- **Line 0:** This is short comment in Haskell which continues until the end of the line. Long comments are constructed with "(-", and "-)" and can be nested, allowing programmers to block off code with internal comments for debugging.
- **Line 1:** This line is called a *type signature*. It denotes that for the function $factorial$, the input is of type $Integer$ and the output type is also $Integer$. This is used by the system (interpreter or compiler) to enforce strict type matching. Fortunately for lazy programmers, the signatures are not necessary if the system can figure them out automatically: the compiler or interpreter will only complain if it finds a mismatch[16]; explicit declaration, however, is recommended for clarity.
- **Line 2:** This is the first part of the function that provides the base case of the recursive definition of factorial. It reads as *if factorial has an input of 0, then the result is 1.*
- **Line 3:** This is the recursive step of the $factorial$ function. It reads as *if input n is greater than 0, then the output is n * factorial (n - 1).* Note that this a second definition for the $factorial$ function. The Haskell interpreter/compiler chooses which definition to run based on $pattern - matching$[4]. This means that when a function is called, the interpreter tries to match the expression with one of the function definitions. In this case, if the expression is "factorial 0", the first definition is matched and executed. Similarly, if the expression is "factorial 1", then the second definition is matched and evaluated.
- **Line 6:** This line denotes that the *main* function performs an IO operation. The *main* function, like in many other languages such as C/C++ is the first point of entry for execution.
- **Line 7:** This line says the main function will *print* the result of $factorial4$. Note that function arguments are not enclosed in parenthesis. Rather the token after the function is taken as the argument. In the case of *print*, we had to enclose to $factorial4$ in parentheses to prevent the system from interpreting it as two arguments to *print*.

Like other aspects of the language, the Haskell syntax diverges sharply from conventional languages. It should be noted, however, that the lexical structure is quite similar to the lambda calculus defined in [9] and as such can be described as a declarative lambda-calculus[4]. For a more complete coverage of the Haskell programming language's syntax, please reference the Haskell 98 report[8].

Execution Model

As with most functional programming languages, Haskell employs an "eval/apply" execution model to process the language[5]. This means that all computation is done by evaluating expressions to yield values[1]. At the backbone of this execution model is the lazy (non-strict) evaluation. As explained before, lazy evaluation means that nothing is ever computed until it is absolutely needed. This evaluation strategy allows for infinite lists (also known as streams), such as the *Fibonacci* example in Figure 2[15]. In this example, the calculation of the n-th Fibonacci number would only cause the execution of the first n items in the list (rather than the whole list) and return the result. A slightly easier to understand example of this concept of infinite lists is in Figure 3[2]. Here, we have a function, $startFrom$ that takes an initial number and counts up. The *sum* line takes the first ten elements of the infinite list start at 1 and sums them up. In the Haskell interpreter, the result is 55.

[4]Other functional programming languages such as ML also employ pattern-matching.

[5]In terms of actual implementations, the GHC employs the "eval/apply" execution model described in [11].

Figure 2: Infinite Fibonacci List

```
--  Infinite list (stream) of Fibonacci numbers
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

(-  Syntax Notes:
    ":" prepends an element to a list
    tail returns a list without its first element
    zipWith uses the specified function to combine corresponding elements of
    two lists to produce a third.
-)
```

Figure 3: Lazy Evaluation

```
> -- All Integers starting from N
> startFrom n = n : startFrom(n + 1)
> sum (take 10 (startFrom 1))
55
```

To achieve this non-strict evaluation, Haskell employs a *call-by-need* evaluation strategy throughout the system to execute expressions[15]. What this means that when a function is called, all the arguments are delayed or put off for future execution[6]. The result of this is that each argument is internally converted into a $future$ which is an packaged or encapsulated computation[9]. Because Haskell is a pure functional programming language, each of these $futures$ is referentially transparent, meaning that it does not matter when or where the $future$ is unpacked. When a value is finally needed, the $future$ is $forced$. If this is the first $force$, then the encapsulated argument is evaluated. Furthermore, no shared expression is evaluated more than once. Once the $future$ has been forced, then the result is shared to all dependent expressions. An example of this in figure 4.

Figure 4: Shared Evaluation

```
    (3 * 3) * (3 * 3) --  evaluate first (3 * 3)
==> 9 * (3 * 3) --  evaluate second (3 * 3)
==> 9 * 9
==> 81
```

In this example, the first $(3 * 3)$ is forced and evaluated. This is then saved. For the second $(3 * 3)$ no new calculation is done as the result of the previous evaluation is returned. Because of of the lazy evaluation strategy, only the minimum amount of calculation is used to determine the result of an expression.

[6]Other functional programming languages such as Scheme allow for this delayed evaluation by explicitly employing *delay* and $force$ functions to form *continuations*.

Unique Features

Referring back to the Haskell 98 report, an overview of Haskell's main features are as follows:

> Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers.

The most unique features out of this list is the non-strict semantics, static polymorphic typing, pattern-matching, and the monadic I/O system. Because lazy evaluation, static typing, and pattern-matching have been touched upon before, this section focuses on the use of monads in Haskell.

Figure 5: Monad

```
class Monad m where
    return  :: a -> m a     -- success
    fail    :: String -> m a     -- failure
    (>>=)   :: m a -> (a -> m b) -> m b     -- bind/augment
    (>>)    :: m a -> m b -> m b     -- then
```

As mentioned before, Haskell is a purely functional programming language meaning it disallows side-effects and the order of evaluation is immaterial. Unfortunately, there are times in programming where the order of evaluation does in fact matter, with I/O being one of these cases. In order to allow for sequencing of expressions and to implement I/O in a pure functional language, Haskell introduces the notion of a monad.

Basically, a monad is a container type (ie Class) with two main operations: *return* and *bind* (also known as $lift$ and *pipe*). The first operation, *return*, simply puts a single value in an instance of the container. The second function, *bind*, extracts the values from the container and filters them through a specified function. With monads, small computational pipelines can be constructed using the *bind* operation. The operations sequence the computations without modifying the values being passed between the functions. Therefore, a monad serves as a combinator for building a compound operation from two smaller operations[10]. It does this by taking in an operation and a function the receives the result of first operation, and constructing a second operation based on the result of the first, and finally running the second operation. In doing so, it maintains a consistent sequence of evaluation between two expressions. Figure 5 demonstrates how a monad can be defined in Haskell.

Figure 6: Do Block

```
main :: IO ()
main = do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn ("Nice to meet you, " ++ name ++ "!")

-- Syntax note: ++ concatenates strings
```

Haskell allows programmers to easily make use of monads. For instance, to explicitly denote a sequence of steps, Haskell provides users the *do* block construct. This construction is exemplified in figure 6. Also, the example also demonstrates the integration of monadic programming with input/operations. Anytime I/O is needed, the operation can be wrapped in a *do*-block or any other monadic construct that will sequence the operations.

For further information on monads, including the "Monad Laws", see [13], [14], [7], [10].

Critical Analysis

To analyze the impact and success of the programming language it is necessary to separately examine its role in academia and industry. In terms of the academic world, Haskell is thriving and spreading in influence. This can be seen in the numerous tutorials and textbooks written for the language, as well as college courses centered around it. Moreover, most research in lazy functional languages is being performed using Haskell. Some offshoots of this research includes a parallel version of Haskell dubbed "Parallel Haskell", and a distributed version named "Distributed Haskell"[15]. The creation of the Haskell 98 standard has provided the research foundation the committee sought to form by standardizing on a single non-strict pure functional programming language.

Regarding the mainstream programming world and the computer industry, Haskell has found less success. Because of its divergence from mainstream languages such as C/C++ or Java, and because of the numerous abstract features, the language appears to many programmers as too complex or difficult to understand. As such, the use of Haskell in industry has been limited. Such notable exceptions include the Darcs current version system (a distributed CVS replacement), and Pugs, which is an implementation of Perl6 in Haskell. Additionally, the Linspire Linux distribution has chosen Haskell as its main language development tool of choice. Despite these cases, however, Haskell remains relatively unused and unknown in industry and mainstream programming circles.

In light of the goals of the committee formulated in designing Haskell 98, it is clear that the programming language is a moderate success. As mentioned before, Haskell flourishes in the academic as specified in in the first goal, while it finds moderate success in industry with large applications such as Darcs and Pugs. Additionally, the free nature of the language as detailed in goal three has allowed other projects and researchers to pick up the language as the basis for further research and experimentation. Going back to the primary motivation, as of today, Haskell has solidified the non-strict pure functional programming language around the language. As such, Haskell is fulfilling its mission as being the de facto lazy purely functional programming language.

References

[1] Aaby, A, "Haskell Tutorial", http://moonbase.wwc.edu/ cs_dept/KU/PR/Haskell.html, 1998.

[2] Collberg, Christian, "Haskell – Lazy Evaluation", http://www.cs.arizona.edu/ collberg/Teaching/372/2005/Html/Html-14/index.html, September, 21, 2005.

[3] Daume III, Hal, "Yet Another Haskell Tutorial", University of Utah, Computer Science. 2002-2006.

[4] "The Glasgow Haskell Compiler", http://www.haskell.org/ghc/.

[5] Hudak, Paul, Peterson, John, Fasel, Joseph, "A Gentle Introduction to Haskell Version 98", http://www.haskell.org/tutorial/index.html, 2000.

[6] "Hugs Online", http://www.haskell.org/hugs.

[7] Jones, Simon Peyton, Walder, Philip, "Imperative functional programming", 20'th Symposium on Principles of Programming Languages, ACM Press, Charlotte, North Carolina, January 1993.

[8] Jones, Simon Peyton (editor), etc. "The Haskell 98 Language Report", http://haskell.org/onlinereport/, 1998.

[9] Kogge, Peter M. "The Architecture of Symbolic Computers", McGraw-Hill, Inc, 1991.

[10] Herman, Dave, "A Schemer's Introduction to Monads", http://www.ccs.neu.edu/home/dherman/research/tutorials/mon for-schemers.txt, July 13, 2004.

[11] Marlow, Simon, Jones, Simon, "How to make a fast curry: push/enter vs eval/apply", Proc International Conference on Functional Programming, Snowbird, Sept 2004, pp4-15.

[12] Martin, R. M. "Haskell Brooks Curry 1900 - 1982", Proceedings and Addresses of the American Philosophical Association, Vol. 56, No. 3 (Feb., 1983), pp. 404-405.

[13] Walder, Philip, "Comprehending monads", Mathematical Structures in Computer Science, Special issue of selected papers from 6'th Conference on Lisp and Functional Programming, 2:461-493, 1992.

[14] Walder, Philip, "The essence of functional programming", Invited talk, 19'th Symposium on Principles of Programming Languages, ACM Press, Albuquerque, January 1992.

[15] Wikipedia.org, "Haskell (programming language)", http://en.wikipedia.org/w/index.php?title=Haskell_%28programmin October 8, 2006.

[16] Wikibooks.org, "Haskell/Write Yourself a Scheme in 48 Hours", http://en.wikibooks.org/w/index.php?title=Haskell/Write_Yourself_a_Scheme_in_48_Hours/First_Steps&oldid=591885, September 2006.