Haskell

Kyle A. Kurzhal

University of North Alabama

Background

The computer science community has a long history of development with functional languages. The earliest of these developments date back to the 1950s, with the introduction of John McCarthy's introduction of Lisp (Hudak, Hughes, Jones, & Wadler, 2007). No longer than a decade later, the importance of lambda calculus in programming languages was identified by Peter Landin and Christopher Strachey (Hudak et al., 2007). These developments were further advanced “in the early ’70s, Rod Burstall and John Darlington were doing program transformation in a first-order functional language with function definition by pattern matching,” while “over the same period David Turner, a former student of Strachey, developed SASL, a pure higher-order functional language with lexically scoped variables...that incorporated Burstall and Darlington’s ideas on pattern matching into an executable programming language” (Hudak et al., 2007). Then, shortly after, a dialect of Lisp known as Scheme was developed by Gerry Sussman and Guy Steele with a more detailed focus on lambda calculus by use of lexical scoping, and ML was created as a meta-language with a polymorphic type system for the theorem prover LCF at Edinburgh by Robin Milner; both Scheme and ML contributed to the landscape of functional languages that would follow (Hudak et al., 2007).

Several key events began to occur that sparked the movement for non-strict or “lazy” evaluation. “Cons should not evaluate its arguments” by Dan Friedman and David Wise, and “A lazy evaluator” by Peter Henderson and James H. Morris Jr. were published in 1976. In the same year, David Turner changed SASL to a non-strict language, and later showed the usefulness of list processing by lazy evaluation. Some individuals were attempting to apply graph reduction principles to software, while research was also conducted on the use of dataflow and graph reduction machines to solve problems as a non-von Neumann architecture. Over the next several years, several conferences and meet-ups began to emerge such as the first Lisp conference in August 1980, the Advanced Course on Functional Programming and its Applications in July 1981, the first conference on Functional Programming Languages and Computer Architecture (FPCA) in September 1981, and the second Lisp conference in September 1982 that marked the beginning of the biannual Lisp and Functional Programming conference (LFP). Many languages like Miranda, Lazy ML, Orwell, and Clean were developed to meet this growing interest in non-strict evaluation, but many seemed to implement similar features and none had much backing as Miranda, developed by David Turner.

Haskell's genesis started with Simon Peyton Jones' visit to Paul Hudak at Yale on his trip to FPCA in the fall of 1987, upon which they “decided to initiate a meeting during FPCA, to garner interest in designing a new, common functional language” (p. 12-3), which was also encouraged by Philip Wadler. At the meeting it was decided that the best place to start developing the new language would be from an existing one. Since Miranda was deemed to be the best language for the task, the committee decided to first approach Turner about the adoption of his language. In response to the request, Turner stated that he wished to prevent multiple dialects of Miranda and declined the committee invitation (Hudak et al., 2007). From this point, the committee decided to set up an email mailing list fplangc@cs.ucl.ac.uk in order to communicate remotely, and eventually met up for their first planned meeting at Yale between January 9-12 in 1988. At this meeting, the group initially decided to establish its goals for the language as following:

* It should be suitable for teaching, research, and applications, including building large systems.
* It should be completely described via the publication of a formal syntax and semantics.
* It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
* It should be usable as a basis for further language research.
* It should be based on ideas that enjoy a wide consensus.
* It should reduce unnecessary diversity in functional programming languages. OL became the language chosen as a baseline.

Ultimately, the group abandoned OL and exclusive use of proven ideas, and never developed formal semantics. The other main items of discussion were concerned with the committee process and the name of the language. At first, the group decided upon “Curry,” named after Haskell B. Curry, whose contribution to Combinatory Logic was the basis upon which functional programming was built; however the group decided to avoid confusion and puns by instead using “Haskell” with permission from Mrs. Curry (Hudak et al., 2007).

The next most important meeting was at the University of Glasgow, April 6-9, 1988, where many unresolved issues were discussed and Hudak and Wadler were designated as the editors for the first Haskell report. This was followed by other meetings and emails until the Haskell version 1.0 Report was published in April 1990, soon after which the original mailing list was disbanded in place of a public mailing list. Other subsequent reports included:

* Haskell version 1.1 Report (August 1991) – Let expressions and operator sections allowed for the first time.
* Haskell version 1.2 Report (March 1992) – Minor changes and an appearance in *SIGPLAN Notices*.
* Haskell version 1.3 Report (May 1996) – A library report was added to enhance portability, Monadic I/O was added and I/O semantics were dropped, type classes were generalized to higher kinds using constructor classes, and algebraic data types were extended with new-types, strictness annotations, and named fields.
* Haskell version 1.4 Report (April 1997) – List comprehensions were generalized to arbitrary monads.
* Haskell 98 Report (February 1999) – The community agreed to support of a stable standard for the language and list comprehensions reverted to just lists.
* Revised Haskell 98 Report (December 2002) – Cambridge University Press published the Report as a book while still allowing the entire text to be freely available online.

Several other significant events occurred during this time as well, such as the founding of Haskell's website, haskell.org, in 1994 which is maintained by Hudak's group at Yale, and the Haskell committee turned over control of the language to the Haskell community while setting Jones over the Report as the sole editor between 1999-2002 (Hudak et al., 2007).

Regardless of Turner's decision to prevent dialects of Miranda, Haskell's development was highly influenced by it, which also makes Haskell a descendant of ML (Sebesta, 2004). Aspects of Miranda that influenced Haskell include the general methods of purity, higher, order, laziness, and static typing. Other similarities are found in terms of syntax for “the equational style of function definitions, especially pattern matching, guards, and where clauses; algebraic types; the notation for lists and list comprehensions; writing pair types as (num,bool) rather than the int\*bool of ML; capitalisation of data constructors; lexically distinguished user-defined infix operators; the use of a layout rule; and the naming of many standard functions” (Hudak et al., 2007).

General Facts

Haskell has non-strict or “lazy” evaluation, meaning that it uses a “call-by-need” methodology. This practice requires the overhead of delaying the evaluation until the moment it is needed, which prevents unnecessary evaluations from occurring, similar to “short-circuit evaluation” in C languages. To keep up with these unevaluated expressions, a record known as a “thunk” is created and managed. Lazy evaluation also requires overwriting the expression with the resulting value so that it is not evaluated twice. Sometimes this means that the memory required to hold all this information grows too large for the space allotted. To combat this problem, some strict features were added to the language (O'Sullivan, Goerzen, & Stewart, 2008).

As a result of Haskell's laziness, it follows that the heart of the language must also be pure. By saying that Haskell is “pure,” it is meant that the language does not allow any side effects to occur internally. This being said, there was initially a problem in dealing with I/O. The problem with I/O was later resolved with the introduction of Monads and monadic I/O, which restricted any side effects to only occur when dealing with external data. Therefore, this method separates pure and impure code, making code more modular and easier to debug. (O'Sullivan et al., 2008).

Graham Hutton (2007) defines types very simply as “a collection of related values” (p. 17), and in order to gain the most benefit of using Haskell, the user should have an in-depth understanding of its type system. As a functional language, Haskell evaluates programs as a set of expressions where each expression has its own type (Hutton, 2007). As a result, the type system allows the programmer to think about a task in a more abstract way when designing programs thanks to an emphasis on strong types, static types, and type inference. When referring to strong types, “strong” means that Haskell will not allow a program to contain type errors (O'Sullivan et al., 2008). That is, since each expression must have its own type and sometimes requires other types (functions), any instance of a mismatched type may be prevented from occurring, such as non-matching parameters or return values to assignment statements. This concept also implies that types cannot be implicitly coerced, however they may be converted by explicitly using coercion functions (O'Sullivan et al., 2008). The benefit to using a strong type system is that any bugs within the program itself that would normally be caused by mismatched types may be eliminated before the initial run ever occurs, but it does also mean that performance of a program is decreased (O'Sullivan et al., 2008). Furthermore, Haskell's type system is static, meaning “that the compiler knows the type of every value and expression at compile time, before any code is executed” (O'Sullivan et al., 2008, p. 19). This principle works hand-in-hand with Haskell's strong type checking to eliminate all possibilities of type errors for an expression given a certain value (O'Sullivan et al., 2008). Lastly, as a consequence of strong and static type checking, a Haskell compiler is also able to infer, most of the time, the type of an expression, which is completely dependent upon the rules that match particular types with certain values (Hutton, 2007). For instance, the integer 555 may be passed to an expression requiring a type of Int without explicitly declaring it, although possible to do so, as such. Even with these key details, a program written in Haskell is not necessarily guaranteed to be completely free from errors such as division by zero or other logic errors (Hutton, 2007).

Among all the compilers available, the Glasgow Haskell Compiler, otherwise known as the GHC, is one of the most popular due to its ability to produce highly efficient programs for major platforms, memory and CPU profiling, its large set of libraries, and an interactive interpreter based on the compiler (O'Sullivan et al., 2008). However, there are several other well known compilers such as the Hugs (“Haskell User's Gofer System”) compiler, a derivative of the Gofer (“Good For Equational Reasoning”) (Hudak et al., 2007). Each of the available compilers are designed to include some form of garbage collection, since Haskell generally abstracts such low-level concepts from the user. However, there are certain tricks that programmers are able to use in order to reduce the amount of memory usage in their programs (O'Sullivan et al., 2008).

In order to incorporate reusable groups of code, Haskell initially presented modules, each with an individual interface and implementation. By the time of publishing the Haskell version 1.4 Report, the interfaces were formally dropped from the language and replaced with the notion that interfaces should be able to be compiled as individual files. As the language grew, it became apparent that a hierarchy was needed to better categorize modules and avoid overlap. Malcolm Wallace headed a solution that used a method similar to Java for subgrouping modules under other parent modules. Following this, even though modules were created to promote code reuse, at first there was no means of distributing them among the community. For this reason, Isaac Jones began work on a system designed to package and distribute modules, known as Cabal, in 2004 which was expanded by David Himmelstrup's Cabal package server, Hackage. Furthermore, although the original release of Haskell included the Prelude library, the general standard for the language, the need arose to create a source of definitions separately from the main language. In order to meet this need and promote a cross-platform design; teams from Hugs, GHC, and nhc worked together on open-source libraries that were compatible with each compiler (Hudak et al., 2007).

Finally, before moving forward, a few final notes must be mentioned. First, Haskell's standard uses keywords. Secondly, files written in Haskell must have the extension “.hs,” and can be compiled by the GHC via the command line. In order to indicate a file as a module, the syntax *module ModuleName () where* is specified at the top of the file. In order to import other modules into a file, the syntax *import ModuleName* should be used at the top of the file, but below the module name for the file. Furthermore, each script must apply the layout rule, in which all related expression definitions must begin in the same column, although it is possible in some cases to use curly braces to group definitions. Next, there are two forms of comments, inline and block. Inline comments may be specified with two dashes (--) and comment out the rest of the line, while block comments are denoted as {- … -} in which comments may fit between the dashes and span multiple lines. Lastly, expressions throughout the Haskell language may, and are encouraged, to be grouped within parentheses in order to eliminate confusion for the programmer and the compiler (Hutton, 2007).

Data Types

When learning about Haskell's type system, it is important to be able to interpret Haskell's definition of types and values and what basic types are built into the language. Expressions and their types are represented in terms of the expression name followed by its type in the format *name* :: *type*, this is also known as the *type signature*, which may be assigned to variables with the assignment operator (=)(O'Sullivan et al., 2008). When observing this representation, one may consider that the expression “is of” the specified type (Thompson, 1999), where each type will always begin with a capitalized letter. Haskell's basic types include the Bool, containing the logical values True and False; Char, an individual Unicode character; Int, a signed integer value with a fixed width; Integer, a signed integer value with an unbounded size; Float, a single precision floating-point number which is normally not used; and Double, a double precision floating point number. The True and False values associated with the Bool type are actually types as well. Commonly used operators of Bool include several that are C-influenced, such as AND (&&), OR (||), and equivalence (==). Other operators include “not” for inversion, and not equal to (/=). For the Char type, relational operators for checking exact equality (== and /=), C-like comparative equality (<, >, <=, >=), and getting the minimum or maximum of two arguments (“min”, “max”) since . There are also several special characters that use the backslash as an escape: the tab ('\t'), newline ('\n'), backslash ('\\'), single quote ('\''), and double quote ('\”'). As for the arithmetic using types of Int, Integer, or Double; there are operators for addition (+), subtraction (–), multiplication (\*), negation of the sign (“negate”), absolute value (“abs”), retrieving the sign in terms of positive or negative one (“signum”), raise to an integer power (^), raise to a floating-point power (\*\*), and also allows the use of relational operators. In the case of the subtraction operator, it may also be placed before the value to represent the negative of that value, which often needs to be grouped within parentheses in order to avoid confusing the compiler. Moreover, integer and floating point types have slightly differing operators. For types of Int and Integer, there are the division (“div”) and modulus (“mod”) operators, while floating point types such as Double have the division (/), reciprocal (“recip”) operators. By convention, operators specified as words would normally have to be specified as in the prefix form, *operator value [value]*, but may also be used as an infix format if surrounded by backticks, *value `operator` value*. In addition, there are several other special operators for these types that are built into the standard Prelude library, for example, rounding (“ceiling”, “floor”, “round”), trigonometry (“cos”, “sin”, “tan”, “acos”, “asin”, “atan”), exponentials (“exp”), PI (“pi”), square root (“sqrt”), and logarithms (“log”, “logBase”) (Thompson, 1999).

Furthermore, Haskell also implements several composite types derived from its basic types. The two foremost examples of composite types are the list and tuple. The type of each list (“[]”) and tuple (“()”) is defined by the types that they contain. That is, a list is allowed to contain any number of elements of a single *type* and may continue to grow in size, where examples could include a list of Int types (“[Int]”) or a list of lists of Double types (“[Double]”). On the other hand, a tuple is allowed to contain any number of various types as originally defined and whose size cannot change, such as a tuple of type Int and Char (“(Int, Char)”) or a tuple of type Double and Double (“(Double, Double)”). Lists and tuples both have a special type that occurs when either is empty, [] for a list and () for a tuple. When a list or tuple is defined, it is considered to be a type. This means that a list could contain other lists or tuples of certain types, including more lists or tuples, and tuples can contain similar types. However, although a list may contain only a single element, there is not an existing notion of a single tuple. Also, since of order of lists and tuples matters, even if quantity and kinds of types in two tuples are the same, a different ordering of types within tuples results in different resulting types. Another common composite type is the String type, which is essentially just a list of Char (“[Char]”), and may be specified as a set of characters listed between two sets of double quotes. Lists also differ from tuples in terms of operators and enumerations. For example, it is possible to enumerate over a list of the type Int, Integer, Double, and Char. These enumerations may be performed by using “..” within the lists for integer and floating point types in the form [1..10], which will produce a list of integers from 1-10, or character types in the form ['a'..'z'], producing a list of characters containing characters 'a'-'z'. It is also possible to add a step by which values will be generated by denoting an enumeration like [1,3..10], resulting in a list counting by two's until the largest value that matches the step up to the last specified value is reached (the last number generated in this case is 9) (O'Sullivan et al., 2008). Another special feature of lists are list comprehensions, the composition of which is *[returnExpression | n ← xs [, testExpression ...]]*. List comprehensions may be used in order to take an existing list and transform it into a new list. In this case, the *returnExpression* is the expression that will be added to the new list, *n ← xs* is referred to as a “generator” that passes expressions from the old list to the *returnExpression*, and the *testExpression* optional conditions that determine if an expression is passed to the new list (Thompson, 1999). The main functions over a list include concatenation of two lists (++) and construction of a list (:), which appends an element to the front of a list and where the format is specified as *element : [list]*, grabbing the first element of a list (“head”), grabbing the list of elements after the first element (“tail”), taking a list of the first *n* elements of a list (“take”), taking all but the first *n* elements of a list (“drop”), getting the length of a list (“length”), and checking if a list is empty (“null”). More of the practical functions for lists may be found in the Data.List module. The common functions for tuples are much smaller in number, that is, retrieving the first element (“fst”) or the second element (“snd”) of a tuple (O'Sullivan et al., 2008).

(? Defining Data Types ?)

In Haskell, there are two main methods of defining new types. The first of these is to define a new type as a synonym of another type using the *type* keyword followed by the capitalized name of the type in the form *type Typename = ChosenType*. It is also possible to parameterize types, allowing a means of polymorphism and thereby also abstracting the type (Hutton, 2007). The second means to define a type is by use of the *data* keyword, where the format is specified as *data Typename = Type1 [arg1, […]] [ | Type2 [arg1, […]]] [deriving (SomeClass)]*. The left-hand side of the expression is known as the “type constructor,” which specifies the resulting type upon declaration, and each type, separated by a guard (|), on the right-hand side is referred to as a “value constructor,” which may be composed of types that may or may not take arguments. Upon further inspection, there is also the keyword *deriving*, allowing a type to extend functionality from other classes (O'Sullivan, 2008). By implementing types in this way, it is considered a use of what Haskell refers to as an algebraic data type. These kind of types are especially useful in terms of abstraction. For instance, although a tuple may be able to be used to generically accomplish the same task, it is easier to draw distinctions in cases where a type has two value constructors that are made up of the same types. On an occasion such as this, checking for equality of two tuples may return True, but equality of two variables of the same type that were initialized with differing value constructors may return False (O'Sullivan, 2008).

Types defined using the *data* keyword have the distinct advantage of the ability to define recursive types. What this means is that a type is able to initialize a value constructor with another value constructor of the same type. One of the most trivial examples of this would be a simple list, defined as *data List a = Cons a (List a) | Nil*. Essentially, this structure parameterizes a type and then allows a choice of two value constructors. If the Cons constructor is chosen, then it must be passed a value of any type as the first argument and a constructor or expression of the List type, otherwise the Nil constructor must be specified. When initializing an expression of this type, it may possibly look something like *z = Cons 'a' (Cons 'b' (Cons 'c' Nil))*. Ultimately, the same effect of templating is also achieved in a very narrow definition, using generic parameters to encourage polymorphic types. In cases such as this, it is important to remember that the type of the evaluated expression will be *List* *SomeType*,where *SomeType* is the type that was filled into the generically specified parameter (Hutton, 2007).

Subprograms

As a general standard, Haskell refers to subroutines as “functions.” Each of these functions, as a result of Haskell's type system, have a type specifically associated with it, and may be adequately described as *name :: ParameterType [→ ParameterTypeN [...]] →ReturnType*, in which “→” means that the type on the left returns the type to the right. When defining functions, the syntax for binding function names to the appropriate expression is in the format *name [arg1 [...]]= expression*, and, if one chooses to do so, it is also possible to specify the type before the actual definition in order to provide better documentation and clarify the intention of the function to the compiler. According to the previous type syntax description, it is possible to pass one or more arguments to a function, but doing so affects the number of parameters denoted in the type. This is because Haskell implicitly uses right associative “curried” functions, meaning that when more than one parameter is specified the overall function is broken up into a set of smaller functions that take a single parameter and return their result as a function to the calling function as its return. In other words, the type of the function *add x y = x + y* where *x* and *y* are of type Int is better described as *add :: Int → (Int → Int).* However, if a function type is explicitly defined before the function definition, then the order of associativity may be changed.Also, because functions are considered expressions, it is possible to pass functions as a parameter to other functions, known as “higher-order” functions (Hutton, 2007). Interestingly, this is easier to understand when considering that Haskell uses lazy evaluation to prevent expression evaluation until it is absolutely necessary. Furthermore, the role of abstraction by polymorphism has a significant impact upon how the actions of functions may be interpreted. For instance, when examining the type of a function, it is possible that the type may include an abstract type represented as something like *name :: [a] → [a]*. Here, the *a* indicates that the type is polymorphic, meaning that the function may take a list of any type and return a list of that same type. Finally, there is one special kind of anonymous function known as a lambda function that is preceeded by the history of lambda calculus. These functions begin with a backslash (\) followed by the list of arguments, followed by an arrow (→) that denotes the beginning of the expression. The lambda function, although only able to contain a single clause, is useful when it is necessary as a sort of “glue” (O'Sullivan et al., 2008).

The means by which parameters are passed further begs the question of what kind of scope each argument has. Because of the notion of purity, it may be determined that arguments passed to a function must be passed by value as the default, since Haskell is guaranteed to to allow side effects, as long as it is not exposed to the outside world, and expressions and their resulting values are interchangeable. That is, each expression, whether a variable or function, is local to the function in which it is defined. The two means of declaring local expressions, apart from passed parameters, is by the use of the *let...in* and *where* keywords. When using *let*, the expression is specified before the group, denoted with *in* between the end of the expression and the beginning of the group, of expressions. The result allows the expression to be local to the subgroup of expressions, that may also use *let...in*, and any subgroups within those expressions and so on and so forth. However, if the name of the expression is redefined in a subgroup, then that definition is local to the scope of its subgroups. On the other hand is the *where* keyword, which defines an expression after the main set of expressions of a function, allowing the function to be more easily read. In the case that global expressions are ever needed, they simply must be defined in the global space of the program file rather than being grouped within other expressions (O'Sullivan et al., 2008).

Functions in Haskell are also able to use conditional expressions and what are referred to as “guards,” similar to the guards in defining types, in order to control the flow of the program. Conditional expressions are the well known *if...then...else* keywords. The *if* is followed by the expression to be evaluated, followed by *then* and the expression that should be evaluated if the conditional expression evaluates to True, followed by *else* and the expression that should be evaluated if the conditional expression evaluates to False. To avoid any problems of compiler confusion that tend to arise in the case of nested *if* statements, also known as the “dangling else,” Haskell requires that all conditional *if* expressions must have a matching *else*. It is also key to remember that the type of expressions denoted by *then* and else must match, otherwise an error will be thrown. Apart from this common construct, there is the ability to use guards to define expressions for a function given the argument meets certain conditions. For a single function, a guard (|) is initially set between the function parameter and an expression that is evaluated to either True or False, followed by the assignment operator and the expression to be evaluated if the conditional expression is true. This initial guard expression may be followed by other guard expressions which must all have the same return type. Each of these expressions is evaluated until there is a True result or if the end is reached. To establish a default behavior in the case that all previous evaluations were False, the *otherwise* keyword may be used in place of the conditional expression along with an expression to be evaluated if it is reached. It also must be noted that the same effect could be achieved by specifying the True type instead of *otherwise* (Hutton, 2007).

Due to Haskell's nature as a functional language, implementing recursion is a fairly straightforward process; in fact, recursion is absolutely necessary for performing loops since there are no loop structures defined by Haskell. For a function to be recursive, it needs only to call itself and pass the required parameters, however, it is crucial, just as in any language that allows recursion, to set conditions to prevent infinite looping until a runtime error from occurring. Also, although there are no explicit loop controls, it is possible to pass accumulator values around to each recursive functional call to emulate the same behavior. Furthermore, even though the use of recursion is very powerful in Haskell, there are some downsides considering the principles of the language. By stating this, it is meant that because of lazy evaluation, purity, and the implications of these in terms of setting aside memory for the values contained within each function call as well as unevaluated thunks, a problem arises when the set of data managed within a program grows very large very quickly as a result of fast-acting function calls; this problem is referred to as a “space leak” in the Haskell community. In order to resolve this issue, the ability to explicitly use strict functions such as “seq,” which forces its first argument to be evaluated as long as it is not an algebraic type and then returns to its second argument. Use of the “seq” function can be extremely beneficial when dealing with a quickly growing space leak, but unnecessary use also incurs overhead such as pattern matching that may actually slow down execution speed or even introduce a new space leak (O'Sullivan et al., 2008).

In addition, functions may also be used to handle expressions based on pattern matching. One such example of pattern matching includes case expressions. By using case expressions, it is possible match the type of an expression and return another expression as a result. The syntax for this is *case expression of* followed by indented lines of *MatchingType → ReturnExpression* for types that should be matched. In the case that the expression passed matches one of the specified types, then the associated expression will be returned (O'Sullivan et al., 2008). Other types of pattern matching may also be performed on lists, tuples, and algebraic types. To pattern match on lists, an argument of a list type may be passed to a function, allowing the use of the construction operator to separate the head and tail of a list, such as *name (x:xs) = expression* where *x* is element from the head of the list and *xs* is the list that makes of the tail of the argument. Expressions of the correct type may also be used in place of *x* or *xs* to specify a match for an exact pattern. To perform this with tuples, simply defining a function in the form *name (arg1, arg2 [, …]) = expression* where the number of tuple arguments matches the tuple sent as a parameter in order to retrieve individual arguments. Lastly, to implement pattern matching with algebraic types, a function is defined as *name (ValueConstructor [argNameN …]) = expression* and individual arguments may be retrieved by the specified name. Finally, if the programmer knows that only certain items are needed within pattern matching, wild cards (\_) may be used in place of argument names. The benefit to using wild cards is the ability to make code clearer, preventing binding of unnecessary arguments, and it prevents bugs that may possibly arise from defining an argument but not using it in an expression (O'Sullivan et al., 2008).

Typeclasses

Typeclasses are one of the most defining features of the Haskell language. The reason for this is because a typeclass is essentially “a collection of types that support certain overloaded operations called *methods*” (Hutton, 2007, p. 24). Incorporation of these structures into the language has a direct impact on the writability of commonly used functions among several specified types by compacting the code and preventing the programmer from having to define functions over and over again for different numbers and types of arguments as well as differing return types. Several basic typeclasses provided by the Prelude library are: Eq, for checking the equality of types; Ord, which handles checking how expressions compare as well as their maximum and minimum possible values; Show is used to display expressions as a string; Read takes a string representing a certain type and value and converts it appropriately; Num, used in order to provide basic arithmetic operators; Integral provides division and modulus to integers; and Fractional, which provides operators for fractional calculations of floating-point values. These typeclasses are what compose the majority of operators for Haskell's basic types (Hutton, 2007).

In order to define a new typeclass, the syntax *class TypeclassName a where* must be used and followed by the type definitions of the functions at the very least. If the designer of the typeclass wished to specifically define the function, then the function definition may be provided after the function type. However, it is still necessary to declare a type as an instance of the typeclass, and this requires that at least one function be implemented. In order to instantiate the chosen type, the syntax is as follows: *class TypeClassName ChosenType where*, and is followed by at least one function definition. Although this is the main way to instantiate a typeclass type, it is also possible for most basic typeclasses to simply use the *deriving* keyword when defining types (O'Sullivan et al., 2008).

Monads

Alongside typeclasses is yet another defining feature of Haskell, the Monad. Monads are derived from a branch of category theory, and are a typeclass within Haskell. In order to determine if a type fits into the category of a monad, it must have a generic type constructor, a function to chain the result of one function into another, and the ability to inject a type (also referred to as “wrapping” the type in the monad) which is normally used for injecting pure code. Essentially, the idea of monads are simple enough, that is, “passing around implicit data or short-circuiting a chain of evaluations if one fails” (O'Sullivan et al., 2008, p. 328). The two main operators used in monads are binding the result of a function to the input of another (>>=) and chaining several expression according to execution order without passing their results as arguments to the next one (>>).

I/O

Although it is certainly possible to build programs that are internally secure, there are also many type of applications that require interaction with the outside world. For this reason, Haskell allows I/O in such a way that it becomes an isolated example of impurity, making it easier to find bugs later. With the introduction of I/O, the new IO type must be considered as well. Any types of IO may be generally considered to be an action, able to be stored, executed, and chained together with other IO types. There are some functions with a return type of IO (), denoting that the expression is I/O without any return. Usually, all I/O that must be executed are placed within the *do* block to sequence two or more actions to be performed. Within this block, the new arrow (←) operator may be used to store a return from some input into a local variable and the *let* keyword should be used to retrieve results from pure code. However, there are also two other seuqencing operators, sequencing actions (>>) and the ability pass the result of an action that has been run to another action returned by a function (>>=). Common functions include reading lines from the predefined stdin handle (“getLine”) and writing out a string to the predefined stdout and stderr handles(“putStrLn”). Interestingly, the main function is actually an I/O action, thereby allowing it to communicate with other I/O. To interface with files, the System.IO module may be taken advantage of, using the openFile function to return Handle to the file, after which operations may be performed until the handle is closed with hClose. Also, interacting with the file system is possible by use of the System.Directory module. I/O actions are performed in a lazy fashion, which allows processing smaller chunks of data only when needed, and therefore keeping memory usage low. Ultimately though, all I/O actions are defined by the IO monad. This means that in order to pass pure expressions to the monad, the *return* keyword may be specified before a group of pure expressions (O'Sullivan et al., 2008).

Exception Handling

Preparing code for any potentially critical errors is of the utmost importance, and may be handled in several different ways. One possibility is to use the Maybe type, defined as a type of Maybe *SomeType* or Maybe Nothing, to set up checks as functions to define the matching result such as Maybe Nothing for certain arguments passed to the function that would cause errors. Similar options would include the Maybe monad, or the Either type. The Either type is able to return Left in case of an error and Right for success. Furthermore, the Control.Exception module holds functions and types relating to defined exceptions. Some common functions include “try,” which wraps an IO action and returns Left or Right, and “throw,” which can throw any type of Exception. There also is the “throwIO” function for throwing errors in the IO monad, and the “ioError” function for “generating an arbitrary I/O-related exception (O'Sullivan et al., 2008, p. 459). Since the majority of errors that could occur are more likely to result from I/O, the programmer must be especially careful to handle anomalies (O'Sullivan et al., 2008).

Concurrency

If increased performance or several simultaneously running tasks are a necessary factor, it could be worth considering the modules and functions that Haskell offers for handling concurrency and parallelism. However, the language makes a distinction between concurrency, which is considered more for tasks networking protocols, and parallelism, thought to have a greater presence in I/O data processing by taking a set of data and splitting it across multiple cores. In order to use threads within Haskell, one may import the Control.Concurrent module and use the forkIO function. Like in other languages, sometimes it is necessary to communicate between two threads, which may be handled through the use of the MVar variable. MVar is able to put a thread to sleep if it attempts to load data into a full MVar or if it attempts to read from an empty MVar. In addition, it is also possible to use the Chan type for one-way communications. Both MVar and Chan are non-strict, and therefore do not evaluate their expressions, but Chan also has the possible disadvantage of being unbounded, meaning that messages can build up if more writing occurs than reading. When working with threads in Haskell, it is also critical to know that when working within the GCH runtime system, when the main thread is completed the other threads are ended by default, but this behavior may be overridden. Furthermore, the default behavior of threads is to run under a single OS thread, and this option must be explicitly changed if multiple threads must be spread among multiple OS threads. Users are also able to get the current state of a thread. Even considering the flexibility of concurrency within Haskell, there is still possible to run into deadlock or starvation, which must be carefully handled by the programmer (O'Sullivan et al., 2008).

Sample Program

To get an idea of Haskell's ability to write concise code to accomplish a certain task, a program was included in Appendix B comparing the a quicksort function implemented in C to Haskell's version of the code. In the Haskell program, the quicksort function is first defined to return an empty list when given an empty list as an argument. Secondly, it is defined to return an expression of the type of list initially passed to it. This list is constructed by recursively passing the list comprehension of values less than the head of the list argument passed to that particular function call, and doing the same but only considering values greater than or equal to the head of the list argument. Those lists of values less than the current head are placed to the left, the head element is placed in the center, and the lists of values greater than or equal to the current head element are placed on the right, and then the lists are concatenated together starting from the outside due to the recursive calls and Haskell's lazy evaluation. There is also a mixedList function that concatenates lists generated using enumerations. Ultimately, a main function's expression is evaluated, starting by sorting the unordered list of integers, then converting the list to a string, and finally printing the string to the console.

Impact and Future

As Haskell is approaching the 25th year since its initial publishing, it is important to stop and take into account its progress in order to correctly assess any promise for the future. In the beginning, Haskell began as an academically focused language with a relatively small following, but has grown from under 100 consistent users in 1995 to about 600 users in 2005. Those who have been impacted by Haskell include both academic and industrial groups. According to a web survey taken by Hudak and his associates (2007) for the academic year 2005-2006, 126 teachers from 89 universities among 22 countries; the highest responses being from the “USA (22%), the UK (19%), Germany (11%), Sweden (8%), Australia (7%), and Portugal (5%)”; were estimated to teach Haskell to a range of 5000-10,000 students (p. 12-41). The largest group of students taking these courses in Haskell was estimated to be 2000-4000 annually for undergraduates learning Haskell as their first or second language. The types of courses offering Haskell included a focus on functional or declarative programming, advanced programming, programming languages, theoretical topics, hardware descriptions, domain-specific languages, music, quantum computing, and distributed and parallel programming (Hudak et al., 2007).

As for industrial uses, companies from around the world have decided to implement Haskell in order to enhance their products. Some of these companies and their respective countries include ABM AMRO, an international bank in Amsterdam, The Netherlands; Aetion Technnologies LLC, a defense contractor that conducted business between 1999-2011 in Columbus, Ohio with a focus on projects involving Artificial Intelligence; Better (or Erudify), founded in 2012 and based in both Zurich, Switzerland and New York, USA, used Haskell for its back-end web-servers and learning logic in order to provide high-quality courses; and bCODE Pty Ltd in Sydney, Australia; Fractis Research, which develops mobile-friendly solutions in Freiburg, and Germany; and Functor AB in Stockholm, Sweden, a company that produces tools to help eliminate bugs (Rheno, 2014). In addition, a man by the name of Curt Sampson attempted to outline his experiences with the transition to using Haskell as a functional language upon founding Starling Software in Tokyo, Japan. After reviewing the benefits of functional languages, Sampson and his team convinced their client to allow the use of Haskell. With the team's previous experience in the imperative languages of C, Java, Ruby, C# and Python, Sampson concluded that using Haskell had the advantages of concise and readable code, no noticeable disadvantages in speed thanks to concurrency and parallelism, portability, and the ability to interface with foreign functions in C. However, Sampson also mentions that problems did arise when dealing with refactoring small bits of code, a lack of profiling tools, and space/memory leaks that arose from use of threads, but then later states that some of these issues could likely be resolved with more experience.

So, is Haskell projected to continue living? According to these developments in the academic and industrial communities, it seems to likely be so. This assessment may be further confirmed when evaluating the the active state of its community both in the IRC channel and the Haskell Wiki, which is the official Haskell website. For instance, the Haskell 2010 Report was just recently released, setting revised standards for the language by those who contributed to the mailing list and discussion boards (Marlow, 2009). Considering all these factors, it may even be possible that an increase in developments may be seen in the near future.

References

1. Sebesta, R. W. (2004). *Concepts of Programming Languages* (10th ed.). Upper Saddle River, New Jersey: Pearson Education, Inc.

2. Marlow, S. (2009, Nov. 24).Announcing Haskell 2010. Message posted to https://www.haskell.org/pipermail/haskell/2009-November/021750.html.

3. Hudak, P., & Hughes, J., & Jones, S. P., & Wadler P. (2007). A History of Haskell: Being Lazy With Class. *HOPL III Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 12-1-12-55. doi: 10.1145/1238844.1238856.

4. Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys (CSUR)*, *21(3)*, 359-411. doi: 10.1145/72551.72554.

5. Thompson, S. (1999). *The Craft of Functional Programming* (2nd ed.). Dorchester, Dorset: Henry Ling Ltd.

6. O'Sullivan, B., & Goerzen, J., & Stewart, D. (2008). *Real World Haskell*. Sebastopol, CA: O'Reilly Media, Inc.

7. Hutton, G. (2007). *Programming In Haskell*. New York, NY: Cambridge University Press.

8. Rheno (2014, Oct. 23). Haskell in industry. Retrieved November 30, 2014 from the Haskell Wiki: https://www.haskell.org/haskellwiki/Haskell\_in\_industry.

9. Sampson, C. J. (2009). Experience Report: Haskell in the “Real World”: Writing a Commercial Application in a Lazy Functional Language*. ACM SIGPLAN Notices – ICFP '09*, *44(9)*, 185-190. doi: 10.1145/1596550.1596578.

10. https://www.haskell.org/haskellwiki/Introduction