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).

General Facts

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, Goerzen, & Stewart, 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).

Types and Scopes

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* (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; and Double, a double precision floating point number.

References

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

(Sebesta, 2004)

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

(Marlow, 2009)

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.

(Hudak, Hughes, Jones, & Wadler, 2007)

(Hudak et al., 2007)

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.

(Hudak, 1989)

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

(Thompson, 1999)

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

(O'Sullivan, Goerzen, & Stewart, 2008)

(O'Sullivan et al., 2008)

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

(Hutton, 2007)