A History of Haskell

Pablo Buiras
pbuiras@seas.harvard.edu

November 2, 2016

1 A History of Haskell: Being Lazy With Class

```
@inproceedings{Hudak2007,
 author = {Hudak, Paul and Hughes, John and
    Peyton Jones, Simon and Wadler, Philip},
 title = {A History of Haskell: Being Lazy with Class},
 booktitle = {Proceedings of the Third ACM SIGPLAN
    Conference on History of Programming Languages},
 series = {HOPL III},
 year = {2007},
 isbn = {978-1-59593-766-7},
 location = {San Diego, California},
 pages = {12-1--12-55},
 url = {http://doi.acm.org/10.1145/1238844.1238856},
 doi = {10.1145/1238844.1238856},
 acmid = {1238856},
 publisher = {ACM},
 address = {New York, NY, USA},
}
```

Summary. This paper presents a thorough account of the history of Haskell from its beginnings in 1987 to 2007, the year when the paper was published. The paper also discusses the language's technical contributions, implementations and tools, and applications and impact.

Haskell came about as a common language to unify notation and streamline collaboration among pure FP researchers in the late 80s. At the time of its inception, there were multiple different languages with similar syntax and features, following a call to arms by John Backus in the late 70s to "liberate programming from the von Neumann style" [1] and drawing from the ideas of *pure functional programming* [4, 8, 10] (recursive functions, type systems, algebraic data types, pattern matching, referential transparency) and *lazy programming* [3, 5, 11] (streams, coroutines, call-by-name) that were developed around that time. Haskell embraced purity and pursued it relentlessly, leading to the development of *monadic I/O* [12, 7], which is one of the first examples of controlled effects. The trend of restricting side-effects in order to write safer code continues to this day in mainstream languages (e.g. regions, ownership types, DSLs for restricted effects).

Throughout its history, Haskell has been a useful testbed for type-system extensions and also a laboratory where new PL ideas could be tried out. The most distinctive feature of Haskell's type system is *type classes* [13], a mechanism for principled overloading and more generally a means for type-driven generation of executable evidence. There has been a useful exchange of ideas between Haskell and mainstream imperative languages such as C# that has led to new programming language features, e.g. LINQ, STM.

Evaluation. This paper is a good summary of Haskell's historical development and its impact on programming languages as a whole. Unfortunately, there have been new developments since 2007 that are not captured in this paper, although some of them were accurately predicted here, such as the inclusion of a strict mode in GHC. Recent developments in the Haskell type system aim to improve its expressive power and precision, by allowing type-level computation [9, 14, 2] with a long-term view to including full dependent types.

Further reading. Apart from the foundational papers referred to in the summary, a paper by John Hughes under the title "Why functional programming matters" [6] makes a compelling case for lazy evaluation (among other features), showing that it can be used as powerful glue that encourages modularity.

References

[1] J. Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.

[2] R. A. Eisenberg, S. Weirich, and H. G. Ahmed. Visible type application. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 229–254, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[3] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. *Automata, Languages and Programming*, pages 257–284, 1976.

[4] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A Metalanguage for Interactive Proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 119–130, New York, NY, USA, 1978. ACM.

[5] P. Henderson and J. H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 95–103, New York, NY, USA, 1976. ACM.

[6] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, Apr. 1989.

[7] S. P. Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2001.

[8] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[9] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM.

[10] D. Turner. *SASL Language Manual*. Document (Functional Language Implementation Project). University of Kent, Canterbury, UK, 1976.

[11] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[12] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.

[13] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[14] S. Weirich, J. Hsu, and R. A. Eisenberg. System fc with explicit kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 275–286, New York, NY, USA, 2013. ACM.