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


