Haskelite: A Step-By-Step Interpreter for Teaching Functional Programming

Pedro Vasconcelos

Departamento de Ciência de Computadores, Faculdade de Ciências da Universidade do Porto, Portugal

LIACC – Laboratório de Inteligência Artificial e Ciência de Computadores, Porto, Portugal

Abstract

This paper describes *Haskelite*, a step-by-step interpreter for a small subset of Haskell. Haskelite is designed to help teach fundamental concepts of functional programming, namely: evaluation by rewriting; definition of functions using pattern-matching; recursion; higher-order functions; and on-demand evaluation. The interpreter is implemented in Elm and compiled to JavaScript, so it runs on the browser and requires no installation.

This is on-going work and has yet to be fully evaluated; we present some initial experience in the classroom and highlight some points for improvement.

2012 ACM Subject Classification Software and its engineering → Functional languages; Information systems → Web applications; Software and its engineering → Interpreters

Keywords and phrases Functional programming, Step-by-step evaluators, Web applications

Digital Object Identifier 10.4230/OASIcs.ICPEC.2023.12

Funding This work was financially supported by: Base Funding - UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory - LIACC - funded by national funds through the FCT/MCTES (PIDDAC).

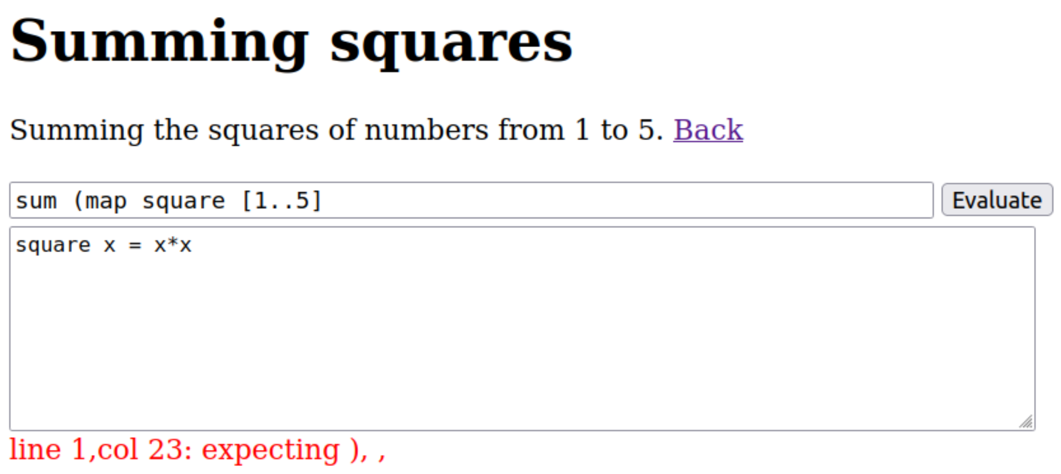
Acknowledgements The author would like to thank Sandra Alves for helpful feedback on a earlier draft of this paper.

1 Introduction

Functional programming has been taught in many universities as part of the Computer Science undergraduate curricula since the 1980s as attested by the steady tradition of textbooks available [2, 4, 9, 20, 1, 15]. Interest in functional programming is also increasing by the adoption of functional features in more mainstream languages, e.g. Java, Kotlin, Swift and Rust. Indeed, the ACM CS Curricula recommendation of August 2022 has increased the functional programming component from 3 to 4 hours relative to the CS2013 edition. However, educators have reported several beginner difficulties while learning functional programming, namely with the evaluation model [19], understanding type error messages [16, 21] or due to an incorrect perception of functional languages being only useful for academic problems [3, 14].

This paper describes *Haskelite*, a step-by-step interpreter for a small subset of Haskell [17]. Haskelite is designed to help with the first of the above problems, namely: understanding evaluation by rewriting; definition functions using pattern-matching and recursion; higher-order functions and on-demand evaluation. We focus on a restricted subset of Haskell covering these fundamental concepts. The Haskelite interpreter is implemented in Elm [6, 5] and compiled to JavaScript [10], so it runs on the browser and requires no installation. The source code is available at `https://github.com/pbv/haskelite` and a demo is available at `https://pbv.github.io/haskelite/`.

Haskelite is not intended to replace a full language implementation (so that students are not discouraged by learning only a "toy" language). It can nonetheless be useful for teachers and students to explain concepts and clarify misunderstandings.



Summing squares

Summing the squares of numbers from 1 to 5. Back

```
sum (map square [1..5]
```

```
square x = x*x
```

```
line 1,col 23: expecting ), ,
```

Figure 1 Haskelite interpreter embedded on a web page.

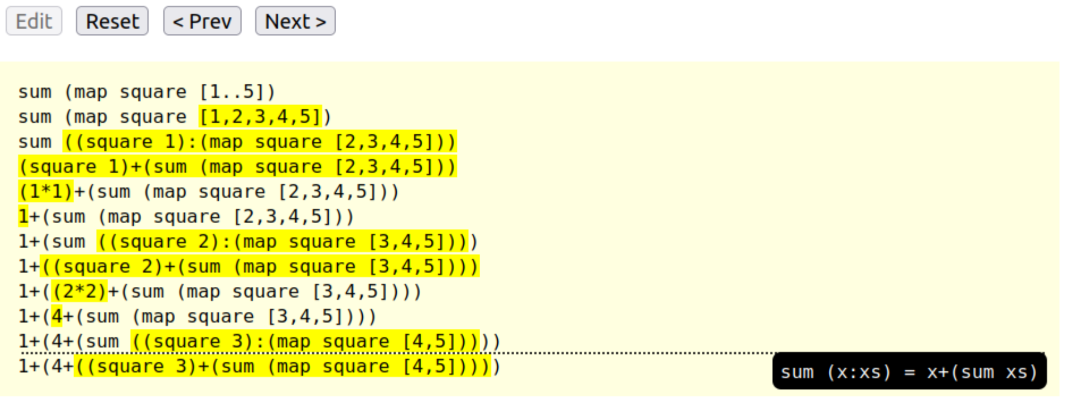
2 Design

2.1 Goals and motivation

The functional paradigm that we want to teach is well presented in the Bird and Wadler's classic textbook [2]: "Programming in a functional language consists of building definitions and using the computer to evaluate expressions. The primary role of the programmer is to construct a function to solve a given problem. (...) *The primary role of the computer is to act as an evaluator or calculator.*" (emphasis added). In principle, this notion of computation by evaluation should be familiar to university students from high-school algebra; in practice, we have noted that students repeatedly experience difficulties:

1. There is a jump in complexity going from numerical expressions (as is done in high-school) to expressions with algebraic data types such as pairs and lists;
2. When defining functions by pattern matching, it is common for students to either miss necessary cases (e.g. the empty list) or to add redundant ones (e.g. special cases for lists of one or two elements);
3. Understanding recursive definitions requires transitioning from thinking by extension (e.g. viewing a list as a sequence of values `[1,2,3,4]`) to thinking inductively (e.g. viewing a list as the nested application of constructors `1:(2:(3:(4:[]))))`;
4. Reasoning about termination also requires thinking inductively about "what is getting smaller" at each step of the recursion;
5. Reasoning about non-terminating processes (like lazy lists) also requires thinking (co-)-inductively about transformations that consume or produce a few elements at a time.

Such difficulties can be overcome by diligently working out calculations on paper, but this often requires individual motivation and sometimes assistance by the teacher. The design goal for Haskelite is to help expedite this learning process by automating the calculation steps. Moreover, it should also allow the student to quickly change the expression and/or definitions and re-try the evaluation, hopefully improving his or her's understanding.



```
sum (map square [1..5])
sum (map square [1,2,3,4,5])
sum ((square 1):(map square [2,3,4,5]))
(square 1)+(sum (map square [2,3,4,5]))
(1*1)+(sum (map square [2,3,4,5]))
1+(sum (map square [2,3,4,5]))
1+(sum ((square 2):(map square [3,4,5])))
1+((square 2)+(sum (map square [3,4,5])))
1+((2*2)+(sum (map square [3,4,5])))
1+(4+(sum (map square [3,4,5])))
1+(4+(sum ((square 3):(map square [4,5]))))
1+(4+((square 3)+(sum (map square [4,5]))))
```

```
sum (x:xs) = x+(sum xs)
```

Figure 2 Partial evaluation of the sum of squares.

2.2 User interface

Haskelite is a JavaScript application embedded in a web page. The user interface starts in *editing mode*, allowing students to type in equations defining functions and an expression. The editor provides feedback for parsing and type errors (see Figure 1). It is also possible for the instructor to prepare pages with filled-in expressions and definitions, so that the student is given particular examples to try.

Clicking the *evaluate* button switches to *evaluation mode*: the current expression is shown and the student can advance one step at a time; each evaluation step is either a primitive operation (e.g. arithmetic) or the application of a student function definition (or one from the *Prelude* of built-in functions). Previous evaluation steps are shown and a tool-tip highlights the *justification* for each step (see Figure 2). The student can also move backwards to previous steps.

Evaluation steps are justified as either primitive operations, one of the program's equations or an equation for a built-in *Prelude* function (such as `sum`). Using equations to justify evaluation steps serves two purposes: firstly, it connects the computation to the student's code; secondly, it helps prepare for the final topic of the course, which introduces proofs of program properties using equational reasoning and induction. This should help transition from *evaluations* with concrete values to *proofs* with variables that stand for arbitrary values.

2.3 Examples

We now present some examples of evaluations using Haskelite, highlighting possible uses in functional programming classes.

2.3.1 Structural recursion over lists

One of the early examples of recursion over lists to to (re)define the `sum` function that adds all values in a list:

```
sum [] = 0                  -- base case
sum (x:xs) = x + sum xs     -- recursive case
```

Haskelite can be used to exemplify evaluation of this function with some arbitrary lists. After this we can ask the students to define an analogous function to compute the product of all values a list. A frequent mistake is to define the base case incorrectly:

```
product [] = 0                  -- base case (WRONG)
product (x:xs) = x * product xs -- recursive case
```

Trying this in the Haskell interpreter gives only the final result (0); using Haskelite students can observe where the computation goes wrong:

```
product [1,2,3]
1*(product [2,3])
1*(2*(product [3]))
1*(2*(3*(product [])))
1*(2*(3*0))
1*(2*0)
1*0
0
```

The student's reaction might be to implement a less general solution where the base case is a list of a single value:

```
product [x] = x
product (x:xs) = x * product xs
```

This will, however, not work for the empty list; it is then possible for the teacher to compare with the sum function and relate the value of base case with the neutral element of operation in order to introduce the more general solution:

```
product [] = 1
product (x:xs) = x * product xs
```

2.3.2 Intercalating values

As a second example, consider the exercise: *Define a recursive function `intersperse` that intercalates a value between successive elements in a list.* For example: `intersperse 0 [1,2,3]` should be `[1,0,2,0,3]`.

This function requires a more complex recursion pattern because we need to distinguish whether the list has at least two values. The following attempt has a mistake in the recursive case:

```
intersperse a [] = []
intersperse a [x] = [x]
intersperse a (x:y:xs) = x:a:y:intersperse a xs
     -- WRONG: should be x:a:intersperse a (y:xs)
```

Again evaluating the test case yields the following trace:

```
intersperse 0 [1,2,3]
1:(0:(2:(intersperse 0 [3])))
1:(0:(2:[3]))
1:(0:[2,3])
1:[0,2,3]
[1,0,2,3]
```

From this trace it should be easier to understand why the definition is wrong: each recursion step should remove only a single element from the list.

2.3.3 Mapping over an infinite list

As a final example, consider mapping the function that multiplies by 3 over the infinite list `[1,2,3,...]`. This illustrates both higher-order functions (`map`) and on-demand evaluation of infinite lists:

```
map (\x -> (3*x)) [1..]
map (\x -> (3*x)) (1:[2..])
((\x -> (3*x)) 1):(map (\x -> (3*x)) [2..])
(3*1):(map (\x -> (3*x)) [2..])
3:(map (\x -> (3*x)) [2..])
3:(map (\x -> (3*x)) (2:[3..]))
3:(((\x -> (3*x)) 2):(map (\x -> (3*x)) [3..]))
3:((3*2):(map (\x -> (3*x)) [3..]))
3:(6:(map (\x -> (3*x)) [3..]))
3:(6:(map (\x -> (3*x)) (3:[4..])))
3:(6:(((\x -> (3*x)) 3):(map (\x -> (3*x)) [4..])))
3:(6:((3*3):(map (\x -> (3*x)) [4..])))
3:(6:(9:(map (\x -> (3*x)) [4..])))
```

Evaluation will not terminate; instead the resulting list is also infinite but new elements are produced one at a time, so that the process is still algorithmic.

3 Implementation

3.1 Technical details

Haskelite was developed in the Elm programming language [6, 5], which compiles to JavaScript and runs in a web browser. No server-side software is required: the parsing, type-checking and evaluation all run in the web browser. Listing 1 illustrates the embedding of a Haskelite interpreter in an HTML page to evaluate the sum of squares example.

The techniques used are well-known: parsing is done using a parser combinator library[1], type-checking implements the standard Hindley-Milner system [7] and evaluation is done by a naive rewriting engine using substitutions. The later choice is adequate: the focus is on the ability to easily relate the evaluation to the student's code (i.e. which equation is chosen) and efficiency is not critical for small programs and data sizes that a student is likely to use while learning.

The performance is perfectly adequate: the minified JavaScript bundle is only about 70KB of size, which is quite small by today's web page standards and parsing, type checking and evaluation of small programs appear instantaneous. Moreover, scalability to a large number of students is ensured because evaluation occurs in the clients' browsers.

Listing 1 Embedding a Haskelite interpreter in an HTML page.

```
<!doctype html>
<html>
  <head><link rel="stylesheet" href="screen.css"></head>
  <body>
    <div id="haskelite"></div>
```

[1] `https://package.elm-lang.org/packages/elm/parser/latest/`

```
    <script src="haskelite-min.js"></script>
    <script>
      Elm.Haskelite.init({
          node: document.getElementById("haskelite"),
          flags: { expression:"sum (map square [1..5])",
                    declarations: "square x = x*x" }
      });
    </script>
    <hr>
  </body>
</html>
```

3.2 Limitations

Haskell is large language with a rich syntax and many extensions. Currently Haskelite supports only a very limited subset of Haskell:

- the only basic values are integers and booleans;
- the only structured data types are tuples and lists;
- definitions are by equations only (no case-expressions) and do not support *guards* (i.e. inline conditions);
- no support for user defined data types;
- no support for list comprehensions;
- no support for type classes.

Another important technical detail is that Haskelite implements a *call-by-name* evaluation strategy rather *call-by-need* (i.e. lazy evaluation), meaning that sub-expressions whose results would be shared in a lazy implementation will be re-computed in Haskelite. This is done to present each evaluation step as a simple expression; the alternative would be to expose an implementation using graph reduction, which would obfuscate the presentation. Due to the purely functional semantics, this change in evaluation strategy does not change the final result, only the number of evaluations steps required.

4 Related work

Helium is a special compiler and interpreter developed at the Utrecht University for teaching a subset of Haskell [13]. Its principal focus is on producing better messages for beginners, particularly type errors. This system appears to no longer be maintained[2].

The Glasgow Haskell Compiler includes an interactive mode (GHCi) that can run programs using an imperative-style debugger, allowing setting break points, inspecting the values of variables and single-stepping execution [8]. It exposes Haskell's lazy evaluation mechanism, e.g. showing function arguments as partially evaluated expressions; thus it is intended more for experienced programmers than beginners.

Python tutor is a web site that allow visualizing the execution of Python, JavaScript, C, C++ and Java programs [12, 11]. The computational model is strictly imperative: the program state is visualized as pointer to the current instruction and the current values of variables in scope.

[2] As of April 2023, the *Hackage* database reports the last successful build in 2015: `https://hackage.haskell.org/package/helium`

There is a long tradition of teaching languages based on the functional programming language Scheme; *DrScheme* (now *DrRacket*) is an IDE for programming used for teaching which include a graphical debugger. This allows setting breakpoints, inspecting variables and step-by-step execution. There is even a web-based version [23]. However, Scheme does not encourage the reasoning by pattern-matching and equations that we are interested in teaching [22].

Haskelite was inspired by the *Lambda lessons* page built by Jan Paul Posma and Steve Krouse [18]. Lambda lessons allows step-by-step evaluation of simple definitions over lists and integers. The desire to extend this work lead to rewriting the interpreter from JavaScript into Elm.

5 Experience and further work

We started using Haskelite in the spring semester of 2022 and in only 2023 was the type-checker added. The interpreter has been used both in lectures and practical classes as soon as pattern matching and recursion are introduced.

The experience in practical classes is that students are keen to try out examples to clarify their understanding. Sometimes they will even try to use Haskelite as a debugger for larger programs (which will not work due to the limited language supported). We have not yet conducted any empirical validation of its application.

Nonetheless, there are already a number of directions for improvement that we have identified:

- adding more basic types to the language (e.g. strings and characters) would increase the range of examples that can be tried;
- similarly, adding support for user-defined datatypes would allow using it to explore more complex patterns of recursion, e.g. over tree-like data structures;
- the current user-interface side is not optimized for responsiveness in mobile devices; improving this should be relatively simple;
- there is currently no persistent state: expressions and definitions entered are lost each time the user closes the web page; adding the possibility of saving and loading programs using browser storage, or exporting links to share with others could be interesting in a classroom setting.

References

1 Richard Bird. *Thinking Functionally with Haskell.* Cambridge University Press, 2015.
2 Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.
3 Manuel M. T. Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1):113–123, 2004. doi:10.1017/S0956796803004805.
4 Manuel M. T. Chakravarty and Gabrielle Keller. *An Introduction to Computing with Haskell.* Pearson SprintPrint, 2002. URL: https://books.google.pt/books?id=qC3dAAAACAAJ.
5 Evan Czaplicki. Elm: A delightful language for reliable web applications. https://elm-lang.org/, 2023. [Online; accessed June 2023].
6 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

7 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. Association for Computing Machinery. `doi:10.1145/582153.582176`.

8 GHC developers. The ghci debugger. `https://downloads.haskell.org/ghc/latest/docs/users_guide/ghci.html#the-ghci-debugger`, 2023. [Online; accessed June 2023].

9 Kees Doets and Jan van Eijck. *The Haskell road to Logic, Maths and Programming.* College Publications, 2004.

10 Mozilla Foundation. Javascript MDN. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`, 2023. [Online; accessed June 2023].

11 Philip J. Guo. Python tutor. `https://pythontutor.com/`. [Online; acessed June 2023].

12 Philip J. Guo. Online Python tutor: Embeddable web-based program visualization for Cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2445196.2445368`.

13 Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell'03), Uppsala, Sweden*, page 62. ACM SIGPLAN, August 2003. URL: `https://www.microsoft.com/en-us/research/publication/helium-for-learning-haskell/`.

14 John Hughes. Experiences from teaching functional programming at Chalmers. *SIGPLAN Notices*, 43(11):77–80, 2008. `doi:10.1145/1480828.1480845`.

15 Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2nd edition, 2016.

16 Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1):49–65, 1993. `doi:10.1017/S0956796800000599`.

17 Simon Marlow. Haskell 2010 language report. `https://www.haskell.org/onlinereport/haskell2010/`, 2023. [Online; accessed June 2023].

18 Jan Paul Posma and Steve Krouse. Lambda lessons. `https://stevekrouse.com/hs.js/`, 2014. [Online; accessed April 2023].

19 Judith Segal. Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, 22:385–411, 1994.

20 Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 3rd edition, 2011.

21 Ville Tirronen, Samuel Uusi-Mäkelä, and Ville Isomöttönen. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming*, 25:e11, 2015. `doi:10.1017/S0956796815000179`.

22 Philip Wadler. A critique of Abelson and Sussman or why calculating is better than scheming. *SIGPLAN Notices*, 22(3):83–94, 1987. `doi:10.1145/24697.24706`.

23 Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. WeScheme: The browser is your programming environment. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 163–167, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/1999747.1999795`.