Design of a Simple Functional Programming Language and Environment for CS2

Brian T. Howard
Department of Computer Science, DePauw University

Abstract

There are several advantages to introducing a functional language early in a student's college experience: it provides an excellent setting in which to explore recursively-defined functions and data structures, it encourages more abstract thinking, and it exposes students to a language paradigm that is likely quite different from their previous experience. Even in a core curriculum based on a traditional imperative (and object-oriented) language, it is valuable to spend two or three weeks investigating a functional language. However, we have found that most existing functional languages and environments pose significant hurdles to the introductory student, especially when the language is only being used for a short time. This paper discusses some of our ideas to simplify the framework, and allow students to experiment easily with the important concepts of functional programming in the setting of CS2.

1 Motivation

There have been many proposals over the years to incorporate functional languages into the introductory computer science curriculum, dating back at least to the mid-1980's with Abelson and Sussman's influential text, *Structure and Interpretation of Computer Programs* [1]. They advocated the use of the Scheme dialect of Lisp because of its simple syntax and support for powerful abstraction mechanisms. Some more recent course designs [2, 3, 10] have also used Scheme, while others have used statically-typed languages such as Miranda, Haskell, or Standard ML [4, 6, 13, 15, 17, 18]. In each case, the reasons given for choosing a functional language include the support for abstractions (including recursion and higher-order functions), the simple semantics (with few or no side-effects), and the exposure to a different language paradigm and problem-solving style.

At our institution, we have not been willing to shift over our entire first or second course to use a functional language exclusively, but for a number of years we have included a short (two or three week) unit on functional programming in our Computer Science 2 course. The unit occurs just after the introduction of recursive function calls in the imperative language (currently Java) and before a discussion of abstract data types, starting with lists. In the unit, we explore typical patterns of structural and generative recursion on numbers and lists. Until recently, the language used was Scheme. When this author started teaching the course, the decision was made to switch to Haskell, in part because of the author's strong previous experience with statically-typed languages and also because a significant number of students were having difficulties with the Scheme material.

After one year's experience using Haskell (specifically, the Hugs system [16]), a project was begun in the summer of 2003 to develop a functional language (which we have named HasCl, for

"Haskell, C-like") and programming environment (named FUNNIE, for "Functional Networked Integrated Environment") specifically tailored to our needs in CS2. This paper is a report of the design decisions that went into the new system. The guiding principles behind the design were that the language should be modeled on the most important features of Haskell, should provide minimal hurdles to students "taking a break" from an imperative language such as Java (since we wanted to spend as little time as possible talking about syntactic issues during the unit), should leverage students' intuitions about function evaluation from high-school algebra, and that the environment should be attractive and easy-to-use.

2 Related Work

The Scheme community itself is well aware of some of the difficulties the language presents to the beginning programmer, particularly with the use of Abelson and Sussman as an introductory text; see for example [9, 22]. Indeed, Wadler's critique identified many of the features we favor from Haskell that are missing in Scheme: pattern-matching function definitions, a mathematics-like notation, static typing, user-defined types, and lazy evaluation. Discussing his experience with teaching the language, Wadler says, "I did not feel that the syntax or idiosyncracies of [Scheme] would be a major barrier. Experience has convinced me otherwise. Although each difficulty by itself is minor, the cumulative effect is significant." [22, page 93]

The TeachScheme! project [10, 20] and the DrScheme environment [7, 11] were developed in part as a response to these problems. By defining a series of "language levels"—subsets of the full language—the DrScheme environment is able to guide the beginners through a much simpler language initially, with correspondingly helpful error messages if the student tries to write something that might be legal in full Scheme but which is inappropriate at the given level. In addition, the project's textbook [10] advocates a design discipline which performs a type-driven case analysis of the data, analogous to that supported by the pattern-matching style of the statically-typed languages. Another important part of the DrScheme system, which directly influenced our own design, is the Stepper, which illustrates the execution of a program by presenting a sequence of algebraic substitution steps.

There are several implementations of Haskell oriented toward educational use. The Hugs system mentioned above [16] is a fairly small implementation of almost the full Haskell 98 language, with an interactive execution console and support for displaying graphics (this is used heavily in the multimedia programming approach of [15]). Unlike the DrScheme environment, which provides an integrated editor, Hugs requires an external tool to edit program source code. A similar environment is provided by the Helium system [14], which has a compiler for a subset of Haskell, designed to be easier to learn. As with the DrScheme language levels, by restricting the language the compiler is able to give more meaningful error messages to the learner. It should be noted that the Glasgow Haskell Compiler [12], generally seen as the standard full implementation of the language, set the model for the Hugs and Helium environments—it too uses an external text editor to develop source, and provides an interactive execution prompt. GHC also supports the same graphics facilities as Hugs.

The Vital project [21] is a more radical implementation of a subset of Haskell. It presents a spreadsheet-like document view of a program, where source code and expressions are intermixed with results. Results can be graphical as well as textual; for example, a list might be displayed as a series of linked boxes. The details of the display can be modified by applying different style-sheets. As with a typical spreadsheet program, individual cells are edited in a text box. Changing the definition of a cell causes re-evaluation of dependent expressions in the document. Only those parts of the document which are currently in view need to be evaluated, so it is easy to evaluate an infinite list and then scroll the screen sideways to see more and more of the list appear.

3 Syntactic Issues

In selecting an appropriate subset of Haskell for our system, we were motivated to minimize the differences from the language of the rest of the course, Java[1]. We knew that we were going to be generating our own course materials for the Haskell unit, so it was not even necessary to restrict ourselves to a strict subset of Haskell, although for consistency—for example, when the full Haskell language is taught in the upper-level Programming Languages course—we tried to stay as close as possible.

One example which seems trivial, but which frequently trips up the newcomer to Haskell, is the fact that parentheses are not required around function arguments. However, parentheses *are* needed when the argument is itself a function application or an expression involving an operator (since all of the operators have a lower precedence level than function application). That is, in full Haskell you may write `factorial 10`, but you must insert the parentheses if you want `factorial (n-1)` or `factorial (twice n)`. This is especially likely to catch the beginner when defining functions by pattern matching—if you try to define the `length` function on lists as follows:

```
length [ ] = 0
length x : xs = 1 + length xs
```

you will get an error because it parses the left-hand-side of the second rule as `(length x) : xs`, which is not legal. We avoid this whole problem by requiring Java-like parentheses around all arguments, so the above function has to be defined as

```
length([ ]) = 0
length([x : xs]) = 1 + length(xs)
```

The extra brackets around the argument of the second rule are explained in the next section. Since parentheses are frequently needed, and since students will expect them from other languages and also from common mathematical notation, this requirement is a small price.

Going along with this decision is a preference for having multi-argument functions take a tuple of arguments, rather than the more idiomatic "curried" form, where the result of applying the function to the first argument is another function which is applied to the next argument. For example, we define the standard function `map`, which takes a function and a list and returns a new list with the function applied to each item in the list, as follows:

```
map(f, [ ]) = []
map(f, [x : xs]) = [f(x) : map(f, xs)]
```

The type inferred for our version of `map` is `((a) -> b, [a]) -> [b]`. In standard Haskell, the type of `map` is `(a -> b) -> [a] -> [b]`, and an example of applying it is `map factorial [1 .. 100]`—that is, the result of `map factorial` is applied to the argument `[1 .. 100]`. We agree with Chakravarty and Keller [4] that, for introductory programming, advanced FP techniques such as

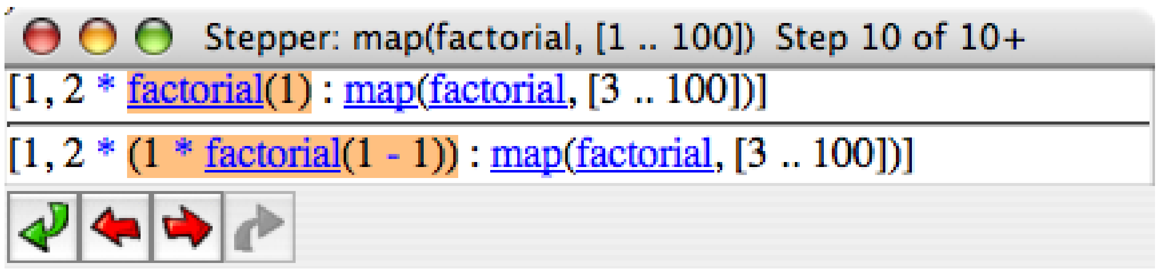
[1]In fact, at the time of the initial design, our CS2 course was taught in C++, but we knew that it was going to change the next year, and the two languages are syntactically very similar anyway.

currying and higher-order functions (beyond `map` itself) are to be avoided, so there is no loss in requiring that a function be provided with all of its arguments at once in a tuple.

One more syntactic issue which we will just mention briefly is that we chose Haskell's explicit layout, with braces and semicolons, rather than the elegant but problematic indentation-based implicit layout. Again, this matches the syntax students will expect coming from Java.

4 Semantic Issues

We have already discussed our desire to take advantage of our students' intuition about evaluating a functional program by repeated algebraic substitution. Haskell's purity (lack of side-effects) and lazy semantics are exactly what we need to enable this substitution—anytime there is a subexpression matching the left-hand-side of one of the equations in the program, it can be replaced by the appropriate instance of the right-hand-side. We took advantage of this to implement an algebraic stepper, similar to the one in DrScheme, so that the user can trace through the steps of program execution. Here is a screenshot of the stepper window evaluating the expression `map(factorial, [1 .. 100])`:



Stepper: map(factorial, [1 .. 100]) Step 10 of 10+

```
[1, 2 * factorial(1) : map(factorial, [3 .. 100])]
```

```
[1, 2 * (1 * factorial(1 - 1)) : map(factorial, [3 .. 100])]
```

The highlight shows the expression `factorial(1)` being replaced by `1 * factorial(1 - 1)`, in accordance with the rule `factorial(n) = n * factorial(n - 1)`.

To enable this behavior, we had to make one surprising change to the syntax of the language. In full Haskell, the operation of prepending an element `x` to a list `xs` is written `x : xs`. However, if we had stuck with that decision, then the stepper would have had to display the intermediate result above as

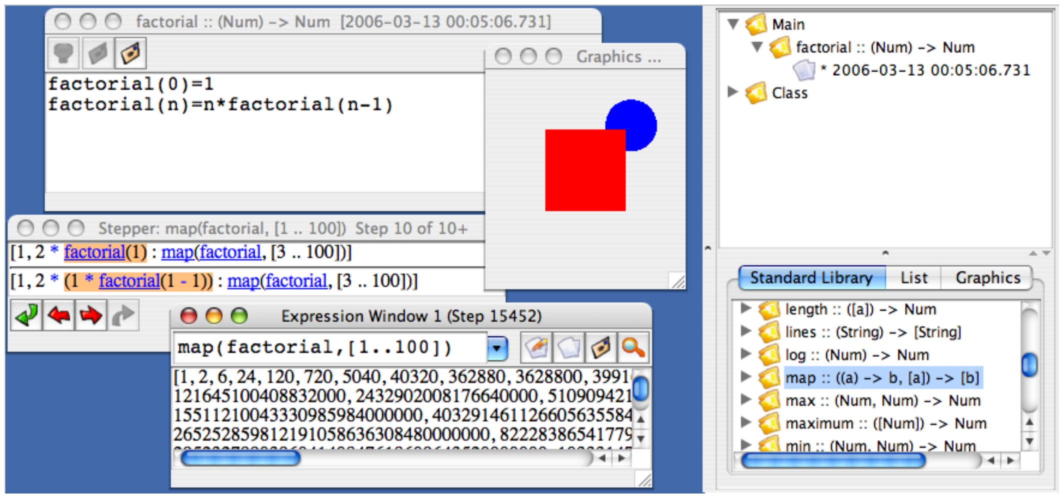
```
1 : (2 * factorial(1)) : map(factorial, [3 .. 100])
```

Only on the last step would the list `1 : 2 : 6 : ...` have been rendered as `[1, 2, 6, ...]`. This is because the Haskell list syntax only allows the brackets around a fully-evaluated list. We took a cue from a relative of Haskell, the Clean language [5], which uses the extra brackets around the list formation operator, `[x : xs]`. We find that this minor change simplifies the student's mental model of list operations considerably, especially in the context of the stepper.

One other semantic issue that we have simplified from full Haskell is the matter of numeric types. Haskell has a rich system (influenced by Scheme) of classes of numbers, including several sizes of integers, floats, and rationals. As a statically-typed language, this necessitates a certain amount of casting back and forth when types are mixed, which can easily frustrate a beginning programmer. We chose to imitate Scheme more directly, and have a single type `Num` of numbers. Internally, the implementation keeps track of whether the number is an int, a bigint, a double, or a rational (pair of bigints), and does its best to choose an appropriate representation for the result of each calculation. Therefore, there is no problem in the above example when it evaluates `factorial(100)` and gets a bigint result with 157 digits. This has also served to motivate a bigint programming project when we move back to dealing with linked lists in the imperative language.

5 Environment Support

Here is a screenshot of the current prototype of FUNNIE in action:


factorial :: (Num) -> Num [2006-03-13 00:05:06.731]
Graphics ...
Stepper: map(factorial, [1 .. 100]) Step 10 of 10+
Expression Window 1 (Step 15452)
Main
factorial :: (Num) -> Num
* 2006-03-13 00:05:06.731
Class
Standard Library
List
Graphics

```
factorial(0)=1
factorial(n)=n*factorial(n-1)
```

```
[1, 2 * factorial(1) : map(factorial, [3 .. 100])]
[1, 2 * (1 * factorial(1 - 1)) : map(factorial, [3 .. 100])]
```

```
map(factorial,[1..100])
```

```
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 3991
121645100408832000, 2432902008176640000, 510909421
15511210043330985984000000, 403291461126605635584
265252859812191058636308480000000, 82228386541779
```

```
length :: ([a]) -> Num
lines :: (String) -> [String]
log :: (Num) -> Num
map :: ((a) -> b, [a]) -> [b]
max :: (Num, Num) -> Num
maximum :: ([Num]) -> Num
min :: (Num, Num) -> Num
```

Visible in this shot are:

- a definition window, containing an editor in which a function may be entered;
- an evaluation window, in which expressions may be entered and their results displayed;
- a stepper window, as discussed above—the arrow controls are single-step forward and backward, and step directly to the beginning or end of the evaluation;
- a graphics window, discussed below; and
- two function browsers, one for the user's definitions and one for the standard library modules.

In the function browser, we keep a history of definitions for each function, organized by timestamp. A user may bring up a definition window on any of the versions, and may choose which one is the "active" definition, to be used in evaluation. Not shown is the importance of the `Class` module in the user's function browser. We have a rudimentary facility for networking an entire class, so that function definitions may be exchanged between students and a moderator. When the moderator sends a definition out to the entire class, it shows up in the `Class` module instead of `Main` to avoid a collision, in case the user already has a function of that name defined.

The graphics window appears when an expression is evaluated of type `Graphic`. At the moment, we support creating simple graphics made up of overlapped rectangles and ellipses, inspired by some of the examples in [15].

6 Future Work

The prototype version of FUNNIE seen above has now been used in our CS2 classes for three years. A newer version is in development with a much-improved evaluator (the prototype is not very efficient, and will easily run out of Java stack space in certain situations) and the concept of a "module window." The idea of a module window is that an entire module of function definitions can be edited in a single window, which will also have tabs corresponding to the evaluation, stepper, and graphics windows. This solves the problem of knowing which module to use when evaluating expressions, since each module provides its own evaluation context.

The windows displaying textual results will have one further enhancement in the new version, inspired in part by the infinitely scrollable document in Vital [21]. When a result is too large to display on a single line, it will be lazily pretty-printed in the available space. Any parts of the result which do not fit in the window will therefore not need to be evaluated, unless the window is resized. The algorithm for this comes from [19].

In addition to displaying `Graphic` values as graphics, we are currently exploring libraries and "visualizers" for music and animation, along the lines of [15] or [21]. Finally, a long-standing goal has been to embed the system in a more powerful and robust development environment such as Eclipse [8], to take advantage of its editing and project management tools. For example, Eclipse provides easy synchronization of a project with a CVS server.

Finally, we are preparing additional teaching materials around our system, and will soon start promoting its use at other institutions. The project has been released as open-source software, at `http://funnie.sourceforge.net/`, and is already being used by at least one other school. After the new version is complete, we will also conduct more formal evaluations of its effectiveness in introducing students to functional programming.

7 Acknowledgments

This work has been performed as part of the DePauw University Computer Science Research Experiences for Undergraduates program, funded by National Science Foundations grant number EIA-0242293, starting in the summer of 2003. Additional funding was provided by the DePauw Science Research Fellows program and a 2004 Eclipse Innovation Grant from IBM.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984.

[2] Dave A. Berque and Gloria Childress Townsend. A new scheme for reinforcing concepts in CS2. In Cary Laxer, Curt M. White, James E. Miller, and Judith L. Gersting, editors, *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education*, pages 327–330. Association for Computing Machinery, 1995.

[3] Stephen Bloch. Scheme and Java in the first year. *Journal of Computing in Small Colleges*, 15(5):157–165, May 2000.

[4] Manuel M. T. Chakravarty and Gabrielle Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1):113–123, January 2004.

[5] Clean. `http://www.cs.ru.nl/~clean/`.

[6] Andrew Davison. Teaching C after Miranda. In Pieter H. Hartel and Marinus J. Plasmeijer, editors, *Proceedings of the First International Symposium on Functional Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 1995.

[7] DrScheme. `http://www.drscheme.org/`.

[8] Eclipse. `http://www.eclipse.org/`.

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, July 2004.

[10] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Srhiram Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

[11] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[12] The Glasgow Haskell Compiler. `\http://www.haskell.org/ghc/`.

[13] Pieter Hartel, Henk Muller, and Hugh Glaser. The functional "C" experience. *Journal of Functional Programming*, 14(2):129–135, March 2004.

[14] Helium. `http://www.cs.uu.nl/helium/`.

[15] Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000.

[16] Hugs. `http://www.haskell.org/hugs/`.

[17] Stef Joosten (ed.), Klaas van den Berg, and Gerrit van der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1):49–65, January 1993.

[18] Tim Lambert, Peter Lindsay, and Ken Robinson. Using Miranda as a first programming language. *Journal of Functional Programming*, 3(1):5–34, January 1993.

[19] Allen Stoughton. Infinite pretty-printing in eXene. In Kevin Hammond and Sharon Curtis, editors, *Trends in Functional Programming*, volume 3, pages 13–24. Intellect Books, 2002.

[20] The TeachScheme! project. `http://www.teach-scheme.org/`.

[21] Vital. `http://www.cs.kent.ac.uk/projects/vital/`.

[22] Philip Wadler. A critique of Abelson and Sussman, or why calculating is better than scheming. *ACM SIGPLAN Notices*, 22(3):83–94, March 1987.