Heat — An Interactive Development Environment for Learning & Teaching Haskell

Olaf Chitil

University of Kent, UK

Abstract. Using a separate editor and interpreter provides many distracting obstacles for inexperienced students learning a programming language. Professional interactive development environments, however, confuse these students with their excessive features. Hence this paper presents Heat, an interactive development environment specially designed for novice students learning the functional programming language Haskell. Based on teaching experience, Heat provides a small number of features and is easy to use. Heat proves that a small portable interactive development environment can be implemented on top of but independent of a particular Haskell interpreter. Heat with Hugs has been used in teaching functional programming at the University of Kent for the past three years.

1 Introduction

Universities teach functional programming in Haskell using a Haskell interpreter: Hugs[1] or GHCi (Glasgow Haskell compiler interactive) [2] or Helium [8]. All these interpreters are command-line based systems where the user loads a Haskell program from a file into the interpreter; if the program is statically correct (no compilation errors), the user can enter expressions that are then evaluated by the interpreter. So a student has to use both an editor and the interpreter. After modifying the program the student has to save the file, switch to the interpreter and load the file into the interpreter before they get any static error messages or can enter expressions for evaluation. Especially students at the beginning of their studies (at Kent we teach functional programming in the second term of the first year) make many mistakes and hence have to repeat the edit–save–switch–load cycle numerous times. Furthermore, in classes at Kent I noticed that students often forgot to either save in the editor or reload the program into the interpreter, so that compiler error messages or the results of evaluation related to an older version of the program, causing substantial confusion to students. Finally, today's students are used to graphical user interfaces, using the mouse and standard key shortcuts for editing — including expressions for the interpreter —, and consider command-line based tools as old-fashioned.

[1] `http://www.haskell.org/hugs/`
[2] `http://www.haskell.org/ghc/`

The answer to this list of problems is to use an integrated development environment (IDE) such as Eclipse[3] or Visual Studio[4] that integrate editor, compiler and program execution plus many further features within a single graphical user interface. Such IDEs are used by many professional software developers and additionally, at Kent, students will already have used BlueJ [9], an IDE for learning and teaching Java programming.

However, we do not teach functional programming using a professional IDE with a plugin for the Haskell language. First of all most plugins are still at an early development stage, installing the IDE with the plugin on their own computers is often too complicated for many students and many IDEs such as Visual Studio [1] do not support all major platforms used by our students (Windows, OS X, Linux variants). Most importantly, these IDEs provide numerous features that students who just learn the main concepts of functional programming do not need, that confuse them and make basic tasks unnecessarily complicate (e.g. such IDEs have a concept of workspace or project).

Therefore we developed Heat, the Haskell Educational Advancement Tool, an IDE for teaching & learning Haskell meeting the following requirements:

1. integrates editor and interpreter console within a single user interface;
2. provides a graphical user interface similar to professional IDEs;
3. is simple; provides only features that support students who are beginning to learn functional programming;
4. is reliable and fool-proof;
5. runs on all major platforms used by our students and staff (Windows, OS X, Linux, Solaris);
6. is easy to install;
7. has small source code and is easy to maintain.

Heat's features include syntax-highlighting, explanations of compiler error messages, code navigation via a summary tree pane, change of font size for lecture presentations and automatic testing.

Requirement number 7 is important in view of the large number of existing half-finished IDEs for Haskell. It implies that Heat does not implement any features of a Haskell interpreter but runs loosely coupled on top of an existing Haskell interpreter, currently Hugs. Heat runs Hugs as a separate process and communicates with it only via textual in- and output. Thus a standard Hugs package can be installed and Heat is independent of specific releases of Hugs.

The requirements for a graphical user interface, portability across platforms and easy installation together lead to the decision to implement Heat in Java. The distribution is a single jar file that the student only needs to click on.

Heat 1.1 has been used annually since the academic year 2005/6 in teaching the module functional programming in the second term of the first year at the University of Kent. The current version is 3.0. Heat is freely available from

`http://www.cs.kent.ac.uk/projects/heat/`

[3] `http://www.eclipse.org/`
[4] `http://msdn.microsoft.com/en-us/vstudio/default.aspx`

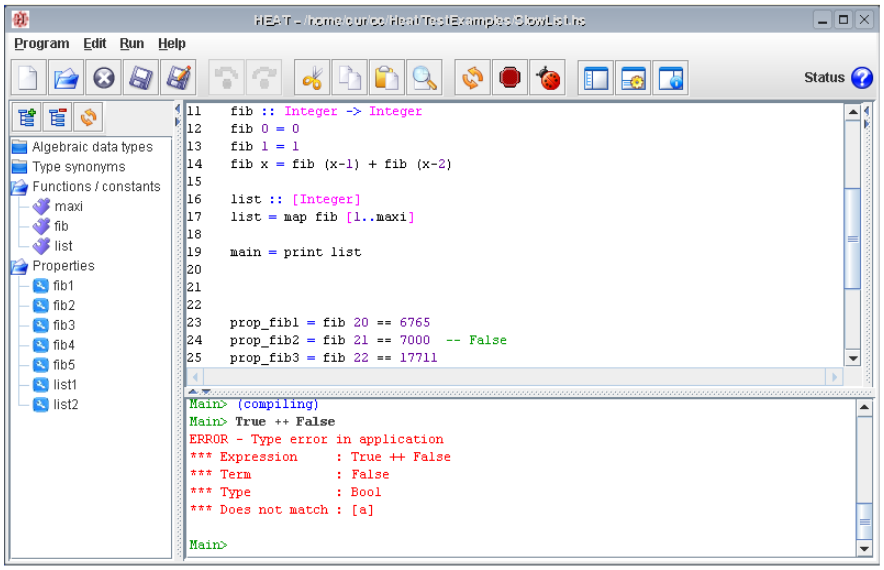

Fig. 1. Screenshot of the Heat window

This paper describes the main features of Heat. The design of Heat has been informed by experience in teaching Haskell and feedback received from students using Heat.

2 Design

Like professional IDEs the user interface of Heat consists of a single window divided into several panes. Thus the user can always see all panes and easily switch activity between these panes. Figure 1 shows the Heat window. The top centre pane is the editor. The bottom right pane is the interpreter console. The (here hidden) top right pane displays help texts. The left pane provides a summary tree and quick navigation to all definitions.

Our students only write programs consisting of a single module, which may use libraries, including specially provided modules. Therefore Heat has no concept of multi-module projects, meeting our requirement of simplicity. Opening a new file replaces the current contents of the editor (after asking whether it should be saved).

The editor provides all standard editing operations, highlighting of matching brackets and syntax-highlighting for Haskell programs. The same syntax-highlighting is used in all example code on lecture slides.

In the top right corner the window displays Heat's current status:

– A blue question mark indicates that the current program has not yet been compiled, it may contain errors.

– A red cross indicates that program compilation produced an error message.
– A green tick indicates that the program compiled successfully and the user may enter expressions for evaluation in the console window.

Only in the last case the user is able to enter expressions for evaluation in the console. Otherwise the console can only be read.

Compilation happens at the press of a button. If compilation gives an error message, the line stated in the error message is shown and highlighted in the editor. If Heat is in novice mode, also an explanation of the error message based on Simon Thompson's list of Hugs error messages[5] is displayed.

Evaluation of expressions works in the console exactly like in a stand-alone interpreter. Highlighting of the prompt and of error messages in different colours improves readability. Any computation can be interrupted by pressing a button.

The summary tree pane provides a list of all type synonyms, algebraic data types, functions and constants, and test properties (see Section 3) of the program. Selecting any of these makes the editor show and highlight the respective definition. Hovering with the mouse pointer over a type synonym or algebraic data type displays a tool-tip containing the definition, hovering over a function or constant displays the respective type. Such overview and navigation is useful not because our students would write large modules, but because they are often given medium-sized incomplete modules which they have to complete. Incomplete definitions are given in the form

```
chessboard = undefined
```

so that even incomplete modules can be compiled and parts can be tested, encouraging an incremental development style.

The few option settings offered by Heat include changing the font size in all window pane components. It is essential to change to a large font size when using Heat in lectures to develop programs interactively together with the students.

3 Checking Properties

Inspired by the book *How to Design Programs* [4] we teach students to define functions by following the steps of a design recipe:

1. Purpose in comment
   ```
   -- Yield square of the input
   ```
2. Type declaration (name the function, decide on types of input and output data)
   ```
   square :: Float -> Float
   ```
3. Example properties (list examples to characterise input-output relationship)
   ```
   prop_square1 = square 2.0 == 4.0
   prop_square2 = square 0.0 == 0.0
   prop_square3 = square (-4.0) == 16.0
   ```

[5] `http://www.cs.kent.ac.uk/people/staff/sjt/craft2e/errors.html`

4. The actual definition body
   ```
   square x = x*x
   ```
5. Test (check that definition meets example properties)

Properties are Boolean constants whose identifiers have the prefix `prop_`. Most of them apply the function to constants and compare with constant expected results. For some types, such as pictures, no suitable constants are available and then other properties are considered:

```
prop_rotate180 = rotate180 (rotate180 horse) == horse
```

All properties are still Boolean unit tests, but they are a basis for introducing more general QuickCheck [3] properties later. Already the simple Boolean tests as above lead to interesting discussions. A simple database using lists of tuples is used to practise list processing. Here the lists actually represent (unordered) bags and hence using a simple comparison with `(==)` quickly proves to be inappropriate, because different implementations yield result lists with different orderings of elements.

Heat supports properties by automatically checking all properties on the press of a single button. Properties are listed separately in the summary tree. Different icons indicate whether a property was not yet checked, evaluated to `True` or evaluated to `False` or a runtime error. Especially to handle non-termination, property checking can be interrupted.

4 Graphics via PDF

We use a version of Simon Thompson's small library for composing pictures, which starts with the picture of a single horse's head. The compositional picture combinators are good for teaching the construction of expressions, writing compositional and reusable definitions, decomposing problem descriptions and writing non-structurally recursive function definitions. Furthermore, while most example programs used in teaching Haskell produce textual output, this library produces pictures. However, the original library produces only ASCII art

```
.......##...
.....##..#..
...##.....#.
..#.......#.
..#...#...#.
..#...###.#.
.#....#..##.
..#...#.....
...#...#....
....#..#....
.....#.#....
......##....
```

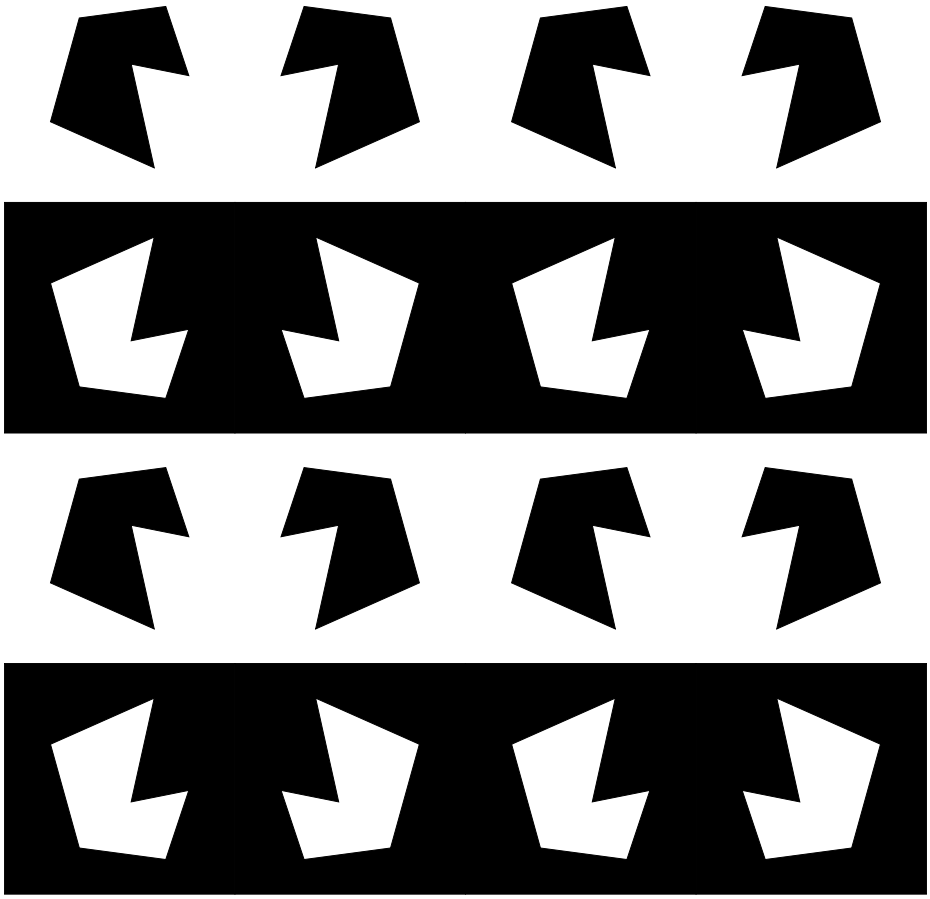

Fig. 2. A PDF picture made with the Picture library

that fails to impress students. Creating graphical pictures in Haskell is highly desirable, but existing graphics libraries do not meet our requirements of being reliable, easy to install and platform independent. Because Heat is implemented in Java it could give such graphic support, but connecting Java and Haskell is not easy.

Instead I developed a Haskell library that produces a picture as a PDF-file. Every platform has a PDF viewing tool which can even be started automatically from the Haskell library. Figure 2 shows a simple picture created with the library. Because the library can still also produce an ASCII picture, it was easy to define equality on pictures, which is essential for defining test properties for functions working on pictures as shown in the preceding section.

The new PDF Picture library is independent of Heat but also works well together with it.

5 Implementation Notes

Instead of implementing jet another editor we aimed for reusing an existing editor package for Java. Few such components seem to be available. We chose to use the jEdit syntax package[6], a stand-alone syntax highlighting text editor JavaBean. It provides syntax highlighting via a user-definable class that implements lexical analysis of tokens. Many students mix up types with other Haskell expressions, especially type constructors with data constructors. Hence it would be desirable to distinguish types from other expressions through highlighting in different colours. However, as in other editors where syntax highlighting is defined by regular expressions, this cannot be achieved easily and hence has not been implemented in Heat.

Heat starts Hugs as a separate process and communicates with it only via the textual input and output stream. This weak interface makes the implementation of a console surprisingly hard and explains why there does not exist any reusable console package for Java. The console provides full editing facilities for the command (usually an expression) sent to the Haskell interpreter. However, when a Haskell program runs interactively, the console must pass all key input unchanged to the Haskell program. The console must be able to distinguish these two states and it does so by recognising the prompt in Hugs' output. To recognise the prompt reliably, Heat changes Hugs' prompt to include unprintable characters that it can easily recognise. All other syntax-highlighting in the console is based on simple syntactic cues in Hugs' output. A Java thread is needed to copy the output of Hugs character by character as it is produced into the console pane, because the user needs to see also a partial result of evaluation, for example the initial part of an infinite list. The Process class of Java provides no method for sending an interrupt signal to the Hugs process. So to interrupt evaluation Heat actually stops the Hugs process and restarts a new one, omitting the start-up banner.

To collect information about definitions for the overview pane a simple parser regularly traverses the editor contents. Unfortunately Hugs' own symbol tables that can be accessed with the `:browse` and `:info` cannot be used, mainly because it exists only after compiling a statically correct Haskell program. In contrast, the simple Heat parser identifies definitions at the hand of a few keywords such as `data` and `=` and also assumes that all top-level definitions start in the first column.

Accessing Hugs only via its textual command interface limits the features of Heat. Hence the Glasgow Haskell compiler provides an application programming interface (API) for developing new user interfaces on top of it. However, more features are non-essential (compare for the next section) and using such an interface would have made it hard to meet our requirements of easy installation, support of all platforms and easy maintenance.

Implementing all compositional picture combinators of the Haskell Picture library in terms of the incremental graphics model of PDF proved to be hard. The

[6] `http://sourceforge.net/projects/jedit-syntax/`

Picture library includes a combinator that inverts the colours of a given picture. There is no such operation in PDF where every graphical output paints on top of the existing picture. To keep the library simple it does not handle interaction of the invert operation and the superimpose operation of two pictures correctly; this limitation does not affect any examples and exercises used in the course.

6 Evaluation

We have used Heat 1.1 in the functional programming module in the second term of the first year at the University of Kent. In the first term students had learned basic Java programming using the BlueJ IDE. We demonstrated and used Heat in the lectures and advised them to use it in classes.

Most students used Heat in classes and on their own laptop or computer at home. Some students, mostly more experienced ones, preferred to use WinHugs or Hugs in a normal console together with their favourite editor. Student feedback on Heat at the end of the term was mixed. Many students recognised its usefulness in principle but complained about bugs.

Most complaints were about one bug: occasionally compiling a correct Haskell program produced the Hugs error message

```
Syntax error in input (unexpected ‘;’)
```

To remove this wrong error message the user has to delete the offending lines and retype them. The Hugs error message is caused by spurious `\r` characters in the code, that are invisible in the editor pane (only `\n` and `\n\r` are line breaks). Apparently the spurious `\r` is occasionally caused by copy and paste operations in the editor. They do not occur on the Solaris platform used by teaching staff and seem to occur most frequently on Vista. Hence originally I didn't consider this bug as serious. Because we could not reliably reproduce it and is located within the reused editor component, I finally resolved it by removing any `\r` characters when saving the editor contents to a file for passing to Hugs.

Other reported bugs included the lack of shortcut keys for commands such as saving and undo; not supporting OS X specific shortcut keys; not supporting automatic indentation. Many bug reports demonstrated the inexperience of the students by being too vague to be comprehensible.

When Heat is started for the first time after installation, it asks for the full path of the Haskell interpreter Hugs. Despite clear instructions, a number of students mistakenly entered the full path of Heat itself. Then Heat starts another Heat process, which reading its settings starts another Heat, ... The student has an early exposure to unbounded recursion that is hard to stop in practise. Therefore it proved useful for later versions of Heat to explicitly test that the given path is not that of Heat itself.

In conclusion experience and student feedback show that reliability is a central requirement for a teaching tool and a single annoying bug can turn many students against the tool. The expectation that a graphical user interface provides all common standard operations is also important. In contrast a lack of

features seems to be unproblematic. I never received any complaint that Heat 1.1 could not interrupt a non-terminating computation of Hugs; the user had to abort Heat. Furthermore, even awkward functionality is surprisingly widely accepted. Heat 1.1 did not have a fully functional console, but only an output pane with input provided awkwardly via a separate input box. Few students complained and asked for a proper console.

7 Why choose Hugs?

The Glasgow Haskell compiler (GHC) is used by most serious Haskell programmers because of its large set of libraries, generation of fast code and language extensions. For teaching novices, however, we prefer to use Hugs. Although the error messages of GHC are more detailed and often clearer than those of Hugs, the latter are more suitable for novices. Hugs stops compiling after the first compile time error, whereas GHC often shocks the novice with a long sequence of error messages. Error messages of GHC contain language extensions unknown to novices, such as explicit `forall` quantification of type variables. Error messages use substantial indentation so that they take far more space than Hugs' error messages, which makes them particularly unsuitable for lecture presentations and slides. If there is an error in an input expression, the error location given by Hugs is rather confusing. Finally, on many platforms GHC is still much harder to install than Hugs.

Helium [8] is a Haskell interpreter specially designed for teaching. It provides sophisticated error messages with detailed explanations and suggestions for correction. However, Helium does not implement full Haskell 98, lacks some documentation and has the chicken-and-egg problem of not being widely used.

So for teaching we at the University of Kent prefer using Hugs. However Heat should still support both GHCi and Helium in the future. This extension should be simple, as discussed in Section 9.

8 Related Work

Heat was mostly inspired by BlueJ [9] and DrScheme [5]. BlueJ supports the objects first approach of teaching Java by making objects visible as icons whose methods can be executed interactively using the mouse and dialogue boxes. This support for objects is not applicable for a functional programming language but the interpreter console is a simple fitting replacement. Dr Scheme integrates an editor and an interpreter console similar to Heat plus many further features. Most advanced features of BlueJ and Dr Scheme rely on these IDEs being closely linked with the language compiler and runtime system, unlike Heat.

Vital [7] is inspiration for a novel and different IDE for functional programming languages that could support learning and teaching. Unfortunately its concept requires a tight integration of user interface and interpreter; Vital implements its own untyped subset of Haskell.

WinHugs comes with the Hugs distribution and is Hugs with a graphical user interface. It provided some inspiration for Heat but it does not integrate an editor and it only runs on Windows. WinHugs is integrated into the source code of Hugs; it is not a separate program running Hugs as a separate process.

Many functional programmers use emacs, vim or similar advanced editors with special extension modes that provide the interpreter console within an editor window. Most of these modes still do not provide the close integration of editor and interpreter provided by Heat, the editors provide confusing excessive features, and the graphical user interface does not meet the expectations of the students. Therefore the Heat user interface looks like a substantially simplified variant of professional IDEs such as Eclipse and Visual Studio.

Yi [2] an editor written in Haskell and extensible in Haskell does not provide an IDE for Haskell and is not targeted at supporting programming novices.

Many projects have aimed for developing a special IDE for Haskell implemented in Haskell: IDE, Haste and most recently Leksah[7]. Because of high ambitions and a single developer these projects have not yet delivered.

9 Future Work

Novice programmers do not write large multi-module programs, but they use libraries. Hence students would profit from the inclusion of library documentation within Heat. Heat could either provide access to library documentation generated with Haddock, the Javadoc-like Haskell documentation tool, or simply provide specially written documentation in the help pane. The former has the advantage that libraries can easily be added, the latter is more suitable for providing documentation at the right level for novices. The Haskell prelude is big and unstructured and many parts are incomprehensible for novices.

Many compile errors are due to students misspelling identifiers. Often they fail to see the spelling mistake and therefore waste substantial time for removing such errors. Heat should help students with preventing such errors. Modern IDEs even help further by providing auto-completion of identifiers. However, correct auto-completion requires knowing which identifiers are in scope at a given point of the program and thus requires parsing of incomplete and probably incorrect programs. Developing such a parser fails our requirement of a simple implementation using the given interpreter. An intermediate solution providing limited spell-checking may be possible. Heat 2.0 included a prototype version.

Heat 3.0 can only check Boolean properties automatically. To support all QuickCheck [3] properties Heat would need to recognise when full QuickCheck is used and process the result of testing appropriately.

Heat 2.0 provided simple support for debugging Haskell programs by observing functions using Hood's observe combinator [6] that is built into Hugs. However, this built-in observe combinator often produces wrong output in current versions of Hugs and hence Hugs needs to be corrected first. The Hood library

[7] `http://leksah.org/`

is less suitable for Heat, because it requires instances of the class `Observable` to be defined for all types used in observations.

An orthogonal direction for future work is making Heat work with other Haskell interpreters than Hugs, especially GHCi and Helium [8]. The textual interfaces of both these interpreters are similar to Hugs. Because all Hugs-specific code is within a single class that handles the interaction between Heat and the Haskell interpreter, it will be easy to support other Haskell interpreters. Additionally, an XML file with simple explanations of the error messages would be needed for a new Haskell interpreter; this feature may be superfluous for Helium. However, to ensure that a Heat user can choose between several Haskell interpreters some further extensions of the options and allowing several Haskell interpreter interaction classes will be needed.

Finally Heat could also serve as a framework for simple interactive development environments for other functional programming languages. Only a few classes such as those for syntax highlighting, the parser for the summary tree and automatic testing are Haskell specific.

10 Conclusions

In this paper I have presented a simple and small but effective graphical integrated development environment that supports inexperienced students in learning a new programming language. I gave examples of how Heat smoothly integrates with our teaching. Most of the design decisions and features are independent of the programming language that the tool supports and thus can be transferred. In practise the most important feature of such a teaching tool is reliability; occasional but hard-to-circumvent faulty behaviour can quickly turn students into strong opponents of the tool. Java is a good programming language for building such a graphical, portable and easy-to-install tool, but unexpected portability problems do exist. Although Heat was planned as a small wrapper around a Haskell interpreter its development cost far more time and effort than initially expected. Still, Heat has already supported many students at Kent in learning Haskell and I hope Heat 3.0 will be used more widely.

History and Acknowledgements

I thank all the developers of Heat. Heat was designed and implemented mostly by students. Heat 1.0 was created by Dean Ashton, Chris Olive, John Travers and Louis Whest as their final year project in 2004/5 at the University of Kent. In particular the first two students are responsible for the good initial design and architecture of Heat that proved to be a reliable and well-structured basis for later extensions. A slightly improved Heat 1.1 by me has been used in teaching since then. In 2006 Jerome J. S. Gedge, Sergei Krot, Stefanos Katsantonis and Danya Nusseir developed Heat 2.0 as their final year project, adding code navigation, checking properties, debugging using Hugs' `observe` function, auto-completion of identifiers and numerous small but useful improvements. In

summer 2008 Ivan Ivanovski, an IASTE student, improved several features including the console and property checking. I improved the code further to obtain Heat 3.0, the first public release.

References

1. Krasimir Angelov and Simon Marlow. Visual Haskell: a full-featured Haskell development environment. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 5–16, New York, NY, USA, 2005. ACM.
2. Jean-Philippe Bernardy. Yi: an editor in Haskell for Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 61–62, New York, NY, USA, 2008. ACM.
3. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
4. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* The MIT Press, 2001.
5. Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: a pedagogic programming environment for Scheme. In *In Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, 1997.
6. Andy Gill. Debugging Haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. 2000 ACM SIGPLAN Haskell Workshop.
7. Keith Hanna. Interactive visual functional programming. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 145–156, 2002.
8. Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71, New York, NY, USA, 2003. ACM.
9. Michael Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4), 2003.