Froskell programming language

Alexander Berntsen alexander@plaimi.net
Stian Ellingsen stian@plaimi.net
Olle Fredriksson olle@plaimi.net

16th January 2015

Abstract

Due to the inherent complexity of computers and computer programs, bog-standard computer users are destitute for knowledge. Learning how to program computer software is an enjoyable and engaging way to teach users a concrete skill, and at the same time provide the necessary foundations for further self-education towards computer erudition. However, the current tools that are used in teaching have several problems, including bad error messages and being difficult to install for non-technical users. We repent the present situation, and present the idea of an implementation of the Haskell programming language with unlockable features as a solution. We also offer musings on an interactive online IDE idea to get users quickly up and running with our language.

Contents

1. The problem

When teaching programming languages, the choice of language is important. Some teachers opt for using languages tailor-made for teaching. These generally fall into two categories: those that are a simplified version of a general-purpose language, and those that are completely new languages. One benefit of both language categories is that their error messages can be better, for instance by never containing references to advanced features that the student has not yet

been taught. As an example, a Haskell student that writes

```
x = abs -3
```

is met with a long error message that contains the following snippets:

```
No instance for (Num a0) arising from
  a use of 'abs'
The type variable 'a0' is ambiguous
```

and

```
No instance for (Num (a0 -> a0)) arising
  from a use of '-'
In the expression: abs - 3
```

The actual error is that the code is interpreted as `abs - 3`, i.e. that the minus sign is taken as subtraction of `3` from `abs` and not as negation. However, the error message speaks of `Num` typeclass instances and type variables – concepts that a beginner Haskell programmer is almost guaranteed to not know.

When using a teaching language we can also avoid "magic incantations". As an example, take the following Java program that prints "Hello world!" to the console:

```
public class HelloWorld {
  public static void
  main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

To teach what this program does the teacher has to either tell the student to not worry about the first three lines for now, probably leaving the student in a puzzled state, or hold a long lecture about concepts not entirely relevant to a beginner.

Another difficulty in teaching programming is that it is often complicated for students to install the programming environment on their own machines. This occurs for several reasons, for instance because the language might have bad or no support for certain operating systems, or simply because the student does not yet have enough computer experience. It can also be difficult to provide consistent versions of the language implementation and any required libraries. These difficulties lead to the teacher having to provide what amounts to technical support instead of being able to focus only on programming.

2. Our idea

To mitigate the problems that beginner programmers often face, we want to make a programming language implementation with unlockable levels of features (subsection 2.1) and an online *integrated development environment* (IDE) (subsection 2.2).

2.1. Language

We want to make an implementation of the Haskell programming language [1] with unlockable sets of features. Haskell is a general-purpose programming language suitable and often preferred for use as the first taught language in computer science courses [2]. It is regarded as an elegant programming language that allows writing programs in many styles, including denotative and imperative.

With unlockable features we mean that a student may start out with a language with very few features turned on. As a result, we can provide very relevant error messages – tailored to their level of expertise – that do not mention features that they have not yet encountered. As the teaching progresses, we can turn on increasingly advanced features of the language. Since we are using an already established, general-purpose language, the knowledge gained using our implementation transfers to ordinary implementations of Haskell and similar languages, and by having unlockable features we avoid the problems associated with using a general-purpose language for teaching.

2.2. Integrated development environment

To solve the problem of installation difficulties, we propose to provide an online IDE for our language. This means that a student can simply point their web browser to a website providing the IDE and whammy!, immediately start writing and running code in the browser; the barrier

to entry is extremely low. It also means that we can provide the same up-to-date version of the language and its libraries to everyone.

An online IDE opens the door to many features that will help the students to learn or just make programming exciting:

Collaborating Students can form groups and collaboratively write code. Since the IDE is online they can do this even if they are not physically in the same location, and they can edit the same file simultaneously.

Sharing Students can easily share their code and applications with friends and classmates.

Writing games We can provide libraries to write games that run in the browser, and provide easy access to art assets in the IDE. Games are fun and rewarding to hack, which should help to increase the adoption of the language. It also lets us focus development, and sets us apart from other educational programming languages.

Learning Teachers can provide learning material with exercises that students can do and have automatically marked directly in the IDE. Language features (or even game assets) can be unlocked as the students progress through the exercises. It is possible to envision branching capabilities in the exercise machinery, so that the user could be prompted for what they would like to learn next. Throw some learning analytics at this baby, and we have informed suggestions as to what to learn next as well.

2.3. Academic use

We have already touched on a few uses of our language and IDE in the academic sector, but there are many more possibilities.

Our IDE could be specialised for programming courses, with virtual classrooms where students and tutors could meet to discuss the course and its contents. Teachers could author their own custom curricula and exercises, and we could integrate the IDE with their learning management system.

By providing different libraries and toggling different sets of features of the language, it could be customised to suit many diverse kinds of courses at different levels in the education system, be it programming fundamentals at the university level or game development for kids.

It is likely that teachers would be very interested in grading systems, and integration with learning management systems. Generated reports would be useful for students and teachers both.

We should try to get some schools or computing groups to use our environment as we develop it, to get useful feedback. Our background in academia will help us finding interested candidates.

2.4. Commercialisation

We would like to keep the basic IDE gratis for everyone, but we still have several ideas that make the idea commercially viable:

Premium features The classroom features could for instance be a premium feature that schools and universities would have to pay for.

Art assets Art assets for use in games and applications could be sold directly in the IDE. Or, we could integrate with existing art asset services. Users should also be allowed to upload their own art assets gratis.

Application store Advanced users could be allowed to publish and sell their work in an associated application store in return for a portion of their revenue. Maybe they could also sell "base games" that are intended to be customised and tweaked into full games, as a learning exercise. Teachers could sell course material as well.

2.5. Societal benefits

Whilst user interface designers keep telling us that user interface design is constantly improving, there is no denying the inherent complexity

of a modern day computer. In a society where only software developers understand the basic science of a computer program, unenlightened computer users are left helpless. This is problematic in the case of proprietary software, where computer users are slaves to the subjugation of the power elite created by these software developers – and worst of all, they often don't even realise it. Learning to program a computer will as a side-effect make computer users more aware of how computers actually work, which in turn gives them the foundation for self-educating further. Moreover, understanding the basics of computer programming means understanding what source code is, and why it must be free for computer users themselves to be free. Illuminating users in this manner is close to the zenith of societal contribution in computer science.

Another aspect to consider is that teaching computer programming in a principled way is The Right Thing. Computer programmers today are typically either lone self-taught hackers, or computer science students. Both of these groups suffer setbacks from learning to program through pedagogically unsound tools. Others never make it to an enlightened state at all. Society at large is starting to take this seriously. Computing at School[1] has been successful in improving national school curricula in the UK, and in Norway we have Lær Kidsa Koding[2]. We believe that our language could have a profound impact in these circles.

2.6. Timetable

Having unlockable subsets of the language makes it natural to develop the language iteratively. We will design the first stage, the basic level of the language, very meticulously, and merely delineate future stages. Then, as development goes on, we will continue this trend of meticulousness for short-term goals, and nice academic hand-waving of the future. It's nice our degrees weren't for naught.

After around 15 months of work, we should have all the complicated language design decisions mostly out of the way (though subject to change). A preliminary implementation of the unlocking concept should be in place, and the basic first level of the language, along with some basic libraries, should all be available. We should also have a third-party IDE connected to our language, for proof-of-concept. Finally, we should also have delineated some of the immediate next levels.

3. Related work

In this section we look at some existing solutions to the described problems, and why they are not ideal. Let's start with some of the most popular historical languages.

3.1. Historical languages

BASIC was authored to provide a programming language which would be easy to learn for students without a rigorous mathematical background. It came about in the 1960s, designed for use with Darthmouth's timesharing system, and became truly influential during the home computer revolution of the 1970s. There have been several versions of BASIC since the original; the most notable dialect arguably being Microsoft's Visual Basic [3].

Pascal was designed in 1971, partly as a simplified version of Algol, partly as a language that encouraged structured programming. It was designed for educational purposes, but evolved into a popular general-purpose programming language [4]. The initial versions were criticised for not being suitable for "real world" programming [5]. Several versions and dialects of Pascal have since emerged [4], that eliminate these problems.

Scheme is a Lisp programming language that was originally designed with tutorial purposes in mind [6]. It was used in the influential book Structure and Interpretation of Computer Programs, which was used at MIT to teach programming. The success of the book and the language itself later lead to Scheme becoming

[1] http://www.computingatschool.org.uk/
[2] http://www.kidsakoder.no/

a popular choice for introductory programming courses at other universities too [7].

Logo is another Lisp programming language, made specifically for teaching programming to children. A notable part of several Logo environments is the use of a turtle avatar that moves around the screen and draws things. It is an influential language which has seen a lot of adaptation since its inception [8].

Though we have much to learn from the languages mentioned here, none of them solve any of the problems we have identified. Furthermore, they are all antiquated.

3.2. Helium

Helium [9] is a dialect of Haskell specifically made for teaching. It focusses on good error messages. The implemented dialect is not full Haskell since it does not include typeclasses. The work on Helium might provide an inspiration for how to implement good error messages, but other than that our work will extend on its functionality in several important ways. Helium does not provide different levels of functionality, and requires a local installation of full Haskell before it can be installed.

3.3. Alice

Alice is a language and environment designed exclusively to teach the concepts of object-oriented programming. It's a drag and drop environment in which the user makes animations by placing 3D models and scripting their behaviour by dragging and dropping control structures (loops, if-statements, and so on) [10]. Alice has multiple shortcomings, including being proprietary, requiring a local install, only vaguely teaching object-oriented concepts, not being even remotely comparable to "real world" coding, having a thoroughly confusing user interface, and so forth. Alice falls short on all of our described problems.

3.4. Mozart

Mozart is an implementation of the Oz programming language. It is featured in the MIT textbook Concepts, Techniques, and Models of Computer Programming. Mozart consists of different subsets that are toggled as the user progresses in their learning [11]. This is a core concept in our language, so we should familiarise ourselves with how Mozart works. Mozart does not, however, offer an online IDE with exercises etc. to make it easy to get started with. It moreover appears to not have received widespread usage.

3.5. Racket

Racket is another Lisp language. What's cool about Racket is that you may enable and disable language features quite freely in the runtime system. This lets you (amongst other things) make what they call "Teachpacks", libraries written in the full language that work with the currently activated subset [12]. In theory you could make a lot of what we are suggesting to do with our language with Racket, though we would argue that using a dynamic language that permits side-effects is categorically The Wrong Thing. Racket might in any event be the most relevant work of all. We should study it closely!

Our idea is still a bit better though. We have an online IDE, remember? What is more, we can have nice exercises that guide the user through the learning process in this IDE. And our language doesn't launch missiles by sheer happenchance.

3.6. Online interactive IDEs not necessarily aimed at teaching

There are several online interactive IDEs that we can learn from. They are not aimed at teaching and as such do not use informed and principled pedagogic methods to teach users programming, but rather act as supplementary resources when learning programming, providing the users with challenging tasks and often a game-ified environment with e.g. achievements or points. In this section we discuss a few of them.

`http://www.codewars.com` lets users train on programming challenges, awarding points for doing so. Users can also vote for the best solu-

tions to a task, and advanced users may author problems themselves. The site supports several programming languages, including Haskell, Clojure, and Ruby. While not useful to learn programming per se, the challenges can act as neat supplements to someone who is learning to program. The IDE lets the user hack solutions in the browser (it includes emacs and vi input modes), and also evaluate whether the solution is correct via the browser. Several of the problems are well-authored with nice unit tests to help the user understand the problem at hand, and the IDE is mostly pleasant to use.

http://www.codingame.com/start offers very visual game programming challenges. Once again there's an online IDE (with emacs and vi input modes), unit tests to guide the user, and in-browser verifying of the user's solutions. The site offers a huge selection of languages, including Haskell, C, and Java. Interestingly the huge selection of languages seems to be a negative, in that they likely have a very limited API for their challenges, leading to all languages feeling slightly awkward in practice. The code you need to write is furthermore often far detached from "real world" code.

http://www.playmycode.com lets users hack and play Ruby games, as well as share them with others. While testing we encountered bugs in the most popular games – although the bugs could be related to the HTML "game player" rather than the games themselves, it's hard to tell. The IDE is severely handicapped as it does not offer emacs or vi bindings, but is at the very least functional. The site lets you upload graphics, and offers an online graphics editor as well. The latter would likely be avoided by experienced artists, but is a good example of the low barrier to entry mentality that we want.

http://elm-lang.org/try lets you hack Elm code interactively. You may compile the code and run it in the browser, and you can even hot-swap code for a running program. The Elm developers have been working on a time-travelling debugger in which you may turn back time in your running program, change the source, and resume the program, seeing previous paths being played out at the same time. It would be natural to assume that this will be featured in the IDE sometime soon. Note that this is just an online IDE. There is, however, http://share-elm.com/ for sharing Elm code, and http://elm-lang.org/Examples.elm has a bunch of Elm examples which you may open in the online IDE. Despite the Elm chaps being clever enough to figure out time travel, they don't have emacs or vi bindings for their IDE. Lacking this is of course the nadir of IDE design. We should learn from the great debugging facilities offered by Elm's IDE.

http://www.fpcomplete.com/page/project-build offers an online IDE for Haskell where you can make projects, compile them, and run them. It also offers the use of libraries, and the IDE features vi and emacs bindings. It is a strong contender that shines with its low barrier to entry. It is, rather unfortunately, proprietary, and seems to be aimed at businesses rather than education. Our language and IDE will have an edge since their implementations will be free and open-source, and will be more suitable for teaching beginners, having unlockable language features and good error messages.

4. Conclusions

People are beginning to realise that programming is a fundamental skill in our increasingly computerised society. Not just for software engineers and computer scientists, but for everyone. However, the present languages and tools used to teach programming have several problems. They often have error messages that may be useful for advanced users, but utterly perplex beginners with concepts that are beyond them yet, and they suffer from having to write "magic incantations". The tools sometimes have little or bad support on some platforms, creating a barrier to entry for inexperienced students.

We propose to solve these issues by making a programming language, based on Haskell, for teaching, with unlockable levels of features. This solves the first two described problems. To make the barrier to entry as low as possible we propose the engineering of an online IDE where users may write and run code in the browser.

This IDE also opens up many other exciting possibilities, such as collaboration, sharing, virtual classrooms with integrated exercises, and game programming.

The existing tools we have analysed have unfortunate shortcomings. Our solution on the other hand is well cool.

References

[1] Simon Marlow et al. Haskell 2010 language report. 2010.

[2] Edsger W. Dijkstra. To the members of the Budget Council. 2001.

[3] Time. Fifty years of basic, the programming language that made computers personal. 2014.

[4] Marco Cantu. *Essential Pascal*. CreateSpace, 2008.

[5] Brian W Kernighan. *Why Pascal is not my favorite programming language*. Bell Laboratories, 1981.

[6] Gerald Jay Sussman and Guy L Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(04):365–378, 2004.

[8] Logo Foundation. What is Logo?, 2011.

[9] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*, pages 62–71. ACM, 2003.

[10] Carnegie Mellon University. What is Alice?, 2015.

[11] Peter Van-Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[12] Matthew Flatt and PLT. The Racket Reference v.6.1.1, 2015.