Forty hours of declarative programming
Teaching Prolog at the Junior College Utrecht

Jurriën Stutterheim Wouter Swierstra Doaitse Swierstra
Department of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{j.stutterheim,w.s.swierstra,doaitse}@uu.nl

This paper documents our experience using declarative languages to give secondary school students a first taste of Computer Science. The course aims to teach students a bit about programming in Prolog, but also exposes them to important Computer Science concepts, such as unification or depth-first search. Using Haskell's Snap Framework in combination with our own `NanoProlog` library, we have developed a web application to teach this course.

1 Introduction

The Junior College Utrecht (JCU) is a joint initiative of Utrecht University and 26 secondary schools. The JCU offers talented students short courses at several different departments of the Faculty of Science of Utrecht University. In their penultimate year of secondary school, students need to do a small research project that encompasses approximately 40 hours of work, including the preparation of a final report and a presentation. In the past two years, we have developed a Computer Science course for these students [18].

Choosing suitable material for such a course is not easy. Part of the purpose of the JCU is to ignite the students' interest in (Computer) Science. While we want the students to learn a bit of computer programming, the course should strive to teach more. We would like our students to learn about something about algorithms, semantics, and other abstract concepts from Computer Science. Upon completing the JCU Computer Science course, students should have a better idea of what a degree in Computer Science entails—and hopefully, choose to enroll at Utrecht University.

We needed to decide what language to teach these students. Should we teach a mainstream programming language, such as Java or C#? Or should we teach a 'toy' language such as Alice [2] or Scratch [12]? Or should we teach languages popular in the functional programming language research community such as Agda [14], Epigram [13], OCaml [10], Racket [8], or Haskell [15]? In the end, we opted for none of the above.

The following requirements helped us to determine our choice of language:

- The language needs to be *simple*. Students should be up and running as quickly as possible. As we only have approximately two and a half days to teach the students, we want a language where with minimal opportunity for syntax errors, type errors or incomprehensible compiler messages.
- The language needs to be non-trivial. Many introductory programming languages target a younger, pre-teen audience by offering on drag & drop IDEs and focussing on drawing and/or animating images. The language should be challenging enough to intrigue clever, seventeen year-old secondary school students. A language like Scratch, for example, is great for simple animations and games, but it does not illustrate what real Computer Science is about and how real-world problems can be solved by programming.

Submitted to:
TFPIE 2012

In the end, we settled upon (a fragment of) Prolog. This may seem like a peculiar choice, so we would like to provide some motivation for our choice:

- The students have no trouble learning the syntax. Typically, they are writing their first Prolog relations within hours. Had we tried to teach Java, for example, we would need a significant amount of time to explain what syntax like `public static void` means and how to deal with curly braces and semi-colons. Choosing Prolog avoids this problem altogether.
- They can solve non-trivial problems by writing out a precise, mathematical specification. In our course, for instance, they develop a small Sudoku solver. The research topics we suggest cover a wide variety of problem domains, including scheduling problems, routing problems, basic genetics, and logical brainteasers. In each of these domains, they learn how to formalize some abstract concept, resulting in an executable program.
- Using Prolog gives us the opportunity to teach the students about backtracking, unification, proof trees, and many other important concepts. We hope that, by doing so, they learn there is more to Computer Science than just programming.
- We can teach them how an interpreter works. We have implemented a minimal interpreter for Prolog in Haskell, and use this to teach our students a bit about functional programming. Not all students study this material, but the students with previous programming experience find this by far the most interesting part of the course.

We have developed a web application to support this course. The application consists of two modes of operation: a web-based interface to our Prolog interpreter and an interactive Prolog proof assistant. This application, including its browser-based front-end, was built entirely in Haskell using the Snap web framework and the Utrecht Haskell Compiler (UHC). In the following sections we will discuss the implementation of our Prolog interpreter (Section 2), the interactive proof assistant (Section 3), and the students' experience working with these tools (Section 4).

2 NanoProlog Interpreter

The core of our interpreter is 150 lines of Haskell, excluding the parser and main I/O loop. This section aims to give you an idea of how our NanoProlog interpreter works. The complete source code is available from Hackage as the `NanoProlog` package. We hope that it is simple enough that motivated students, with our help, will be able to understand how this interpreter implements Prolog's backtracking search.

The central data types we use to represent Prolog programs are defined as follows:

```
data Term
  = Var String
  | Fun String [Term]
data Rule = Term ⊢ [Term]
type Program = [Rule]
```

A Prolog *Term* is either a variable or a function constant applied to a list of terms. To simplify parsing, we require all variables to start with a capital letter; all constants should start with a lower-case letter. A *Rule* describes a single Prolog inference rule. The rule $t \vdash ts$ states that to prove the goal t, it is sufficient to prove each sub-goal in the list ts. For example, a Prolog rule such as

$ancestor\ (X,Y) \mathrel{:\text{-}} parent\ (Z,Y), ancestor\ (X,Z)$

can be represented as:

```
(Fun "ancestor" [Var "X", Var "Y"]) ⊢
  [Fun "parent" [Var "Z", Var "Y"]
  ,Fun "ancestor" [Var "Z", Var "Y"]]
```

Finally, a program consists of a list of such rules.

The Prolog interpreter takes a program as input, together with a goal query. It constructs an inhabitant of the following *Result* type:

```
newtype Env = Env {fromEnv :: Map String Term}
data Result
  = Done Env
  | Apply [Result]
```

The answers we are looking for are substitutions, mapping variables to Prolog terms. We represent such substitutions as environments, using Haskell's *Data.Map* library. If the answer to our query is simple enough, there might not be any variables and the environment may be empty. The *Result* data type has two constructors: the *Done* constructor returns the required solution; the *Apply* branches over all possible rules that can be applied to solve the current goal.

The resolution process now proceeds in two steps: we begin by constructing a *Result* data type, representing the search tree leading to all possible answers. By traversing this tree in any order, we can search for answers.

The *solve* function below forms the heart of our interpreter. The function unifies the current goal with the conclusion of every possible rule. When unification succeeds, we proceed by resolving the right-hand side of the rule in depth-first fashion.

```
type Goal = Term
solve :: Program → Goal → Result
solve rules goal = steps (Just (Env empty)) [goal]
  where
    steps :: Maybe Env → [Goal] → Result
    steps Nothing _       = Apply [ ]
    steps (Just e) [ ]    = Done e
    steps e        (g : gs) =
      Apply [steps (unify (g, c) e) (cs ++ gs) | c ⊢ cs ← rules]
```

The actual implementation also tracks which rules are applied at every step, so that we can not only return the successful substitutions, but also the trace of all the rules that were applied to solve the initial goal.

The *unify* code is shown below. When the function is applied to a two terms and a non-empty environment, the two terms are unified after the variables in the terms have been substituted by values from the environment. If one of the two terms is a variable, a substitution from the variable to the other term is inserted in the environment. If both terms are rules, the right-hand sides of the rules are unified if and only if both rules have the same name and the same number of terms on the right-hand side.

```
unify :: (Term, Term) → Maybe Env → Maybe Env
unify _      Nothing                  = Nothing
unify (t, u) env@(Just e@(Env m)) = uni (subst e t) (subst e u)
  where
    uni (Var x)      y         = Just (Env (insert x y m))
    uni x            (Var y) = Just (Env (insert y x m))
    uni (Fun x xs) (Fun y ys)
      | x ≡ y ∧ length xs ≡ length ys = foldr unify env (zip xs ys)
    uni _            _         = Nothing
```

The *Subst* typeclass is used to make the substitutions before unification.

```
class Subst t where
  subst :: Env → t → t
instance Subst a ⇒ Subst [a] where
  subst env = map (subst env)
instance Subst Term where
  subst env (Var x)      = maybe (Var x) (subst env) (lookup x (fromEnv env))
  subst env (Fun x cs) = Fun x (subst env cs)
instance Subst Rule where
  subst env (c ⊢ cs) = subst env c ⊢ subst env cs
```

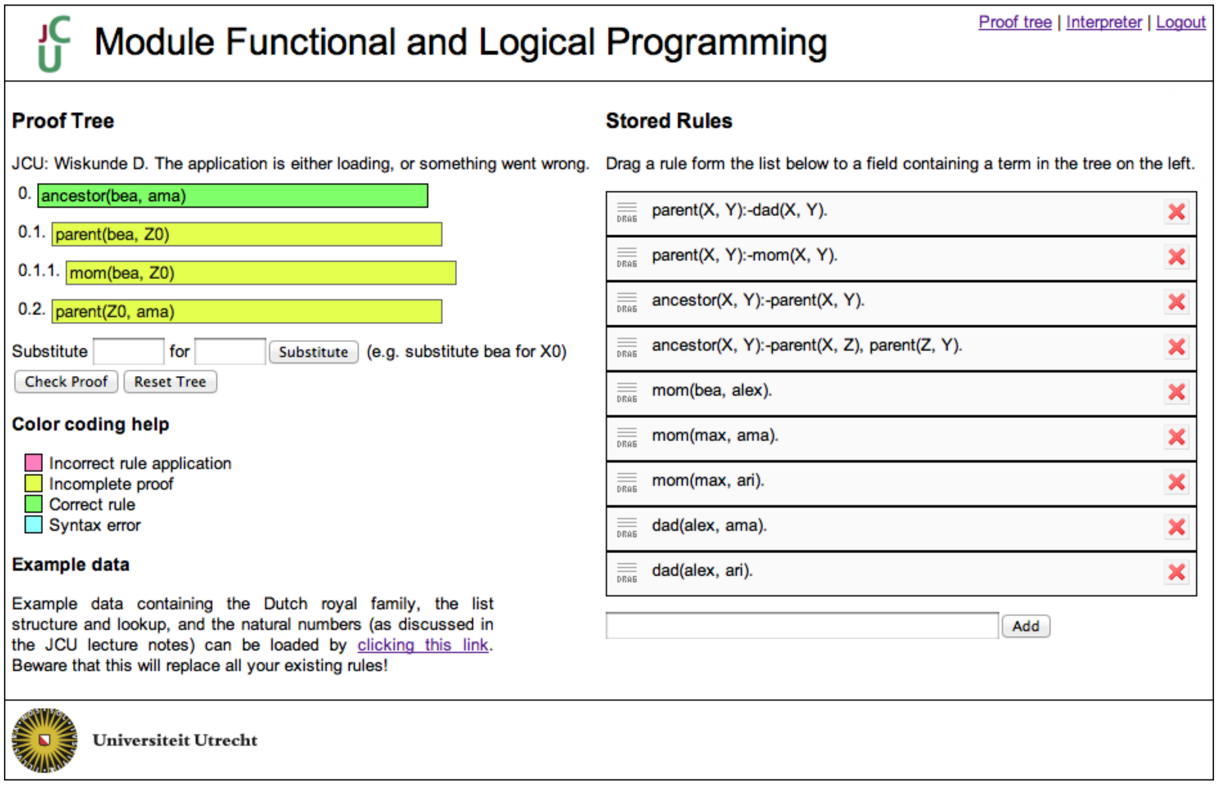
Our NanoProlog interpreter does have several restrictions. There is no *cut* operator; there is no way to define negation; there are no built-in integers, strings, lists, IO, or other functionality. To keep the implementation minimal, there is no occurs check. Yet the resulting, simple language is just enough to solve simple problems.

3 Web Application

To allow the students to experiment with Prolog, without having to install any software themselves, we have developed a web application. The application has two modes of operation: an interactive 'theorem prover' and a Prolog interpreter.

In the first mode, we aim to teach the students how unification and backtracking work. Students can enter a Prolog query and then try to 'prove' it themselves by dragging and dropping rules from a list onto the query, thereby constructing a proof tree. Dropping a rule on a term unifies the conclusion of the rule with that term. The body of the rule may then introduce new sub-goals, expanding the proof tree. This is repeated until all the proof tree's leaves are basic Prolog facts, and the proof is complete.

A student can ask for feedback while working on the proof. When a branch of the tree still has open sub-goals, its leaves are rendered with a yellow background, indicating that there is still some work to be done. Nodes that are have been successfully completed turn green. When all nodes have turned green, the proof is complete and the student is congratulated with a message box. Figure 1 on the facing page shows an incomplete proof. The root node has successfully been proved, but the remaining nodes still contain open goals. By dragging and dropping the rules, listed on the right, on the open sub-goals the students can complete the proof.



Module Functional and Logical Programming

Proof Tree

JCU: Wiskunde D. The application is either loading, or something went wrong.

0. ancestor(bea, ama)
0.1. parent(bea, Z0)
0.1.1. mom(bea, Z0)
0.2. parent(Z0, ama)

Substitute for (e.g. substitute bea for X0)

Color coding help

🟥 Incorrect rule application
🟨 Incomplete proof
🟩 Correct rule
🟦 Syntax error

Example data

Example data containing the Dutch royal family, the list structure and lookup, and the natural numbers (as discussed in the JCU lecture notes) can be loaded by clicking this link. Beware that this will replace all your existing rules!

Stored Rules

Drag a rule form the list below to a field containing a term in the tree on the left.

```
parent(X, Y):-dad(X, Y).
parent(X, Y):-mom(X, Y).
ancestor(X, Y):-parent(X, Y).
ancestor(X, Y):-parent(X, Z), parent(Z, Y).
mom(bea, alex).
mom(max, ama).
mom(max, ari).
dad(alex, ama).
dad(alex, ari).
```

Figure 1: An incomplete proof

Variables can be substituted in two ways. When a rule's conclusion is unified with an open subgoal, a substitution is produced and applied to the entire tree. Alternatively, the students can manually apply a substitution by using the controls just below the proof tree.

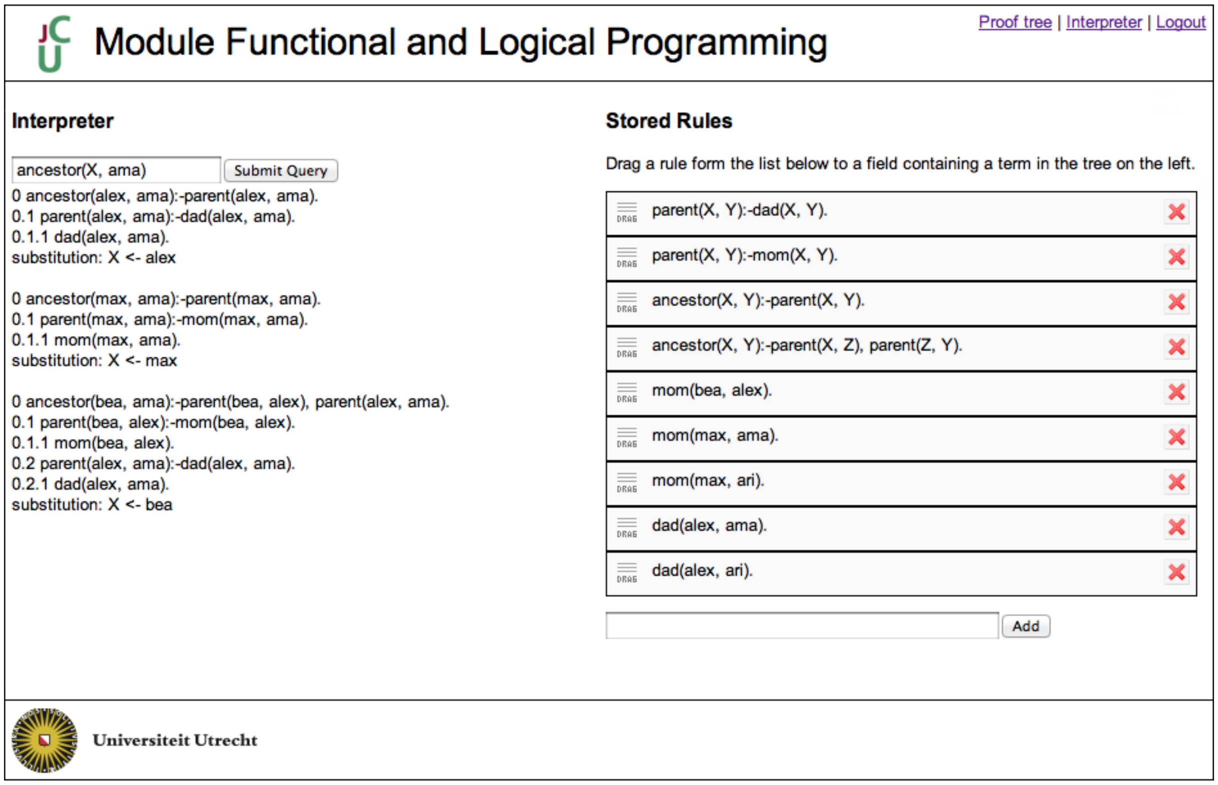
The second way the students can use the application is by using it as an interpreter. Figure 2 on the next page illustrates this mode of development. Students are presented with a single text field in which they can enter a Prolog query. If the query can be proven, using the rules on the right, the interpreter will print the corresponding proof trees and substitutions on the screen. Students can define new rules, using the text field at the bottom-right.

Before being able to use the application, students need to register an account. This enables us to store a set of rules for each individual student, giving them their own playground to experiment with Prolog.

Implementation

We have implemented both the client and the server part of the application entirely in Haskell. On the server side, we use the Snap Framework [1] to expose a RESTful [7] API to the NanoProlog interpreter and a database in which we store the Prolog rules the user has defined.

On the client side, we make use of the Utrecht Haskell Compiler's (UHC) [5, 4] JavaScript backend [6]. This enables us to write Haskell code and cross-compile it to JavaScript, which is run in the students' browser. Earlier revisions of the software were implemented in plain JavaScript and delegated parsing terms, unification and proof verification to the NanoProlog library on the server. By compiling the NanoProlog library to JavaScript, these operations can be executed client-side, eliminating additional


Module Functional and Logical Programming
Proof tree | Interpreter | Logout
Submit Query
Add
Universiteit Utrecht

Interpreter

ancestor(X, ama)

```
0 ancestor(alex, ama):-parent(alex, ama).
0.1 parent(alex, ama):-dad(alex, ama).
0.1.1 dad(alex, ama).
substitution: X <- alex

0 ancestor(max, ama):-parent(max, ama).
0.1 parent(max, ama):-mom(max, ama).
0.1.1 mom(max, ama).
substitution: X <- max

0 ancestor(bea, ama):-parent(bea, alex), parent(alex, ama).
0.1 parent(bea, alex):-mom(bea, alex).
0.1.1 mom(bea, alex).
0.2 parent(alex, ama):-dad(alex, ama).
0.2.1 dad(alex, ama).
substitution: X <- bea
```

Stored Rules

Drag a rule form the list below to a field containing a term in the tree on the left.

```
parent(X, Y):-dad(X, Y).
parent(X, Y):-mom(X, Y).
ancestor(X, Y):-parent(X, Y).
ancestor(X, Y):-parent(X, Z), parent(Z, Y).
mom(bea, alex).
mom(max, ama).
mom(max, ari).
dad(alex, ama).
dad(alex, ari).
```

Figure 2: A proof found by the web-based interpreter

AJAX requests. Unfortunately, the JavaScript code generated by the UHC is currently not fast enough to implement the complete interpreter on the client-side, hence we keep it on the server.

4 Discussion

Results

In practice, we only have about two and a half days to teach the students. The remaining time is reserved for the preparation of a presentation, the writing of a final report, and a mini-symposium where all students across the different Sciences present their results.

The first year we ran this course, it was quite ‘free-form’, in part because we had not assembled all the required material. Instead of spoon-feeding the students, we hoped that they would be capable of some individual exploration: the JCU specifically aims to teach students how scientific research works. In practice, however, these students, regardless of how motivated they are, are too young to work independently.

The second time we ran the course, we provided the students with a set of course notes, containing numerous small exercises. This gave the students a clear initial goal: read the course notes, complete the exercises, and learn to use the web application. As they were working on this, we provided them with a list of ‘research’ projects. When they were comfortable using Prolog and our web application, they could start working on one of these projects. This gave the students the freedom to work on a small research project, but still provides enough guidance.

By working through the course notes, the students learn some important concepts from Computer Science. The course notes start with a short introduction on substitutions, equations, and variables in terms of the high-school mathematics with which the students are already familiar. Next, they learn a bit about Prolog through a series of examples, starting with a set of relations to describe a family tree. We introduce *recursion* very early on, starting with a definition of an *ancestors* relation. We then move on to simple Peano arithmetic, list processing, and our largest example, a small Sudoku solver. At this point, the course notes shift towards describing how Prolog resolution works. At first, we present a pseudo-code algorithm, exposing the students to concepts such as *unification*, *search trees*, and *depth-first search*. The final chapter of the notes explains how this pseudo-code can be implemented in Haskell. Upon completing the course notes, the students have been exposed to several fundamental Computer Science concepts.

The research projects that the students tackle in the last day or so takes the form of a more open ended challenge. Some of these problems are inspired by Sudoku-like puzzles, such as Kakuro. Others build on the family tree example, asking them to define new family relations or to extend the relations with some basic genetics (e.g., if both your parents exhibit the same recessive trait, you share that trait). Furthermore, we offer them a choice of several other Prolog programming task, such as defining a toy route planner system or simple scheduling software. Finally, we suggest several projects that modify the interpreter, such as implementing a breadth-first search or adding a cut operator. These last exercises tend to be quite daunting. Few students complete their project, but all students do manage to achieve some basic results.

Student Reflection

Throughout the course, the students were under constant supervision, which allowed them to frequently ask questions. While working through the course notes, the concept that the students seem to struggle with most was the scope of variables. The students found it hard to keep in mind that a variable X in one rule is not the same variable X in another rule, especially in the case of recursion. Luckily, by the time they moved on to their projects, this confusion had nearly completely disappeared.

At the end of their research project, all of the students are required to reflect on their research by writing a short report. We summarize the experiences of the spring 2011 class here, which consisted of eleven students. Some of them had some prior programming experience, but most did not.

Especially the students without programming experience indicated that they did not really know what Computer Science was or how difficult the course was going to be. Some even indicated that this was an important reason for choosing to take part in this project. Despite the lack of prior experience, nearly all of the students enjoyed the course, even if they found it to be quite challenging. Many students would have liked to have had more time. They would have liked to get a better understanding of what they were doing and to be able to do more programming.

Even though the web application and interpreter were available for exploring Prolog, the students indicated that they were glad that they had the opportunity to ask questions in class.

The final reports of the students of the 2011–2012 class have not been submitted yet. We hope to include a short discussion about them in the final version of this paper.

Related Work

There are several similar initiatives aimed at teaching programming to secondary school students. Bootstrap [16] is a curriculum aimed at middle-school children of ages 11 to 14. It focuses on teaching kids how to program video games using algebraic and geometric concepts that they know from their mathematics classes. As such, the curriculum is designed to work in conjunction with the regular middle-school program. It consists of nine units and some supplemental lessons. Students work with the functional programming languages Racket and Scheme. It also offers a web-based environment in which the children can write and execute code and see the result appear in the web browser.

Haskell for Kids [17] targets a similar audience as Bootstrap, namely children around the age of 12 or 13. It teaches the basics of Haskell programming by letting the children draw images and even animations using the Gloss library [11]. Similarly to bootstrap, Haskell for Kids offers a web-based development interface.

Khan Academy [9] offers a collection of introductory video lectures on Python. Exercises are not included. The target audience is not specified, but the level of the videos suggests that it is aimed at teenagers and above.

Alice [2] takes a more graphical approach to teaching programming. It focusses on visual learners by enabling them to drag and drop objects into a program to create an animation or a game. As opposed to the previous works, it uses a Java-like language, wrapped in a graphical user interface of a custom IDE. It is targeted at post-secondary education students.

Scratch [12] provides a drag and drop interface to programming in a custom IDE. The language itself is imperative, with an event-driven concurrency model. The language has a simple, but intuitive type system that shows types as different shapes. Scratch is aimed at children between 8 and 16 years of age.

Microsoft have developed Small Basic [3] to encourage people to learn how to program. Small Basic is a simplified Basic dialect. The IDE is simple and intuitive, and focuses on providing information about the Small Basic libraries API. It has a similar target audience to Khan Academy.

Conclusion

The module we teach is aimed at secondary school students. Besides teaching them a bit about computer programming, we aim to teach them about *Computer Science*. We expose them to concepts such as unification and substitution; explain search trees and search algorithms; and teach them to solve problems using recursion. We feel that this really distinguishes our course from many of the existing approaches already available.

References

[1] Gregory Collins, Doug Beardsley, Shu-yu Guo, James Sanders, Carl Howells, Shane O'Brien, Ozgun Ataman, Chris Smith & Jurriën Stutterheim: *Snap Framework*. Available at `http://snapframework.com`.

[2] Matthew J. Conway (1997): *Alice: Easy-to-Learn 3D Scripting for Novices*. Ph.D. thesis, University of Virginia.

[3] Microsoft Corporation: *Microsoft Small Basic: An introduction to programming*. Tutorial available online.

[4] Atze Dijkstra, Jeroen Fokker & S. Doaitse Swierstra (2009): *The Architecture of the Utrecht Haskell Compiler*. In: *Haskell Symposium*.

[5] Atze Dijkstra, Jeroen Fokker & S. Doaitse Swierstra (2009): *UHC Utrecht Haskell Compiler*. Available at `http://www.cs.uu.nl/wiki/UHC`.

[6] Atze Dijkstra, Jurriën Stutterheim, Alessandro Vermeulen & S. Doaitse Swierstra: *Building JavaScript applications with Haskel (Experience Report)*. Submitted to ICFP 2012.

[7] Roy Thomas Fielding (2000): *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine.

[8] M. Flatt (2010): *PLT. Reference: Racket*. Technical Report, Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang. org/tr1.

[9] Sal Khan: *Khan Academy*. Available at `http://www.khanacademy.org/#computer-science`.

[10] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy & Jérôme Vouillon (2011): *The OCaml system release 3.12: Documentation and user's manual*. Technical Report, Institut National de Recherche en Informatique et en Automatique.

[11] Ben Lippmeier (2010): *Gloss*. Available at `http://gloss.ouroborus.net/`.

[12] J Maloney, L Burd, Y Kafai, N Rusk, B Silverman & M Resnick (2004): *Scratch: A Sneak Preview*. In: *Second International Conference on Creating, Connecting, and Collaborating through Computing*, pp. 104–109.

[13] Conor McBride & James McKinna (2004): *The view from the left*. *Journal of Functional Programming* 14(1), pp. 69–111, doi:10.1017/S0956796803004829.

[14] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

[15] Simon Peyton Jones, editor (2003): *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

[16] Emmanuel Schanzer: *Bootstrap*. Available at `http://www.bootstrapworld.org/`.

[17] Chris Smith (2011): *Haskell for Kids*. Available at `http://cdsmith.wordpress.com/category/haskell-for-kids/`.

[18] Wouter Swierstra, S. Doaitse Swierstra & Jurriën Stutterheim (2011): *Logisch en Functioneel Programmeren voor Wiskunde D*. Technical Report UU-CS-2011-033, Universiteit Utrecht.