Haskell in Middle and High School Mathematics

Fernando Alegre
Cain Center for Scientific, Technological, Engineering and Mathematical Literacy
Louisiana State University
Baton Rouge, Louisiana, USA
falegre@lsu.edu

Juana Moreno
Dept. Physics & Astronomy and Center for Computation and Technology
Louisiana State University
Baton Rouge, Louisiana, USA
moreno@lsu.edu

This paper describes an ongoing project to create programming activities that can be incorporated into the high school mathematics curriculum so that eventually they become part of the core instruction and not just supplementary activities. While our project is in its preliminary stages, we have developed some material illustrating our approach, which can be described as minimalist. We want to focus on those parts of programming that are directly related to mathematics lessons and ignore as much as we can about any other aspect of programming, and in particular about interfaces and input-output operations. We have run a summer camp and several saturday workshops for middle and high school students to teach them basic concepts of programming and the connections between programming and mathematics. We chose to use Haskell for all the instruction due to its similarity to mathematical language, and our pilot tests showed that students with no previous programming experience were, after a short time, able to understand and modify the Haskell scripts we gave them. In fact, most of the difficulty they had was due to poor understanding of the underlying mathematics, and the programming exercises helped them recognize and, for some students, correct their deficiencies. In partnership with our local public school district, we are preparing a schoolwide deployment of the activities in a magnet high school. The activities will be embedded in the regular Algebra I and Geometry courses, and if the project is successful, it will be later extended to other high schools in the district.

1 Introduction

We are trying to address two problems in the curriculum of most high schools in the US.[1] First, most mathematics courses do not use any programming technology beyond graphing calculators, which offer a limited programming interface in a custom version of BASIC that did not substantially change in the last 25 years. Second, most schools offer a very limited selection, if any, of computer science courses, and those courses are usually optional and have low enrollment. In consequence, very few high school students are exposed to modern high-quality programming languages and tools.

Current efforts to change the high school curriculum to introduce new computer science courses are not likely to change the underlying problem of the current situation, which is that most students can graduate high school without having been exposed to any programming experience[60, 83]. Even programs such as CS10K, a massive program sponsored by the US National Science Foundation, which

[1] While we are familiar only with the US educational system, we expect similar situations to occur in many other countries.

Submitted to:
TFPIE 2015

© F. Alegre, J. Moreno
This work is licensed under the
Creative Commons Attribution License.

aims to train ten thousand teachers to teach new computer science courses, will reach at most half the high schools in the US, and it will still provide programming opportunities in optional, non-core courses.

We are exploring an alternative approach to tackle the problem based on embedding programming instruction in existing compulsory core subjects. While this approach will guarantee that all students are exposed to some programming experience during their high school years, it presents some new challenges of its own. First, the programming exercises must be relevant to the subject in which they are embedded, as teachers in core subjects do no have much discrectionary time to use for supplementary activities. Second, the depth of the programming instruction must be suitable for a teacher who may have never taken any computer science course and who must learn the material in a few professional development sessions. Third, the programming environment must have low threshold and high ceiling, so that simple problems can be easily solved but there is no limit to what a motivated learner or a knowledgeable teacher can do with it.

Given the previous constraints, we found it natural to choose mathematics as the core subject and Haskell as the programming language. We also had to keep our goals modest: provide a first exposure to programming to students so that they decide whether they may want to explore the subject further, ensure that students are not scared of reading code, get students to make small changes to existing code and encourage teachers to use code snippets in their regular instruction. In order to keep the programming as simple as possible, we chose to use only a subset of Haskell without monads, higher-level categorical interfaces or IO. In fact, while the learning curve for the advanced features of Haskell is allegedly very steep, the beginning of the curve is flatter than in most mainstream programming languages. Introducing basic programming concepts in Haskell requires almost no instruction time due to the strong similarity of the concepts and notation to the algebraic language used in mathematics, and the following informal rules cover basically all the syntactic differences between Algebra and Haskell:

1. In Algebra, functions have parameters enclosed in parentheses and separated by commas, while in Haskell, parameters follow the function name separated by spaces with no enclosing symbols.

2. In Haskell, auxiliary definitions that are local to a particular function definition are introduced by the keyword `where` and must form a left-aligned block that is indented to the right with respect to the function definition, while in Algebra, there is no convention for denotating auxiliary definitions.

3. In Haskell, newly introduced symbols must appear on the left-hand side of an equation, while in Algebra, they may appear on either side.

4. In Algebra, sets are used for two different purposes: to denote the range of possible values for a symbolic expression and to group several values together as a single entitiy. In Haskell, the former is done with type annotations while the latter is done with lists. However, sets do no specify the order of their elements, and duplicate elements are counted only once in sets. On the other hand, lists maintain the specified order for their elements, and duplicates are kept in lists as many times as they are listed.

The rest of the paper is organized as follows. Section 2 gives a summary of the pilot tests we ran and an account of the use of Haskell in the tests. Sections 3, 4 and 5 go into more detail about how arithmetic, algebra and geometry were used in the pilot tests. Section 6 gives a historical perspective of similar projects. Finally, section 7 provides concluding remarks and our plans for the future.

2 Pilot tests

We have run a summer camp and several saturday workshops for middle and high school students to teach them basic concepts of programming and the connections between programming and mathematics. Our pilot tests showed that students with no previous programming experience were, after a short time, able to understand and modify the Haskell scripts we gave them. In fact, most of the difficulty they had was due to poor understanding of the underlying mathematics, and the programming exercises helped them recognize and, for some students, correct their deficiencies.

Haskell has a reputation of being a difficult language. Part of the difficulty of Haskell arises from the somehow rigid syntactic rules regarding layout. In our case, students frequently made trivial syntax mistakes, but they were able to correct them when told to carefully search for typos and improper indentation. While having to pay attention to such errors was a nuisance for both students and instructors, the total time lost due to these issues was insignificant compared to the amount of time the students needed to come up with correct solutions to the problems we posed them.

Other sources of difficulty in Haskell are the constraints on the input-output interface to make it logically consistent with the rest of the language and the very-high-level categorical constructs that make it terse and extremely powerful. However, in introductory programming activities such as ours, it is easy to just avoid both. Our interface was restricted to interactive use of the GHCI interpreter and graphics based on the Gloss library, which abstracts away all interface details and presents a purely functional interface. We did not make any use of Applicative or Monad, and in fact we did not introduce the `do` notation, and we introduced the composition (`.`) and application (`$`) operators as *shortcuts to reduce the number of parentheses* without delving into the subtleties of operator precedences or the mechanics of currying and its relation to evaluating multiple arguments. This approach to teaching Haskell did not actually produce any confusion or dissonance on the students, who seemed to accept very naturally the role of composition and application as shortcuts, which in a sense is just what they are.

Even without advanced features, Haskell had a lot to offer. Particularly useful were its referential integrity, essential for performing equational reasoning, and its lazy evaluation model, which facilitates the manipulation of infinite objects. While the former is shared with many functional programming languages, the latter is more particular to Haskell, which is probably the only lazy language that has a non-negligible active community behind it. In addition, the fact that the syntax for assignments closely resembles the usual mathematical syntax was essential for a seamless transition between programming and mathematics that was repeatedly used in all exercises. Other advantages of Haskell that were less explicitely exploited in our tests but nevertheless played an important role are the strong and rich type system, always enforcing correctness in the background, and the clear separation between interface programming and algorithmic programming, which helped us cleanly delimit which features were allowed in the programs.

We organized the summer camps before we had an agreement with the school district, and at that time we were planning on targetting mainly middle school students. However, our pilot tests convinced us that many middle school students lack the necessary maturity to handle the abstract constructs normally used in algorithmic programming. Reported success of graphical programming environments such as Scratch[3] among middle schoolers may be due to the lower abstraction requirements of the effect-oriented programming encouraged by those environments rather than to the graphical appeal of the interface. This premise was reinforced by the results of our tests with high school students, who felt comfortable using just an editor to handle the programs and did not seem to miss a graphical interface.

We will now show three examples of the type of activities we proposed to the students. The first two examples were done in sessions for middle school students, and the last example was done by high school

students. These activities focus on areas where the students had either deficiencies in their mathematical understanding or enough basic understanding to be exposed to enriching instruction, and our goal was to find whether the deficiencies can be corrected (or the enrichment can be carried out) by having the students program the underlying mathematics.

3 Arithmetics, Induction and Recursion

Our first pilot test targetted students entering eighth grade. We chose a subject that all of them were familiar with: arithmetic of natural numbers. We thought that illustrating how to implement the four rules in a computer could be a good introduction to mathematical induction and to recursion, its programming counterpart.

Pattern matching, incremental definition of functions and the identification of strings and lists were some of the useful features of Haskell that were exploited to implement the usual algorithm for addition using decimal digits, as shown in listing 1. Natural numbers were represented as strings of digits, and list operations were used to construct and deconstruct them. The students were taught that strings of digits can be manipulated with these functions: concatenation (`++`), list construction (using square brackets), prefix extraction (`init`) and suffix extraction (`last`). They were also told that string literals must be enclosed in double quotes and single digit (character) literals must be enclosed in single quotes. The definition of the functions `next` and `prev` were provided, and the syntax for multiple-case function definitions was explained. Then the students worked on exercises aimed to practice those constructs, and finally they were asked to write the definition of the `count` function and the addition algorithm.

Listing 1: Algorithm for addition using decimal digits.

```
module NoMath where
import Prelude hiding ((+),(-),(*),(/),(^),sqrt,(<),(<=),(>),(>=))

next "0" = "1"; next "1" = "2"; next "2" = "3"; next "3" = "4"
next "4" = "5"; next "5" = "6"; next "6" = "7"; next "7" = "8"
next "8" = "9"; next "9" = "10"

next x | last x == '9' = next (init x) ++ "0"
       | otherwise = init x ++ next ([last x])

prev "10" = "9"; prev "9" = "8"; prev "8" = "7"; prev "7" = "6"
prev "6" = "5";  prev "5" = "4"; prev "4" = "3"; prev "3" = "2"
prev "2" = "1"; prev "1" = "0"

prev x | last x == '0' = prev (init x) ++ "9"
       | otherwise = init x ++ prev ([last x])

count initial final
    | initial == final = initial
    | otherwise = initial ++","++ count (next initial) final

"0" + x = x; x + "0" = x; x + "" = x; "" + x = x

x + y  | init x == "" = prev x + next y          -- x has a single digit
       | init y == "" = next x + prev y          -- y has a single digit
```

```
    | otherwise = (init x + init y + init p) ++ [last p]
  where p = [last x] + [last y]

-- Sample run:
-- *NoMath> count "73" "105"
-- "73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,
-- 95,96,97,98,99,100,101,102,103,104,105"
-- *NoMath> "929" + "741"
-- "1670"
```

It took the students about four hours and a half (divided into three sessions of one hour and a half each) to complete this exercise, and considerable guidance was provided during the sessions, but in the end, many eighth graders were able to write their own addition function, and even those who did not come up with their own were able to understand and use what their peers wrote. This was a difficult exercise, and the students were at times disoriented, given that they were asked for some intellectual effort very different from what they are usually accustomed to do in regular mathematics classes. After they completed the exercise, the students were enthusiastic about trying out the functions they wrote. We were surprised by how delighted they were to add and count up to ridiculously large numbers, competing among themselves to see who got the largest number on their screens. It seems that they had not realized before that computers can calculate with numbers in a way that surpases the abilities of any human, probably because modern uses of computers involve advanced graphics manipulations that require mathematical knowledge beyond the level of an eighth grader. On the other hand, experiencing this phenomenon with mathematics that they understand, such as counting and addition, made them aware in a visible way of the utility of computers as very powerful calculators.

Note how line 2 in listing 1 is used to prevent the students to use the built-in arithmetic operators. The students were told that this line *made the computer forget about arithmetics,* so their task was to teach the computer how to count and add numbers again. This pretend game was initially well accepted, but the students were still perplexed by the fact that they could not solve the counting problem just by repeatedly adding one to the input. It seemed that they did not understand the difference between working around a problem and solving it. This exercise may have been beyond the level of maturity of the typical eighth grader, and we expect that it will be more effective with older students.

4 Basic Algebraic Expressions

Our second pilot test also targetted middle school students and was about unit conversions.

Listing 2: Unit conversion

```
fl_oz = 1  -- This is our basic unit of volume measurement

-- These are derivative units:
gallon = 4 * quart
quart = 2 * pint
pint = 2 * cup
cup = 8 * fl_oz
tsp = 1/3 * tbsp
tbsp = 1/16 * cup

-- Problem 1: How many tsp are in 2 pints?
```

```
solution1 = (2 * pint) / tsp

-- Problem 2: In general, how many units are in some amount?
convert_to unit amount = amount/unit
```

As listing 2 illustrates, laziness can be exploited to present relationships between variables in a natural top-down order that follows logical dependencies instead of the bottom-up order required by eager evaluation. This example was the first one we gave to eighth grade students, who were not provided with the solutions to the problems. We guided the students to use equational reasoning repeatedly to arrive to the expected solution.

First, the students were asked to solve Problem 1 by hand. A typical solution went like this:

```
One pint is 2 cups.
So, 2 pints are 4 cups.
One cup is 16 tablespoons.
One tablespoon is 3 teaspoons.
So, one cup is 16*3=48 teaspoons
So, 4 cups are 4*48=192 teaspoons.
```

We then asked the students how did they know that one cup is 16 tablespoons, and more explicitely, we asked them to write an expression for the number of tablespoons in a cup, which can be obtained by dividing the volume filled by one cup by the volume filled by one tablespoon:

```
tbsp_in_a_cup = (1 * cup) / (1 * tbsp)         -- by definition
              = (1 * cup) / (1 * 1/16 * cup)   -- by substitution: tbsp = 1/16 * cup
              = 1 / (1 * 1/16)                 -- by cancelling out cup
              = 16                             -- by simplifying fractions
```

We then asked them to write similar expressions for all the derivations which they did by hand before:

```
-- One pint is 2 cups.
pint = 2 * cup              -- given
cup_in_a_pint = 2           -- since pint/cup = 2*cup/cup = 2
-- So, 2 pints are 4 cups.
two_pint = 2 * pint = 2 * 2 * cup = 4 * cup
cup_in_two_pint = 4         -- equivalent expression
-- One cup is 16 tablespoons.
tbsp_in_a_cup = 16          -- by above computation
-- One tablespoon is 3 teaspoons.
tsp_in_a_tbsp = tbsp / tsp = tbsp / (1/3) * tbsp = 3
-- So, one cup is 16*3=48 teaspoons
tsp_in_a_cup = tbsp_in_a_cup * tsp_in_a_tbsp = 16 * 3 = 48
-- So, 4 cups are 4*48=192 teaspoons.
tsp_in_two_pint = cup_in_two_pint * tsp_in_a_cup = 4 * 48 = 192
```

The point of the exercise was to show the students that equational reasoning works equally in Algebra and in Haskell. We also wanted to remark the emerging pattern:

```
number_of_'unit'_in_a_'amount' = amount / unit
```

That pattern allows us to write a shorter program to solve Problem 1:

```
tsp_in_two_pint = (2 * pint) / tsp
```

and let the computer do the work for us, which will internally go something like this:

```
tsp_in_two_pint = (2 * pint) / tsp = (2 * 2 * cup) / tsp = (4 * cup) / (1/3 * tbsp)
                = 12 * cup / tbsp = 12 * cup / (1/16 * cup) = 12 * 16 * cup / cup
                = 192
```

Note that the previous expansion may not be the exact expansion performed by the computer, but due to the referential integrity of Haskell, it does not matter, as the result will not depend on the order of expansions. In an imperative language, this type of reasoning would be misleading, as it does not resemble what the computer is actually doing and would not work in general. Executing a Haskell program is basically letting the computer do the equational substitutions that we would otherwise do by hand.

We can now convert the informal `in_a` pattern above into a legal Haskell expression by just using backticks:

```
unit `in_a` amount = amount / unit
```

which gives us the solution to Problem 2, except that the problem asked for a function named `convert_to`, which is straightforward: `convert_to = in_a`.

In total, students spent about 90 minutes with this example until they were able to come up with the solution to problem 2. The guidance provided made them understand better the role of reduction to normal forms in mathematical expressions and how it can be mechanized. It also emphasized the Church model of computation, which can be summarized by the slogan *computation is substitution.* Finally, it reinforced an abstract algebraic view of unit conversions as consisting of a single law (`convert_to`) that performs any conversion between two units no matter whether the intermediate units are defined forwards (`a = k*b`) or backwards (`b = k*a`).

5 Euclidean Geometry

The ability to solve problems while operating within the given constraints is essential for both programming and mathematics, but is made more vivid when programming is used to solve a mathematical problem. Both middle school students and high school students had difficulty understanding constraints, but this fact was particularly patent when we asked high school students to use dynamic geometry software [2] to perform simple geometric constructions using only the ruler and compass tools.

We originally intended to compare the understanding of high school geometry that can be achieved by the use of dynamic geometry software with what can be achieved by an approach based on programming turtle geometry[1]. While observing the students use the dynamic geometry software, we realized that even some students who had already taken Geometry were not achieving the goals we proposed. Our observation identified two main reasons. First, the interactive nature of the point and click interface was distracting, sometimes driving them away from the goal into more open exploration, and most of the time that exploration was fruitless, which led the students to get frustrated with the software. Second, their approach to problem solving was too shallow. Instead of trying to get the geometric constructions based on the constraints they need to satisfy, many students tried to come up with a construction that

[2] We used GeoGebra in this task, but we do not think the students behavior depended on the particular brand of software used.

looked right, but that was not based on constraints and thus did not last when the points were moved. For example, when we asked them to construct a perpendicular line to a given one without using the built-in perpendicular tool, instead of delimiting a segment and then constructing the bisector of that segment using intersection of circles, many students used a free line that cut the given line at an arbitrary point, and they kept adjusting the incidence angle between the two lines to make it, by trial and error, as close to 90 degrees as possible. Not only that, but some students had a hard time understanding why that approach was incorrect and kept insisting they had solved the posed problem.

Listing 3: Euclidean geometry functions

```
module Geometry where

type Number = Float
type Point = (Number,Number)

-- apart x y tells whether x and y are distinguishable from each other
apart :: Point -> Point -> Bool

-- Let x' be the projection of point x on the line ab. Then
-- beyond (a,b) x tells whether point b lies between a and x' or not.
beyond :: (Point,Point) -> Point -> Bool

-- Intersection between lines and circles
-- Lines are given by two points
-- Circles are given by the center and a point on the circumference
-- These functions may produce 0, 1 or 2 points in return

circle_circle :: (Point,Point) -> (Point,Point) -> [Point]
line_circle :: (Point,Point) -> (Point,Point) -> [Point]
line_line :: (Point,Point) -> (Point,Point) -> [Point]
```

Listing 4: Predicates implementing the betweenness relationship.

```
outside (a,b) x = beyond (a,b) x || beyond (b,a) x
between ab = not . outside ab
across x l = not . sameside x l

sameside x (y,y') x' | not (apart y y') = error "sameside"
                     | not (apart x x') = True
                     | otherwise = case line_line (x,x') (y,y') of
                                     [p] -> not (between (x,x') p)
                                     [] -> True
```

This experience, which contradicted previous research on dynamic geometry software[31, 32, 37], led us to change the design of our sessions. Instead of programming open-ended turtle geometry, we opted for implementing Euclidean geometry in a library. Our goal was to facilitate a programming model where it was just impossible to have constructions that look right but were incorrect. In the libraries we implemented, students have no control over the look, as the system chooses randomly the initial points on which the construction should be based. There are no instructions in our vocabulary to change the position of the points or the lines, and the only way to create new points and lines are as intersections of ruler and compass constructions based on the random input data. The restricted vocabulary helped

the students enormously to focus their effort on just finding the right constraint. Open exploration is not hindered in this approach[43], but it takes place in a more parsimonious way, as the students need to follow the typical program development cycle consisting of design, write, execute, test, debug and repeat phases. Students can no longer act as users of technology, but are set in the mindframe of producers of technology.

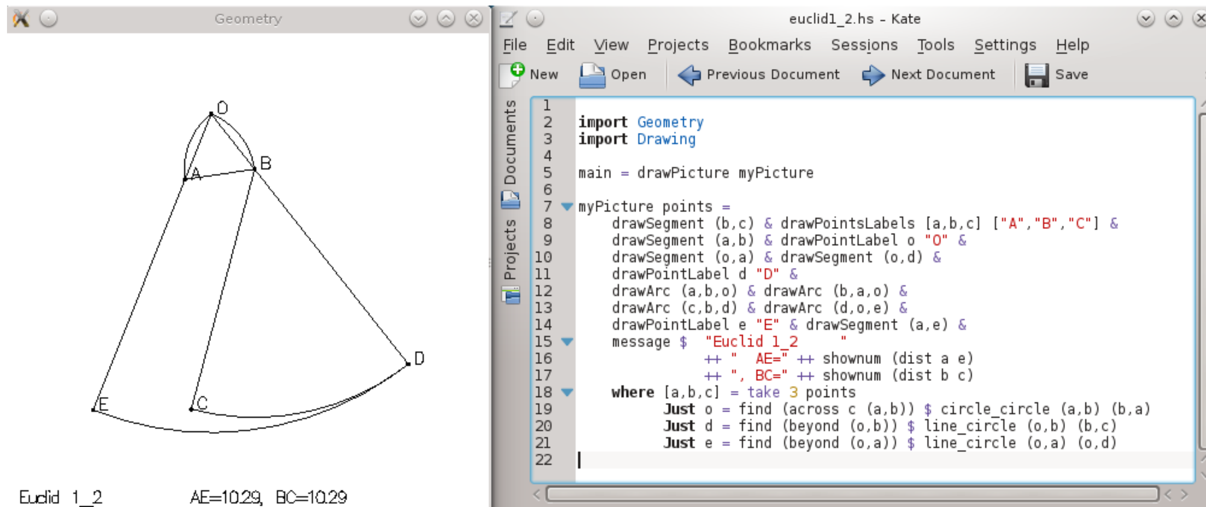

Figure 1: Script that implements the algorithm in Euclid's Elements Book I Proposition 2. Given a point A and a segment BC, construct a segment AE congruent to BC and anchored at A.

Listing 3 shows the basic interface we implemented. We used types minimally and opted not to use the type system to distinguish between lines and circles. We also followed constructive mathematics conventions and did not implement point equality. When two points are not apart, it may or may not be true that they are equal, and no assumptions should be made about that. The simplicity of the interface came at a cost, as students often confused the arguments and passed a circle when a line was expected or vice versa, so we are not entirely satisfied with that choice. We will probably introduce different types for lines and circles in our next version. We also implemented functions to draw points, lines, segments, circles, and to add labels to points. The interface for those is so straightforward that we will not spend more time on it. However, those functions were not enough to implement Euclidean geometry constructions, because intersections of lines and circles may contain two points, which are returned in no particular order. For certain constructions, we need to be able to choose one of them based on additional geometric constraints. A predicate that provides a way to order points was implemented in terms of the `beyond` function. Once this function is defined, it is easy to also have predicates for `outside`, `between`, `sameside` and `across`. Those are shown in listing 4.

Using this library, many problems were posed to students, such as equilateral triangles, squares, parallelograms, perpendiculars, hexagons, and barycenters, incenters and orthocenters of triangles. Surprisingly, the students, most of whom had already taken Geometry, struggled with the geometry, but once they understood what needed to be done had no trouble implementing it without help. They were encouraged to actually use the programming as a way to explore the geometry, but it took them a while to get accustomed to that idea. At the end of each saturday session, we had students who produced figures on their own, like saw-toothed polygons and octogonal stars, but the time allotted (about 4.5 hours each session) was not enough to proceed further. Nevertheless, the students seemed satisfied that they had

learned a lot of geometry in a relatively short amount of time.

Figure 1 shows an implementation of the algorithm I.2 from Euclid's Elements, which was given to the students to follow as an example. Lines 8-17 are just for drawing, and the actual computation is in lines 18-21. In line 18, the variable `points` holds an infinite supply of random points. Lazy evaluation makes this abstraction possible. Then, pattern matching is used to give the names `a`, `b` and `c` to the first three points in that list. In line 19, the intersection of two circles is computed, which produces two points. From these two points, point `o` is chosen in such a way that it lies across the segment (a,b) from point `c`. In line 20, the intersection of a line and a circle produces again two points, and point `d` is chosen as the point in line (o,b) that is not between `o` and `b`. Finally, in line 21, the point `d` is transported to line (o,a) by choosing again a point from the two points in the intersection of a circle and a line. Use of betweenness predicates is essential for this algorithm to be correct, and it is something that is implicit in Euclid's construction and easy to overlook when this algorithm is explained informally. The function `find`, part of the standard library of Haskell, takes a predicate and a list and returns `Just` the first point in the given list that satisfies the given predicate, or `Nothing` in case no such point exists. We would usually do case analysis to distinguish those two cases, but in this algorithm we are sure that `find` always returns a point.

6 Related Work

Papert[25] invented the programming language LOGO as a new tool for teaching mathematics and advocated the use of constructionist discovery[53] using the LOGO language as medium, claiming that transfer of creative thinking from computing to other areas would occur spontaneously. However, empirical studies by several independent researchers[55, 40] showed no evidence of such transfer. Butterfield[9] argues that analyses of the failure of LOGO did not separate the technology from the teaching methodology. Unguided discovery could have been the cause of failure, a point made by Mayer[43], who concludes that guided discovery is more effective than pure discovery.

On a somewhat related note, Shon[70] says that experience alone does not necessarily lead to learning; deliberate reflection on experience is essential. **Reflective practice** is used mainly in practice-based professional learning, such as nursing [44], teaching [41] or architecture, but it could also be applied to the practice of programming. Hazzan[29] proposes following a model for software engineering education similar to the studio model in architecture. Reflective practice is also important to avoid "production without comprehension" [55], i.e., "engage in haphazard hacking to seek out a solution" [72], a practice which is especially unproductive in scientific programming.

When an adequate teaching methodology was used, transfer between LOGO and mathematics was actually found. McCoy et al[45] point to increased students' understanding of geometry and variables, and similar results are cited by others with respect to spatial and and symbolic mental representations [33], a better understanding of elementary algebra [50], increased problem solving abilities [80], and better understanding of shape, measurement, symmetry, and arithmetic processes [12]. In addition to the teaching methodology, many other factors, such as the number of instruction hours, influence the impact of learning computer programming on learning mathematics [34, 94].

Rich[64] studied the conditions that could favorably influence the occurrence of transfer between two related disciplines, and coined the term **convergent cognition** to describe the mutual reinforcement that takes places when the disciplines are complementary along two dimensions: the declarative-procedural focus (learning of one subject should be mostly declarative while the other should be mostly procedural) and the abstract-applied scope (one subject should establish an abstract conceptual framework while the other should be centered on practical applications.) In this context, programming plays the role of the

procedural application of the abstract declarative framework learned in mathematics courses.

Palumbo et al[52] also point out that the choice of programming language may be an important factor determining the type and amount of transfer, and base their argument on studies where instead of LOGO, other languages such as BASIC or Pascal were used. McCoy et al[45] cite studies based on LOGO, BASIC, FORTRAN and Pascal where evidence for transfer was found, but with inconclusive results with respect to the choice of programming language used.

All the programming languages mentioned so far are procedural languages, which are based on sequences of commands that instruct the computer to modify the current state of memory, and dynamically replacing the values that variables represent with new computed values. A different paradigm, functional programming, uses functions as the basic building block and views a program as a large function built from composition of many smaller functions, where a function is a deterministic rule to transform input data into output data, just like a mathematical function does.

Recent studies using **functional programming** are promising. Schanzer et al[69] have used functional programming successfully for teaching algebra. Advanced variants of functional programming have also been successfully used in introductory programming courses at the undergraduate level [61]. Functional programming for undergraduates has also been used beyond computer science. Walck[86] created a curriculum for Computational Physics using Haskell as the medium, because "the structure of Newtonian mechanics is clarified by its expression in a language (Haskell) that supports higher order functions, types and type classes." Thus, functional programming exhibits the *low threshold and high ceiling* property for an easily usable but arbitrarily extensible classroom tool[71].

Schanzer et al[68, 69] attribute the favorable results of their Bootstrap methodology to their use of a **functional language** as the medium and to the absence of distracting features. In particular, they claim that their choice of a text-based interface over a more appealing visual program interface is actually a design feature, because "textual syntax is closer to what students must master in mathematics classes".

Functional programming, which fundamentally depends on the concept of recursion, is also expected to facilitate the understanding of mathematical induction. Program recursion and mathematical induction go naturally hand in hand with each other. Recursion is a fundamental construct that is usually considered difficult to teach, but Ginat et al[26] show that this may be a shortcoming of the mainstream procedural languages, and they conclude that "understanding recursion is easier in declarative, abstract levels than procedural ones".

Other ongoing efforts to bring programming to high schools are based on Computational Thinking[5, 27, 57, 51], which focuses on the algorithmic, problem-solving part of programming, and on projects such as CS Unplugged[6], which is based on running the typical CS algorithms without a computer. It is still too soon to say whether courses based on these ideas work as intended.

In summary, while the idea of bringing mathematics and computing together is rather old, there is still no clear consensus on how to best exploit their similarity, and the success rate does not still overweight the failure rate, so we can only hope to slowly add more knowledge points to this simple looking but difficult to accomplish goal.

7 Conclusion and Future Work

We have shown several examples of an ongoing project that aims to introduce programming exercises in the standard high school mathematics curriculum. The use of Haskell as the programming language allowed us to ignore most syntactic concerns and other accessory technical details that accompany the use of more mainstream languages, except possibly Python, which is similarly succint. However, unlike

Python, Haskell also offers the ability to freely mix algebra and programming in a single argument, as the referential integrity of Haskell means that Haskell equations are also equations in the algebraic sense.

We have performed some preliminary pilot tests and are currently under a three-year contract with our local public school district to develop and deploy the project in actual classrooms. We have additional modules ready for navigating the syntax tree of an algebraic expression, for performing symbolic algebra computations, for matrix operations, linear regressions and plotting of functions, including conic sections. All the modules will be published as open source and will be available in a public repository (http://github.com/alphalambda/k12math).

In our pilot studies, we did pre- and post-tests and surveys for both motivation towards mathematics and mathematical knowledge, and while the tests and surveys showed some positive results, they were not significant enough to be presented. Our sample consisted of a total of 15 middle school students and 10 high school students, which was too small to have a quantitative study done. However, our qualitative assessment shows that this approach has potential but needs a carefully designed progression of scaffolded activities. We also confirmed our hypothesis about the adequacy of Haskell as the programming language to use in this type of settings, where the emphasis is stronger on the teaching of mathematics than on the teaching of programming, and a language that does not take the focus away from mathematical thinking is desired. In particular, lazy evaluation is the computation model of hand-written mathematics, and along with pattern-matching and multiple-case function definitions, it provides a programming environment that seamlessly integrates with hand-written mathematics with minimal cognitive effort.

Our future plans also include moving the development environment to a Web-based interface, using either CodeWorld[13] or Blank Canvas[8], since deploying a local compiler in schools is somewhat problematic given their strict policies on software.

8 Bibliography

References

[1] Harold Abelson (1986): *Turtle geometry: The computer as a medium for exploring mathematics*. MIT press.

[2] Angelos Agalianos, Richard Noss & Geoff Whitty (2001): *Logo in mainstream schools: the struggle over the soul of an educational innovation*. British Journal of Sociology of Education 22(4), pp. 479–500.

[3] Michal Armoni, Orni Meerbaum-Salant & Mordechai Ben-Ari (2015): *From Scratch to Real Programming*. Trans. Comput. Educ. 14(4), pp. 25:1–25:15, doi:10.1145/2677087. Available at `http://doi.acm.org/10.1145/2677087`.

[4] Ilana Arnon, Jim Cottrill, Ed Dubinsky, Asuman Oktaç, Solange Roa Fuentes, Maria Trigueros & Kirk Weller (2013): *APOS theory: A framework for research and curriculum development in mathematics education*. Springer Science & Business Media.

[5] Valerie Barr & Chris Stephenson (2011): *Bringing Computational Thinking to K-12: What is Involved and What is the Role of the Computer Science Education Community?* ACM Inroads 2(1), pp. 48–54, doi:10.1145/1929887.1929905. Available at `http://doi.acm.org/10.1145/1929887.1929905`.

[6] Tim Bell, Frances Rosamond & Nancy Casey (2012): *The Multivariate Algorithmic Revolution and Beyond*. chapter Computer Science Unplugged and Related Projects in Math and Computer Science Popularization, Springer-Verlag, Berlin, Heidelberg, pp. 398–456. Available at `http://dl.acm.org/citation.cfm?id=2344236.2344256`.

[7] RK Blank & N de las Alas (2009): *Effects of teacher professional development on gains in student achievement: How meta-analysis provides evidence useful to education leaders*. Washington, DC: The Council of Chief State School Officers.

[8] Blank Canvas: `https://hackage.haskell.org/package/blank-canvas`.

[9] Earl C Butterfield & Gregory D Nelson (1989): *Theory and practice of teaching for transfer*. Educational Technology Research and Development 37(3), pp. 5–38.

[10] Alan CK Cheung & Robert E Slavin (2013): *The effectiveness of educational technology applications for enhancing mathematics achievement in K-12 classrooms: A meta-analysis*. Educational Research Review 9, pp. 88–113.

[11] S. S. Choi-Koh (1999): *A student's learning of geometry using the computer*. The Journal of Educational Research 92(5), pp. 301–311.

[12] Douglas H. Clements, Michael T. Battista & Julie Sarama (2001): *Logo and Geometry*. Journal for Research in Mathematics Education. Monograph 10, pp. 1–177.

[13] Codeworld: `https://github.com/google/codeworld`.

[14] Common Core Standards for Mathematics: `http://www.corestandards.org/math/`.

[15] Committee for the Workshops on Computational Thinking (2010): *Report of a Workshop on the Pedagogical Aspects of Computational Thinking*. National Research Council.

[16] Robert L. Constable (2002): *Naïve Computational Type Theory*. In H. Schwichtenberg & R. Steinbruggen, editors: Proof and System-Reliability, pp. 213–259.

[17] José N Contreras (2011): *Using Technology to Unify Geometric Theorems about the Power of a Point.* Mathematics Educator 21(1), pp. 11–21.

[18] National Research Council (2010): *Report of a Workshop on the Scope and Nature of Computational Thinking*. National Academies Press.

[19] CSTA K-12 Computer Science Standards: `http://www.csta.acm.org/Curriculum/sub/K12Standards.html`.

[20] Stephen Emmott (2006): *Towards 2020 science*. Microsoft Research.

[21] M. Felleisen, R.B. Findler, M. Flatt & S. Krishnamurthi (2001): *How to Design Programs*. MIT Press.

[22] Matthias Felleisen (2010): *TeachScheme!- A Checkpoint*. In: The 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010).

[23] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2009): *A Functional I/O System or, Fun for Freshman Kids*. In: The 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009).

[24] Matthias Felleisen & Shriram Krishnamurthi (2009): *Viewpoint: Why Computer Science Doesn't Matter*. Communications of the Association for Computing Machinery 52, pp. 37–40.

[25] Wallace Feurzeig, Seymour A Papert & Bob Lawler (2011): *Programming-languages as a conceptual framework for teaching mathematics*. Interactive Learning Environments 19(5), pp. 487–501.

[26] David Ginat & Eyal Shifroni (1999): *Teaching Recursion in a Procedural Environment - How much should we emphasize the Computing Model?* In: Proceeding SIGCSE '99. The proceedings of the thirtieth SIGCSE technical symposium on computer science education, ACM, New York, pp. 127–131.

[27] Shuchi Grover & Roy Pea (2013): *Computational Thinking in K–12 A Review of the State of the Field*. Educational Researcher 42(1), pp. 38–43.

[28] Allison Gulamhussein (2013): *Teaching the teachers: Effective professional development in an era of high stakes accountability*. Center for Public Education. September.

[29] Orit Hazzan (2002): *The reflective practitioner perspective in software engineering education*. Journal of Systems and Software 63(3), pp. 161–171.

[30] Orit Hazzan & James E Tomayko (2005): *Reflection and abstraction in learning software engineering's human aspects*. Computer 38(6), pp. 39–45.

[31] K. F. Hollebrands (2003): *High school students' understandings of geometric transformations in the context of a technological environment*. Journal of Mathematical Behavior 22(1), pp. 55–72.

[32] K. F. Hollebrands (2007): *The role of a dynamic software program for geometry in the strategies high school mathematics students employ*. Journal for Research in Mathematics Education 38(2), pp. 164–192.

[33] Celia Hoyles & Richard Noss (1987): *Synthesizing mathematical conceptions and their formalization through the construction of a Logo-based school mathematics curriculum*. International Journal of Mathematical Education in Science and Technology 18(4), pp. 581–595.

[34] Celia Hoyles & Richard Noss (1992): *Learning mathematics and logo*. MIT Press.

[35] Sarah Hug, Josh Sandry, Ryan Vordermann, Enrico Pontelli & Ben Wright (2013): *DISSECT: Integrating Computational Thinking in the Traditional K-12 Curricula Through Collaborative Teaching (Abstract Only)*. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13, ACM, New York, NY, USA, pp. 742–742, doi:10.1145/2445196.2445452. Available at `http://doi.acm.org/10.1145/2445196.2445452`.

[36] Keith Jones (2005): *Using Logo in the Teaching and Learning of Mathematics: a research bibliography*. MicroMath 21(3), pp. 34–36.

[37] Cenk Keşan & S Çalişkan (2013): *The effect of learning geometry topics of 7th grade in primary education with the Dynamic Geometer's Sketchpad geometry software to success and retention*. Turkish Online Journal Of Educational Technology 12(1).

[38] J. Kramer (2003): *Abstraction - is it teachable? 'the devil is in the detail'*. In: Proceedings. 16th Conference on Software Engineering Education and Training. (CSEE&T 2003), The Institute of Electrical and Electronics Engineers, Inc, p. 32.

[39] Jeff Kramer (2007): *Is abstraction the key to computing?* Communications of the ACM 50(4), pp. 36–42.

[40] D Midian Kurland, Roy D Pea, Catherine Clement & Ronald Mawby (1986): *A study of the development of programming ability and thinking skills in high school students*. Journal of Educational Computing Research 2(4), pp. 429–458.

[41] J John Loughran (2002): *Effective reflective practice in search of meaning in learning about teaching*. Journal of Teacher Education 53(1), pp. 33–43.

[42] 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.

[43] Richard E Mayer (2004): *Should there be a three-strikes rule against pure discovery learning?* American Psychologist 59(1), p. 14.

[44] Barry McBrien (2007): *Learning from practice–Reflections on a critical incident*. Accident and emergency nursing 15(3), pp. 128–133.

[45] Leah P. McCoy (1996): *Computer-based mathematics learning*. Journal of Research on Computing in Education 28(4), p. 438. Available at `http://libezp.lib.lsu.edu/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=9609115664&site=ehost-live&scope=site`.

[46] Katherine L McEldoon, Kelley L Durkin & Bethany Rittle-Johnson (2013): *Is self-explanation worth the time? A comparison to additional practice*. British Journal of Educational Psychology 83(4), pp. 615–632.

[47] Don Monroe (2013): *A New Type of Mathematics? New discoveries expand the scope of computer-assisted proofs of theorems*. Communications of the Association for Computing Machinery 57, pp. 13–15.

[48] National Center for Education Statistics: `https://nces.ed.gov/fastfacts`.

[49] T J Nokes, C D Schunn & M T H Chi (2010): *Problem Solving and Human Expertise*. International Encyclopedia of Education, vol. 5, pp. 265-272.

[50] Richard Noss (1986): *Constructing a conceptual framework for elementary algebra through Logo programming*. Educational Studies in Mathematics 17(4), pp. 335–357.

[51] Rex Page & Ruben Gamboa (2013): *How Computers Work: Computational Thinking for Everyone*. In M. Morazán and P. Achten, editor: *Trends in Functional Programming in Education 2012 (TFPIE 2012)*, Electronic Proceedings in Theoretical Computer Science, vol. 106, pp. 1–19.

[52] David B Palumbo (1990): *Programming language/problem-solving research: A review of relevant issues*. *Review of Educational Research* 60(1), pp. 65–89.

[53] Seymour Papert (1980): *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.

[54] Michael Quinn Patton (2011): *Developmental evaluation: Applying complexity concepts to enhance innovation and use*. Guilford Press.

[55] Roy D. Pea (1983): *Logo Programming and Problem Solving. [Technical Report No. 12]*.

[56] James W Pellegrino, Margaret L Hilton et al. (2013): *Education for life and work: Developing transferable knowledge and skills in the 21st century*. National Academies Press.

[57] Pat Phillips (2009): *Computational Thinking: a problem-solving tool for every classroom*. *Communications of the CSTA* 3(6), pp. 12–16.

[58] Renuwat Phonguttha, Sombat Tayraukham & Prasart Nuangchalerm (2009): *Comparisons of Mathematics Achievement, Attitude towards Mathematics and Analytical Thinking between Using the Geometer's Sketchpad Program as Media and Conventional Learning Activities*. *Online Submission* 3(3), pp. 3036–3039.

[59] Jan L Plass, Bruce D Homer & Elizabeth O Hayward (2009): *Design factors for educationally effective animations and simulations*. *Journal of Computing in Higher Education* 21(1), pp. 31–61.

[60] Marc Prensky (2008): *Programming is the new literacy*. *Edutopia magazine*.

[61] Prabhakar Ragde (2013): *Mathematics Is Imprecise*. *arXiv preprint arXiv:1301.5076*.

[62] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman & Yasmin Kafai (2009): *Scratch: Programming for All*. *Communications of the Association for Computing Machinery* 52, pp. 60–67.

[63] Peter J. Rich, Keith R. Leatham & Geoffrey A. Wright (2013): *Convergent cognition*. *Instr. Sci.* 41, pp. 431–453.

[64] PeterJ. Rich, KeithR. Leatham & GeoffreyA. Wright (2013): *Convergent cognition*. *Instructional Science* 41(2), pp. 431–453, doi:10.1007/s11251-012-9240-7. Available at `http://dx.doi.org/10.1007/s11251-012-9240-7`.

[65] B Rittle-Johnson, M Saylor & K. E. Swygert (2008): *Learning from explaining: does it matter if mom is listening?* *J. Exp. Child Psychol.* 100, pp. 215–24.

[66] Spencer Rugaber (2000): *The Use of Domain Knowledge in Program Understanding*. *Annals of Software Engineering* 2000, pp. 9–143.

[67] Douglas Rushkoff: *Teach U.S. kids to write computer code*. `http://www.cnn.com/2012/12/10/opinion/rushkoff-code-writing/`.

[68] Emmanuel Schanzer, Kathi Fisler & Shriram Krishnamurthi (2013): *Bootstrap: Going Beyond Programming in After-School Computer Science*. SPLASH-E (Education track of the OOPSLA/SPLASH conference).

[69] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi & Matthias Felleisen (2015): *Transferring Skills at Solving Word Problems from Computing to Algebra Through Bootstrap*. *Circles* 6(4), p. 5.

[70] Donald A. Schön (1983): *The reflective practitioner: how professionals think in action*. New York: Basic Books.

[71] Pratim Sengupta, John S. Kinnebrew, Satabdi Basu, Gautam Biswas & Douglas Clark (2013): *Integrating Computational Thinking with K-12 Science Education Using Agent-based Computation: A Theoretical Framework*. *Education and Information Technologies* 18(2), pp. 351–380, doi:10.1007/s10639-012-9240-x. Available at `http://dx.doi.org/10.1007/s10639-012-9240-x`.

[72] Nick Senske (2011): *A Curriculum for Integrating Computational Thinking*. In: *Parametricism: ACADIA Regional Conference Proceedings. Lincoln, NE.*

[73] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney & Kate Sanders (2006): *Commonsense computing: what students know before we teach (episode 1: sorting)*. In: *Proceedings of the second international workshop on Computing education research*, ACM, pp. 29–40.

[74] Nathalie Sinclair (2008): *The History of the Geometry Curriculum in the United States*. Charlotte: Information Age Publishing, Inc.

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

[76] Juha Sorva et al. (2012): *Visual program simulation in introductory programming education*.

[77] Sheryn Spencer-Waterman (2014): *Handbook on Differentiated Instruction for Middle & High Schools*. Routledge.

[78] SRI International (2014): *Principled Assessment of Computational Thinking*. `http://pact.sri.com/publications.html`.

[79] Jurriën Stutterheim, Wouter Swierstra & Doaitse Swierstra (2013): *Forty hours of declarative programming: Teaching Prolog at the Junior College Utrecht*. In: *Proceedings TFPIE 2012, arXiv:1301.4650.*

[80] Taisir Subhi (1999): *The impact of LOGO on gifted children's achievement and creativity*. *Journal of Computer Assisted Learning* 15(2), pp. 98–108.

[81] Marilla D Svinicki (2010): *A guidebook on conceptual frameworks for research in engineering education*. *Rigorous Research in Engineering Education NSF DUE-0341127, DUE-0817461.*

[82] Allison Elliott Tew (2010): *Assessing fundamental introductory computing concept knowledge in a language independent manner*.

[83] Annette Vee (2013): *Understanding Computer Programming as a Literacy*. *Literacy in Composition Studies* 1(2).

[84] Philip Wadler (2012): *Church's Coincidences*. In: *Keynote SICSA PhD Conference.*

[85] Philip Wadler (2014): *Propositions as types*. *Unpublished note, http://homepages. inf. ed. ac. uk/wadler/papers/propositions-as-types/propositions-as-types. pdf.*

[86] Scott N Walck (2014): *Learn Physics by Programming in Haskell*. *arXiv preprint arXiv:1412.4880.*

[87] Sylvia Weir (1987): *Cultivating minds: A Logo casebook.* ERIC.

[88] Joseph J. Williams & Tania Lombrozo (2010): *The Role of Explanation in Discovery and Generalization: Evidence From Category Learning*. *Cognitive Science* 34, p. 776806.

[89] Jeannette M Wing (2006): *Computational thinking*. *Communications of the ACM* 49(3), pp. 33–35.

[90] Jeannette M Wing (2008): *Computational thinking and thinking about computing*. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366(1881), pp. 3717–3725.

[91] Geoff Wright, Robert Lee, Peter Rich & Keith Leatham (2011): *An Analysis of the Influence on Students Mathematics Skills Participating in the Bootstrap Programming Course*. In: *World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education*, 2011, pp. 986–991.

[92] Geoff Wright, Peter Rich & Robert Lee (2013): *The influence of teaching programming on learning mathematics*. In R. McBride & M. Searson, editors: *Society for Information Technology & Teacher Education International Conference*, 2013, pp. 4612–4615.

[93] Geoffrey A. Wright, Peter Rich & Keith R. Leatham (2012): *How Programming Fits With Technology Education Curriculum*. *Technology and Engineering Teacher* April, pp. 3–9.

[94] Nicola Yelland (1995): *Mindstorms or a storm in a teacup? A review of research with Logo*. *International Journal of Mathematical Education in Science and Technology* 26(6), pp. 853–869, doi:10.1080/0020739950260607. Available at `http://dx.doi.org/10.1080/0020739950260607`.