Using Elm to Introduce Algebraic Thinking to K-8 Students

Curtis d'Alves, Tanya Bouman, Christopher Schankula, Jenell Hogg,
Levin Noronha, Emily Horsman, Rumsha Siddiqui, Christopher Kumar Anand
McMaster University
Hamilton, Ontario, Canada
{dalvescb,boumante,schankuc,anandc}@mcmaster.ca

Recent years have seen a large increase in the interest in developing a Computer Science curriculum for K-8 students. However, there have been significant barriers to creating and deploying a Computer Science curriculum in many areas, including teacher time and the prioritization of other 21st-century skills. At McMaster University, we have developed both general computer literacy activities and specific programming activities. Each activity also supports existing curricular goals, making them easy for teachers to include in their planning. In this paper, we focus on programming in the functional language Elm and the graphics library GraphicSVG. Elm is in the ML family, with a lean syntax and easy inclusion of Domain Specific Languages. This allows children to start experimenting with GraphicSVG as a language for describing shape, and pick up the core Elm language as they grow in sophistication. Teachers see children making connections between computer graphics and mathematics within the first hour. Graphics are defined declaratively, and support aggregation and transformation, i.e., Algebra. Variables are not needed initially, but are introduced as a time-saving feature, which is immediately accepted. Since variables are declarative, they match students' expectations. Advanced students are also exposed to State, and the syntax required to implement touches and clicks closely follows the theoretical concepts, making it easy to understand how it works. For each of these concepts, the exposition is a condensation of the presentations we make to students, and can be used as lesson plans, to duplicate the success we have had, teaching 5200 children in 2016.

Finally, we describe ongoing work on a touch-based Elm/GraphicSVG editor for iPads, which features (1) type highlighting (as opposed to mere syntax highlighting), (2) preservation of correct syntax and typing across transformations, (3) context information (e.g. displaying parameter names for GraphicSVG functions), and (4) immediate feedback (e.g. restarting animations after every program change).

1 Introduction

There is a lot of interest in developing a Computer Science curriculum for K-8 [15], with curricula being defined by some education authorities, e.g. year-by-year goals in the UK [1]. At McMaster University, we have developed a number of outreach activities in collaboration with local teachers and their students, and have been motivated by our successes and failures to develop a learning environment where children can develop programming skills initially through exploration and then through discovery. As we will describe, this is an update to the approach first advocated by Papert and coworkers at MIT [14].

After trial and error, we have found a Domain Specific Language for two-dimensional drawing embedded in the functional language Elm to be by far the most successful. Matching the semantics (and vocabulary) of drawing with stencils allows children to begin exploring with very little stumbling over new concepts. Children can start right away modifying a list of shapes, so initial learning is restricted to a short list of functions with small numbers of obvious arguments and syntax for lists and forward function application—pipelining—to better expose the combinatorial nature of shape construction and composi-

Submitted to:
TFPIE 2017

© d'Alves, Bouman, Schankula, Hogg, Noronha, Horsman, Siddiqui & Anand
This work is licensed under the
Creative Commons Attribution License.

tion. Almost all language features can be introduced in a planned way after children are confident in their ability to handle programming.

Younger children (less than 10 years old) do still stumble over syntax, probably because they are just discovering grammar in English, so they cannot build on that knowledge. For these students we will describe ongoing work to create an error-free editor just for them.

In all cases, using a language in the ML family is important because

- pure functions match the child's inherent idea of tools, like stencils, pens and paint brushes,
- it is easy to create a domain-specific language,
- there is minimal syntax to learn.

1.1 Adoption

Teachers are busy, and the curriculum is already full enough. While progress in software technologies are viewed as positive by most computer scientists, many developed regions are already suffering from high unemployment attributed to automation, and the coming changes are predicted to displace many more [8]. In this context, teachers and schools in our area are trying to develop creativity and teamwork (21st-century skills), which further squeezes time for pre-Computer Science or Computational Thinking. And while adding new skills, the fundamental importance of reading cannot be neglected. In this context, what we need is a method of introducing Computer Science that provides at least as much progress toward existing curricular goals as the instruction it will displace.

Fortunately, we have earned the trust of teachers on this point:

> This morning's coding session was fantastic! Love that I see math curriculum connections with integers and placing co-ordinates on a grid!

So, the connection with coordinates meets teachers' needs to cover curriculum, and their students are so excited by their ability to independently explore the material, that (at least anecdotally) their engagement spills over into other subject areas.

> My students were still talking about [the workshop] last week over the last few days so it certainly had an impact on their learning and engagement!

So, although we see the acquisition of algebraic thinking as the bigger and more important target, the visible gains in enthusiasm for geometry easily justifies the time taken, while we figure out how to measure the impact on algebra.

1.2 Algebraic Thinking

Why do we characterize our approach as "algebraic thinking", and not "computational thinking" or "algebraic reasoning" or "algebraic program construction"? The term "computational thinking" was introduced by Wing [20] to draw attention to the need for increased enrolment in CS at all levels. There are now many outreach programs aimed at K-12, including well-funded non-profit organizations, however there are also some pointed questions being asked. In particular, what is the difference between "computational thinking" of 2010 and "programming" of 1980, and the answer seems to be not very much. This is despite work by scholars of literacy: "code's discreteness also enables one to build complex and chained procedures with the confidence that the computer will interpret them as precisely as one writes them [...] For these reasons, computational literacy is not simply a literacy practice—a subset of textual

literacy. It is instead a (potential) literacy on its own, with a complex relationship to textual literacy. " [18]

Recently, Guzdial [9] surveyed the literature on teaching computer science, focussing on K-12. He is also critical of the simple definition of "computation thinking", and he discusses at length Soloway's Rainfall problem which highlights the consistent failure of conventional approaches to teach even undergraduates the basics of programming. In contrast, Fisler found that students using functional languages "made fewer errors than in prior Rainfall studies and used a diverse set of high-level composition structures" [7]. So there is some evidence that a functional approach is superior for beginning undergraduate and high-school learners. Such an approach could be called "functional thinking".

We prefer to call our approach "algebraic thinking", going back to the original definition of algebra as "taking apart and putting together". By focusing on shapes, children learn about recursive (tree) data structures by building increasingly complex pictures. By decomposing shape construction into Stencils, Shapes and Shape Transformers, we expose the combinatorial structure of shapes. Children's "inner scientist" loves experimenting with the possibilities, and rapidly assimilates the patterns.

Patterning is used in a more restrictive sense in K-8, but it is a recognized part of pre-algebra [12]. We have found a growing awareness of the importance of algebra among teachers, in parallel with research finding that high schools needing to offer fewer (remedial) pre-algebra math courses have lower drop out rates [13].

While Soloway's Rainfall problem would seem artificial to our target age group, we have developed our own metric of success: "is the language and activity compatible with socially constructive learning?" Socially constructive learning is thought to be better for acquiring higher-level skills, but it can only work if children are engaged enough to stay on task (or, even better, invent their own tasks). When we used Python, we needed to interleave girls and boys in the lab as a way to prevent friends from distracting each other from their programming task. However, after switching to Elm, we can encourage children to consult each other. Now, if we one team asks for help to make their shape blink, we will soon see half the class following suit in their animations.

1.3 Why Elm?

Elm (http://elm-lang.org) was imagined as a vehicle for delivering best practices in program language design to web front-end developers [6]. It is deliberately simple, for example, dropping support for Functional Reactive Programming in version 0.17 [2]. It looks like other ML-derived languages, but it does not have user-defined type classes, although it has built-in classes such as number, comparable and show. It has a foreign-function mechanism designed with safety first. When it was initially developed, it was the natural language for Haskell programmers needing to create a JavaScript app, and was a radical simplification of the complexity of JavaScript, HTML and CSS. Today, such developers should also look at PureScript (http://www.purescript.org), TypeScript (https://www.typescriptlang.org/), or Flow (https://flow.org/).

As a teaching language, Elm offers simplicity in both syntax and semantics, and because it compiles to JavaScript, it is accessible in schools without installing software. It works on every platform with a web browser, and it is easy to build collaboration and distance-learning tools around it.

1.4 The Elm Architecture

In Elm 0.17, Functional Reactive Programming features were removed in favour of The Elm Architecture (TEA) ([2]) with subscriptions. It has three parts, which in our case are typed as

- a `model` data type for state, with an initial value,
- `view : model -> Collage (Msg msgs)`, and
- `update : msgs -> model -> model`.

The `model` is a data type which contains information about the current state of the program. For example, the model might contain information about the current time and what slide you're on in the case of a presentation.

The `view` function visually representing the state stored in the `model`, usually through HTML, or, in the case of GraphicSVG, through SVG. Elm's functional paradigm eliminates many pitfalls found in traditional web development, as it guarantees that the same state always produces the same view.

The `update` function takes both the `model` and a data type (known as the "Message") as a parameter. The `update` function implements all transition functions from one state of the program to the next. Some messages, such as time updates, are requested from the Elm runtime, while special transformers that can be applied to any Shape tell the run-time to generate specific messages based on user interaction. Elm's runtime redraws automatically based on need.

1.5 GraphicSVG

Our Graphics library for teaching, GraphicSVG, is based on the original Elm Graphics module which targeted HTML canvas elements, and it is partially backwards compatible. GraphicSVG's principal types (`Stencil`, `Shape` and `Collage`) model real-world concepts: `Stencil` describes a recipe for creating a shape; for example, a circle with a certain radius, a rectangle with a width and height or text with a certain font and size:

```
circle, square, triangle: Float -> Stencil
rect, oval: Float -> Float -> Stencil
roundedRect: Float -> Float -> Float -> Stencil
text: String -> Stencil
```

But, like a real-life stencil, a visible shape is not created until the user fills it in or traces its edge:

```
filled: Color -> Stencil -> Shape a
outlined: LineType -> Stencil -> Shape a
```

Thus, a concrete analogy explains why shapes cannot show up on the screen unless they are filled or outlined. This architecture limits the number of parameters each function takes, and the types match students intuition closely enough that we do not have to talk about them. They never use constructors for these types directly, using exposed functions instead, some of which simplify the underlying type construction. Fortunately, Elm's type errors (e.g. found a `Stencil` where a `Shape` was expected) also match their intuitive understanding of these types, and need little explanation, and, so far, students who choose to attempt more complicated user interaction are able to build a workable understanding of Elm types from there. (See the last section to learn how we plan to use types as a teaching tool in the future.)

1.5.1 Design Motivation: Instructional Scaffolding

One aspect of Elm's original Graphics library that we found successful in classrooms was its support for instructional scaffolding. [4] The instructor shows students how to draw basic shapes on the centre of the collage. The next inquiry most students have is a logical progression from the current state: "How can we

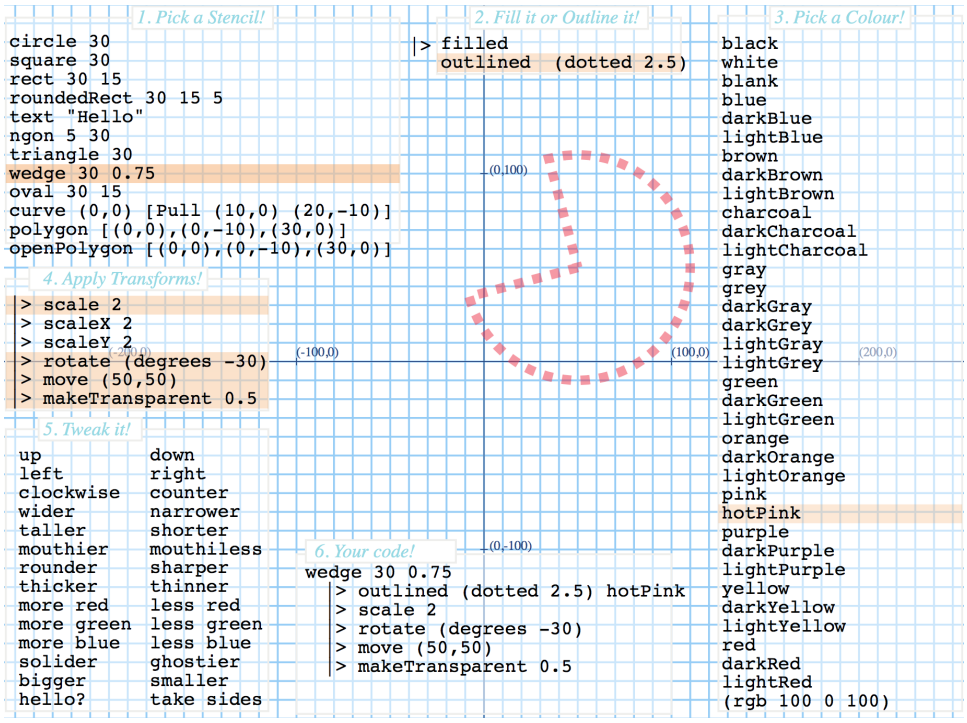


Figure 1: ShapeCreator: As a consequence of GraphicSVG's design, we were able to expose the combinatorial nature of shape construction in an interactive tool for API discovery.

move the shape from the centre of the collage?" Like with `filled`, we use forward function application (`|>`)[1] to apply transformations to `Shapes`, thereby visually laying out the combinatorial nature of shape construction.

These functions have type (parameters) `-> Shape -> Shape`:

```
move: (Float, Float) -> Shape a -> Shape a
scale: Float -> Shape a -> Shape a
rotate: Float -> Shape a -> Shape a
```

There are several advantages of this approach. First and foremost, it allows a very fast startup. By separating transformations from the shape itself, within a minute or two, the instructor can create a shape on screen simply by defining the bare minimum amount of information; for example, a radius and a colour is all that is needed for a circle, which follows the students' expectations about how to represent shapes. It is also easier to remember parameter order than for similar functions in other languages which either have multiple parameters, confounding size and position, or use stateful drawing models.

For many students, this is their first exposure to a "real" programming language—or any programming language—and as such, seeing text produce a shape onscreen is very exciting, and encourages them to ask questions which can lead the rest of the presentation.

1.5.2 GraphicSVG Apps

GraphicSVG contains three types of "apps" graded by complexity.

[1] Elm provides two function application operators, `<|` is like Haskell's `$` and `|>` flips the argument order.

The first and most basic one, `graphicsApp`, allows the static drawing of graphics on the screen. It can be easily understood as requiring only the "view" portion of The Elm Architecture's model-view-update architecture, hiding the ideas of model and update from the student.

An intermediate app, `notificationsApp` allows the user to add interaction to their graphics by adding notify transformers which cause messages to be sent to the student's update function when, for example, a shape is clicked:

```
notifyTap: msg -> Shape a -> Shape a
notifyEnter: msg -> Shape a -> Shape a
notifyLeave: msg -> Shape a -> Shape a
```

See § 2.2 for an example using notificationsApp to add interaction to a GraphicSVG application.

Finally, `gameApp` provides the functionality of `notificationsApp`, but also has a parameter for a special kind of `Tick` message type, which, on each frame, will send their `update` function the time in seconds since the app was started as well as information about keyboard presses. Given that `gameApp` has only thus far been used for sporadic half-day workshops, few students have thus far taken advantage of the advanced features provided by `notificationsApp` and `gameApp`, other than support for animation. However, `gameApp` has been used internally to develop game templates used for day-long Hackathons where students compete in teams to create educational games, which have been very well received.

1.6 Additional Related Work

As a rule, tools meant for experts are not suitable for learners. This is certainly true of integrated development environments with lots of key-press-saving state. Many practitioners who see functional programming as the reserve of elite programmers have asked us why we use it for beginners. In this case, functional programming's fundamental advantages actually play out in the favour of beginners. Pure functions are much easier for beginners to reason about. Declarative "variables" match preexisting expectations from algebra, and will not cause confusion for students who have yet to learn algebra. They are readily accepted as being shortcuts to avoid typing the same thing over and over. And finally, the surfacing of structure in functional programs is of benefit to designers at all experience levels. Hughes' observations apply equally to beginners [10], although they need to be incorporated into the instructional design, creating a new shared experience for these students, because it cannot be described relative to previous practices which have no meaning to beginners.

Our separation of Stencils from Shapes and use of transformation functions parallels the approach taken by Walck in creating a similar platform for exploration of somewhat more advanced material [19].

2 Educational Psychology

Social constructivism is the theory of learning which focusses on peer interactions, especially the sharing of information and construction of shared explanations and procedures. As previously mentioned, it is closely associated with Papert, but although far from novel, it is underexploited. We have relied on it in our other outreach activities, which as inspired our attempts to use it more for Elm programming. We describe the two most important activities: when teaching binary numbers, we encourage children to help each other out in using our iPad app Image 2 Bits (link), in which students encode black and white images with binary numbers. The students each create an image on their iPad; the iPads share the encoding of that image with other students, and the students then decode each other's images. Once they

are done decoding the image, students have the option to send a "like" to the student who created the image. Similarly, when teaching a (simplified version of) Computed Aided Tomography (CAT scans), we rely on group reinforcement, motivated by a game. Students arrange themselves in a grid to represent the body part being imaged. They act out the x-rays passing through the body and partly being absorbed by bone and muscle tissue. Each student represents either a piece of muscle or bone. The x-rays pass directly through the muscle tissue, but get partly blocked by the bone tissue. In order to correctly calculate where the bone and muscle tissues are, each student in the group must pay attention and correctly report how much light comes through. Then the students work together to solve where the bone and muscle tissue are. Where the numbers allow, we have two or more groups work in parallel, competing to reconstruct their image first. That games have rules is no surprise, and they immediately realize that everyone must fully understand their role, or the whole effort will collapse. Furthermore, they actively think about the algorithm they are applying in an effort to find "shortcuts".

One of the advantages of focussing on graphics so early, is that it allows every student or small group of students to create their own image. Working in pairs works best for most students, and since each team has their own project, interactions between teams is positive and focussed on sharing techniques. We encourage students to think about it in this way, and we present more advanced topics as tips they may find useful in accelerating their development, or incorporating interesting new features. None of our students know trigonometry, so we give them an example using sine to make an object move back and forth, usually after one group has already asked about it. They have no trouble incorporating time as a variable any place a number can be used. Once they start repeating elements, we introduce variables, and once they ask about user interaction, we introduce state.

2.1 Variables and Functions: a tool for code reuse

Graphics can be used as an effective way for students to learn the fundamental concepts behind mathematical functions. After a brief introduction of the "input-process-ouput" model, we asked a class of Grade 7/8 students to come up with real-world examples of processes which can be described using this model. They shared a wide variety of ideas: printers (inputs: blank paper, ink, information, energy; process: place ink on to paper in the correct pattern; output: printed paper), schoolwork (inputs: constraints, objectives, rubrics; process: completing the work according to necessary steps; output: completed assignment/presentation) and, somewhat comically, paperwork (inputs: forms to fill out; process: collect and write down information; output: completed paperwork), among others. The mentor stressed that almost any real-world process can be generalized using this idea.

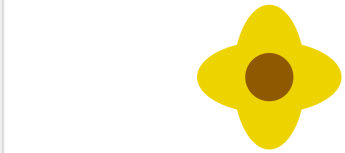
Directly following the discussion, we used the example of the flower in (Figure 2). The mentor demonstrated how defining a separate variable with a GraphicSVG group,

```
group: List (Shape a) -> Shape a ,
```

can be used to easily create copies of a graphic on the collage plane. However, this time we presented a new motivation: how can we create copies of the flower which are different colours? Once again, students suggested the obvious "copy and paste" approach, with `flowerRed`, `flowerGreen`, `flowerBlue`, etc. variables being created and then referenced in their collage. The volunteer instructor then emphasized the point that similar code should be reused, instead of duplicated. The ultimate motivation then becomes that learning how functions apply to Elm would benefit their more rapid and straightforward creation of new artwork and animations. Thus, the concept of functions was presented as a "tip" rather than a traditional lesson. The instructor then demonstrated how to add an input to the flower function, a term with which they were familiar due to the aforementioned discussion:

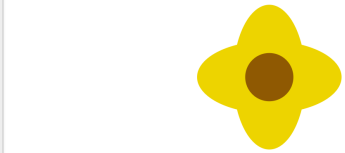
a)

```
myShapes = [
               oval 60 30
                 |> filled yellow
           ,   oval 30 60
                 |> filled yellow
           ,   circle 10
                 |> filled darkBrown
           ]
```



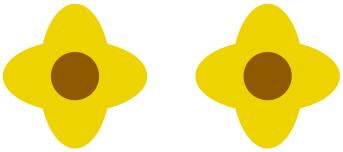
b)

```
myShapes = [
               flower
           ]

flower = group [
                   oval 60 30
                     |> filled yellow
               ,   oval 30 60
                     |> filled yellow
               ,   circle 10
                     |> filled darkBrown
               ]
```



c)

```
myShapes = [
               flower
                 |> move(-40,0)
           ,   flower
                 |> move(40,0)
           ]

flower = group [
                   oval 60 30
                     |> filled yellow
               ,   oval 30 60
                     |> filled yellow
               ,   circle 10
                     |> filled darkBrown
               ]
```



d)

```
myShapes = [
               flower pink green
                 |> move(-40,0)
           ,   flower red orange
                 |> move(40,0)
           ]

flower colour1 colour2 =
  group [
           oval 60 30
             |> filled colour1
       ,   oval 30 60
             |> filled colour1
       ,   circle 10
             |> filled colour2
       ]
```

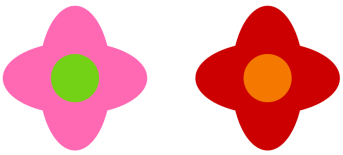


Figure 2: Introducing variables and functions as progressively more powerful ways of reusing code. Initially in a) there is one flower, and without instruction, children will copy and paste the shapes to produce multiple flowers, but in b) they are shown that the "flower" can be given a name `flower` to make the purpose of the shapes clear, and so that in c) it can be used multiple times. Finally in d) it can be transformed into a function to create less boring flowers.

```
flower colour = group [...]
```

upon receiving a compiler error on the projection screen, students were then able to successfully determine that the colour would need to be added after the name of the shape in the `myShapes` collage:

```
myShapes = group [
                      flower green
                           |> move(-50,0)
                  ,   flower blue
                           |> move(50,0)
                  ]
```

Upon compiling the program, the students then found that the flowers had not yet changed colour. After some discussion, they were able to determine that the `colour` variable must be used in lieu of a specific colour:

```
[oval 50 30 |> filled colour]
```

After this guided example, students were able to independently come up with a strategy when asked: what if we want to be able to set an arbitrary colour for the centre of the flower? After a subtle hint of adding a "1" to the colour input, becoming `colour1`, students were able to leverage their knowledge from creating a single-input function and solve the same kinds of problems that came up in the single-input example, which allowed them to describe to the mentor how to implement a function with two variables using Elm (Figure 2 d)). Thus, students were able to map the "input-process-output" concept to three different realms: an abstract concept, describing real-world processes, and, finally, implementing the idea algebraically by leveraging graphics programming with GraphicSVG. Many students made the connection to their math classes where they had begun to talk about mathematical functions and lines but were motivated by being able to use the concept to improve their Elm graphics. Future work would include devising methods to determine quantitatively if the students who learned functions in terms of Elm programming have a more concrete idea of the concept than students who learned it only mathematically.

2.2 Interaction

Although the Elm Architecture depends on separating state, updates and views, we do not explain state until we are ready to introduce interaction. Although we use different types of interaction in our learning tools, we only teach touch or click interactions. Any object can act as a button by simply adding the `notifyTap` transformer in the same way that they add spatial transformation. So, unlike with most languages, syntax is a very low barrier, and the essential barrier is understanding state and transformations. To make it memorable, we introduce state by putting up a simple state diagram, see Figure 3, and acting it out until the whole class understands the game. Children are very attuned to games, and understand the need to learn the rules, so by making a State Diagram into a game, they control for their own learning.

Once they have understood states and transitions, we can show them how to translate the state diagram into Elm by turning each state into a constructor for a state data type, and each transition function into a function. We have only taught interaction to self-selected groups of children, so we have been able to jump from the translation of one diagram to the identification of other state diagrams they encounter in real life, and then state as it exists in games. After that, we show them a simple example like Figure 4, and point out that the time they have been using in their animations is also as state which was updated for them in code they were ignoring from the game template. There are only four steps to adding support for interaction:

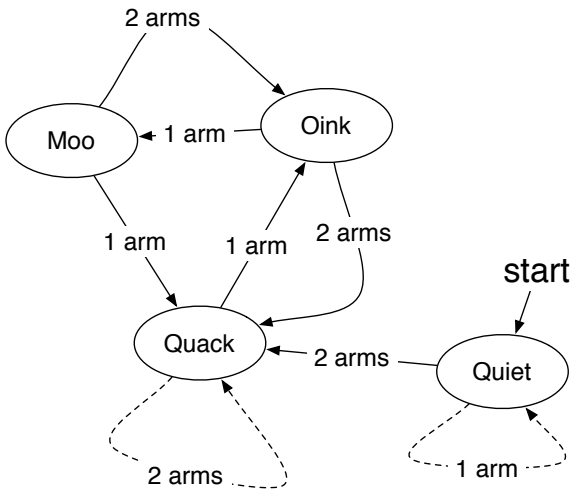


Figure 3: Without explanation, students are given a state diagram similar to this one, and an instructor waving arms (or clapping, flashing lights, etc.). It doesn't take long for them to understand what state is.

1. adding messages to a user-defined message type,
2. adding a notification transformer with the appropriate message to any "buttons",
3. adding to the update function, possibly calling simple state transformation functions, and
4. adding initial values for any new state components.

Of course they can make arbitrarily complex use of state in their existing views.

```
type Msg = Tick Float GetKeyState
         | MoreRed                                          -- (1)

myShapes model = [ circle 50
                     |> filled (rgb model.red 0 100)
                     |> notifyTap MoreRed                   -- (2)
                 ]

update msg model = case msg of
                     Tick t _ -> { model | time = t }
                     MoreRed  -> { model | red = moreRed model.red }
                                                            -- (3)

moreRed red = if red < 250 then red + 5 else 255

init = { time = 0, red = 0 }                                -- (4)

```

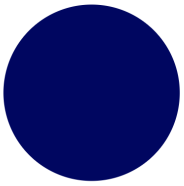


Figure 4: When students are familiar with animations using model.time, it is relatively painless to add user interaction by (1) adding messages to the `Msg` type, (2) adding a notify transformer to any shape, (3) adding a case to the update function, and (4) adding initial values to any new state components.

2.3 Distance Education

Also in the theme of providing them tips, we have created an online mentoring system. Each student signs in and pick a "game slot" to work in. Some slots have challenges with specific goals, like adding a new ice cream flavour to a vending machine, but most students opt to create pictures or animations in one of the free-form slots, or one of one of the slots containing a template game. Each slot has a chat,

accessible only by that student in that slot. See Figure 5. Mentors have access to a view of all active discussions, Figure 6, which indicate which discussions have unanswered questions.

The discussions are presented as "help lines" where students can ask any questions about their code (working or broken), and mentors can send back answers or even links to differences between the student's code and code modified by the mentor. The mentor's modifications are to a private copy, not the student's code, or the code of another mentor.

This system is new, but students are already using it to ask questions after school hours, as well as to get answers faster during lab time when the on-site instructors are busy helping another student. We anticipate using it to support teachers who have received training, but who would not attempt to teach programming without such backup support. We have two teachers who plan to pilot this usage.

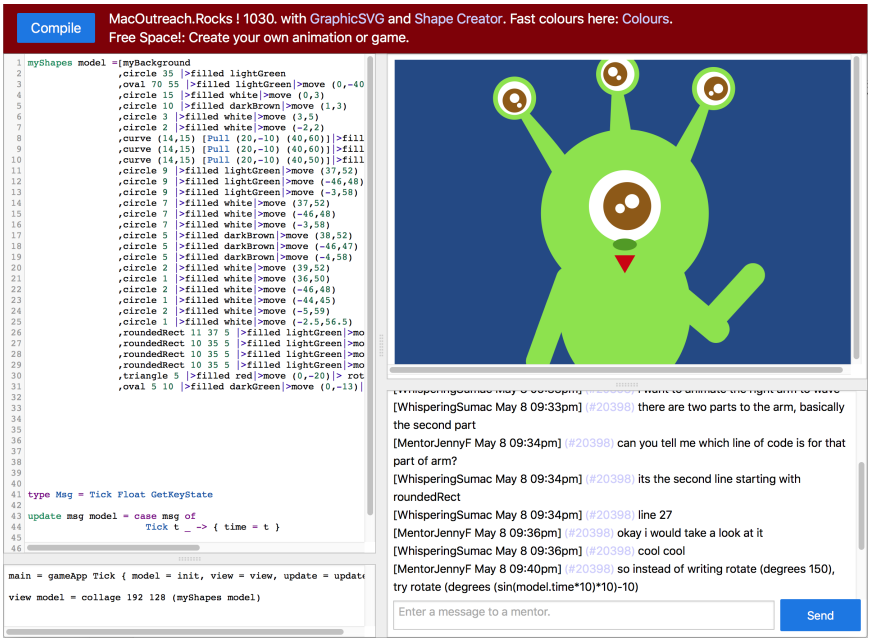


Figure 5: Students are given unique identifiers and can send questions to mentors, which will give mentors access to the student's code. The mentor can reply with suggestions or fixes, and even attach working code that the student can manually incorporate into their version.

3 Discussion and Future Work

Although our workshops initially targeted grades 7 and 8 (ages 12-14), with experience across age ranges, our instructors started expressing a preference for teaching younger children, despite the obvious technical issues (before children learn punctuation in English, they have trouble paying attention to syntax in Elm), which make it harder to advance to the concepts like functions and state. Younger children seemed more open to discovery, and unperturbed by making mistakes. In her PhD thesis, [16], analyzing four early approaches to integrating computers into schools, Solomon provides a possible explanation. She describes the evolution of the four approaches as growing organically from differing assumptions about educational psychology. The most revolutionary, the Logo school, draws on Piaget (as interpreted by Papert) who tells them that children are natural scientists, discovering the rules of their world by bump-

Room	Messages	Last Poster	Answered	Last Message
MentorTonyS-1000	2	MentorKareemK	yes	2017-05-04 21:13:58
StrongReed-1000	2	MentorTonyS	yes	2017-05-04 20:41:13
BusyZebra-1000	2	MentorTonyS	yes	2017-05-04 20:39:59
RemarkableNectarine-1000	8	MentorKareemK	yes	2017-05-04 18:06:14
ResponsibleNigella-1000	9	MentorKareemK	yes	2017-05-04 18:01:55
EnergeticChimpanzee-1000	4	MentorKareemK	yes	2017-05-04 17:49:14
AdventurousAardvark-1000	9	MentorKareemK	yes	2017-05-04 17:47:15
EncouragingChicory-1000	11	MentorKareemK	yes	2017-05-04 17:45:12

Figure 6: Mentors have access to a sortable list of questions posed by students as part of our new distance learning approach.

ing up against it, and later by observing an interacting with older children and adults. But at some point, organic experience of mathematics fails to motivate the learning of arithmetic, and learning objectives must be imposed, which unfortunately turns off the instinct to make numbers their own. Papert wanted to extend this intrinsic phase of learning by providing an environment for mathematical exploration through programming. This vision extended to creating physical simulations with alternative laws of physics, and seeing which laws matched experience. Unfortunately, the range of Papert's vision has not been taken up by current practitioners. Perhaps this is because the "Computational Thinking" school sees teaching sequence and selection of actions as the end goal. Perhaps Papert's dislike for contemporary empirical methods in education research, [21], did not win over decision makers.

So, if a half century on we are to take up the banner and advocate for the child as scientist, why should we be more successful?

Measurement Whereas Logo took children out of school mathematics into Turtle geometry, this made measurement harder. Specific knowledge of coordinate geometry will be easy to measure, and we hope that transfer to algebra will also be measurable.

Better Hardware Whereas Logo started in all caps, because that was what contemporary terminals supported, we have touch interfaces and more processing power than a 1970s' supercomputer.

Better Languages Logo was designed when structured programming was just beginning to be understood. We know a lot more now, especially about types.

While others projects are taking advantage of power to wrap their interfaces in child-friendly graphics, we hope to get a lot farther using programming languages best practices.

3.1 Touch Editing

We have done workshops where some students have written Elm programs on iPads and productivity is obviously lower. All of the limitations of tablets are highlighted, and none of the advantages. Given the prevalence of iPads in younger grades—with some schools phasing out desktop and laptop computers entirely—we knew we had to do better. There are many examples to guide us: successful non-textual editors (Scratch, Hopscotch, Lightbot, etc.); early experiments in user-directed program transforming editors: "by precluding the creation of syntactically incorrect files, the Synthesizer lets the user focus on the intellectually challenging aspects of programming" [17]; and "autocompletion" in contemporary IDEs.

In rethinking the division of labour between programmer and computer, it seems obvious that there is no advantage in providing hints to the programmer which are later found by the compiler to cause type errors. To avoid type errors, instead of editing text, our users do surgery on typed Abstract Syntax Trees (ASTs), via touch-based interaction with the textual representation of the AST. Interesting points about our implementation:

- Syntax highlighting helps readability and guides the user to create syntactically correct programs. Our programs are always correct and typed, so we can use colour for type, with a unique colour for every type. Holes are typed, and have default values so partially complete programs can be visualized. Figure 7.
- Program transformations are supplied by the "editor". This is the hard part. While strong typing restricts substitutions of values and fully applied functions to reasonable numbers (when compared to the tens and hundreds of autocompletions offered by conventional editors), beginners need more structure so they can learn in layers. We are working out the presentation by trial and error as we support increasingly complicated transformations. Figure 8.

This editor is still in development, but we have tested it with a split Grade 2/3 class (ages 7 and 8) and within half an hour they were productively creating graphics with moving shapes. Some Grade 7/8 students were able to effectively use variables within an hour.

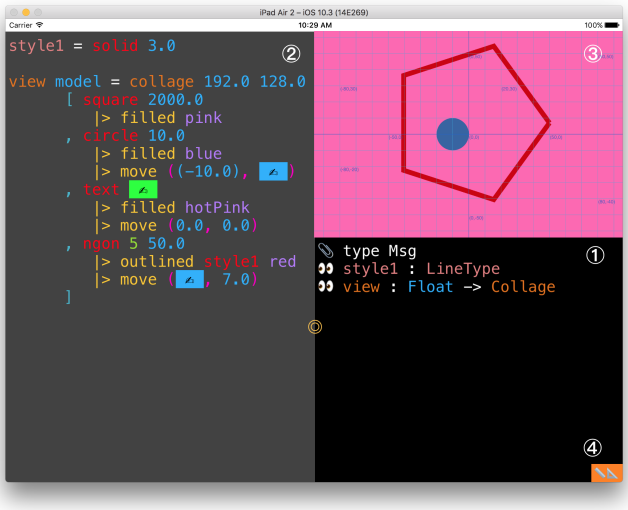


Figure 7: The prototype app consists of three resizable panes: (1) is a list of definitions in the user's code, the visibility of which can be toggled on or off. In the main editor pane, (2), code is highlighted according to type. Tapping different elements brings up context-specific panes that allow the insertion of syntactically correct code (shown in Figure 8). The top-right pane (3) is the output generate by the user's code. Finally, the buttons at (4) allow the student to toggle helpful tools such as the Cartesian grid and a bezier curve helper.

3.2 MacVenture

In MacVenture [3], students create their own gamebook with nodes (places) and edges (ways). This game arose out of a desire to make graph structures interesting to children. Each place has a textual

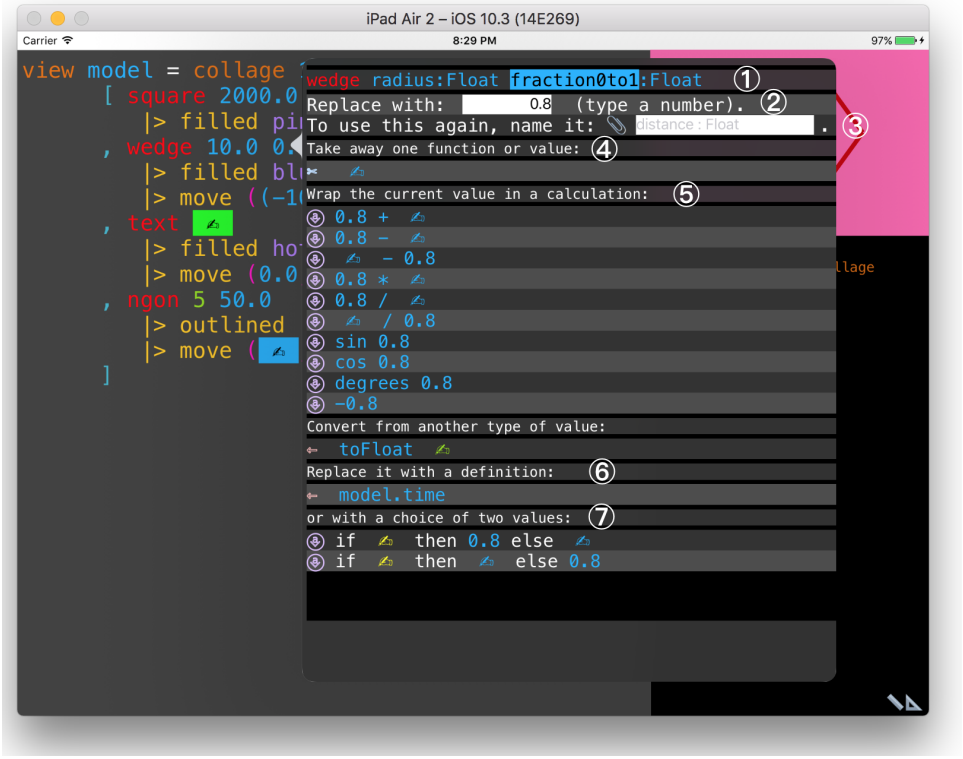


Figure 8: Context- and Type-Aware Transformations: After tapping on an element, a popup gives the user options to modify the value. The signature of the function using the element and position of the selected argument is shown at the top (1). The user can type in a constant in (2), or make a new definition out of the selected value (3). This is the only time that the user types the variable name, so there are no concerns about misspelled variables. There is then the option to remove the current element (for example, an element from a list, or a transformer application), or replace the value with a hole (4). Calculations incorporating the current value follow (5), as well as the definitions which already are in the context and match types (6). In this example, there is only `model.time`, which is an record element of the `model` argument, but there could also be user-defined variables (3). Finally, there are conditional options (7).

description, including textual keys, and ways can be annotated with matching locks. Currently, students also choose images for each place, but this frequently causes network problems, and makes sharing (as demanded by students and teachers) impractical. Incorporating touch editing for Elm graphics to replace the images will solve these problems.

3.3 Spatial Awareness

Another possible explanation for the relative success of our approach is the reported connection between spatial awareness and programming achievement. Cooper *et al* [5] found a strong correlation between spatial awareness and programming achievement in high school students as measured by an AP CS exam. Even while taking time out of programming instruction, spatial awareness training increased programming performance. Spatial awareness and two-dimensional drawing are not exactly the same skills, but this is an interesting connection which should be investigated.

We thank the Dean of Engineering, NSERC PromoScience, and Google igniteCS for funding, our many teacher partners for their advice and support over the years, and the many little scientists whose boundless enthusiasm is so infectious.

4 Bibliography

References

[1] (2013): *National curriculum in England: computing programmes of study*. Technical Report, UK Department of Education. Available at `https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study`.

[2] (2016): *A Farewell to FRP*. Available at `http://elm-lang.org/blog/farewell-to-frp`.

[3] Helen Brown (2016): *MacVenture: An iPad Application Design for Social Constructivist E-Learning*. Master's thesis, McMaster University, http://hdl.handle.net/11375/20478.

[4] Kathleen F. Clark & Michael F. Graves (2005): *Scaffolding Students' Comprehension of Text*. *The Reading Teacher* 58(6), pp. 570–580, doi:10.1598/RT.58.6.6. Available at `http://dx.doi.org/10.1598/RT.58.6.6`.

[5] Stephen Cooper, Karen Wang, Maya Israni & Sheryl Sorby (2015): *Spatial Skills Training in Introductory Computing*. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, ACM, New York, NY, USA, pp. 13–20, doi:10.1145/2787622.2787728. Available at `http://doi.acm.org/10.1145/2787622.2787728`.

[6] Evan Czaplicki (2012): *Elm: Concurrent FRP for Functional GUIs*. *Senior thesis, Harvard University*.

[7] Kathi Fisler (2014): *The Recurring Rainfall Problem*. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, ACM, New York, NY, USA, pp. 35–42, doi:10.1145/2632320.2632346. Available at `http://doi.acm.org/10.1145/2632320.2632346`.

[8] Carl Benedikt Frey & Michael A. Osborne (2017): *The future of employment: How susceptible are jobs to computerisation?* *Technological Forecasting and Social Change* 114, pp. 254 – 280, doi:https://doi.org/10.1016/j.techfore.2016.08.019. Available at `http://www.sciencedirect.com/science/article/pii/S0040162516302244`.

[9] Mark Guzdial (2015): *Learner-Centered Design of Computing Education: Research on Computing for Everyone*. *Synthesis Lectures on Human-Centered Informatics* 8(6), pp. 1–165,

doi:10.2200/S00684ED1V01Y201511HCI033. Available at http://dx.doi.org/10.2200/S00684ED1V01Y201511HCI033.

[10] J. Hughes (1989): *Why Functional Programming Matters*. The Computer Journal 32(2), p. 98, doi:10.1093/comjnl/32.2.98. Available at +http://dx.doi.org/10.1093/comjnl/32.2.98.

[11] Johan Jeuring & Jay McCarthy, editors (2016): *Proceedings of the 4th and 5th International Workshop on Trends in Functional Programming in Education, TFPIE 2016, Sophia-Antipolis, France and University of Maryland College Park, USA, 2nd June 2015 and 7th June 2016*. *EPTCS* 230. Available at https://doi.org/10.4204/EPTCS.230.

[12] Carolyn Kieran (2004): *Algebraic thinking in the early grades: What is it*. The Mathematics Educator 8(1), pp. 139–151.

[13] Valerie E. Lee & David T. Burkam (2003): *Dropping Out of High School: The Role of School Organization and Structure*. American Educational Research Journal 40(2), pp. 353–393, doi:10.3102/00028312040002353. Available at http://dx.doi.org/10.3102/00028312040002353.

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

[15] Elizabeth Schofield, Michael Erlinger & Zachary Dodds (2014): *MyCS: CS for Middle-years Students and Their Teachers*. In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, ACM, New York, NY, USA, pp. 337–342, doi:10.1145/2538862.2538901. Available at http://doi.acm.org/10.1145/2538862.2538901.

[16] Cynthia Solomon (1988): *Computer environments for children: A reflection on theories of learning and education*. MIT press.

[17] Tim Teitelbaum & Thomas Reps (1981): *The Cornell Program Synthesizer: A Syntax-directed Programming Environment*. *Commun. ACM* 24(9), pp. 563–573, doi:10.1145/358746.358755. Available at http://doi.acm.org/10.1145/358746.358755.

[18] Annette Vee (2013): *Understanding Computer Programming as a Literacy*. Literacy in Composition Studies 1(2). Available at http://www.licsjournal.org/OJS/index.php/LiCS/article/view/24.

[19] Scott N. Walck (2016): *Learn Quantum Mechanics with Haskell*. In Jeuring & McCarthy [11], pp. 31–46. Available at https://doi.org/10.4204/EPTCS.230.3.

[20] Jeannette M. Wing (2006): *Computational Thinking*. *Commun. ACM* 49(3), pp. 33–35, doi:10.1145/1118178.1118215. Available at http://doi.acm.org/10.1145/1118178.1118215.

[21] 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.